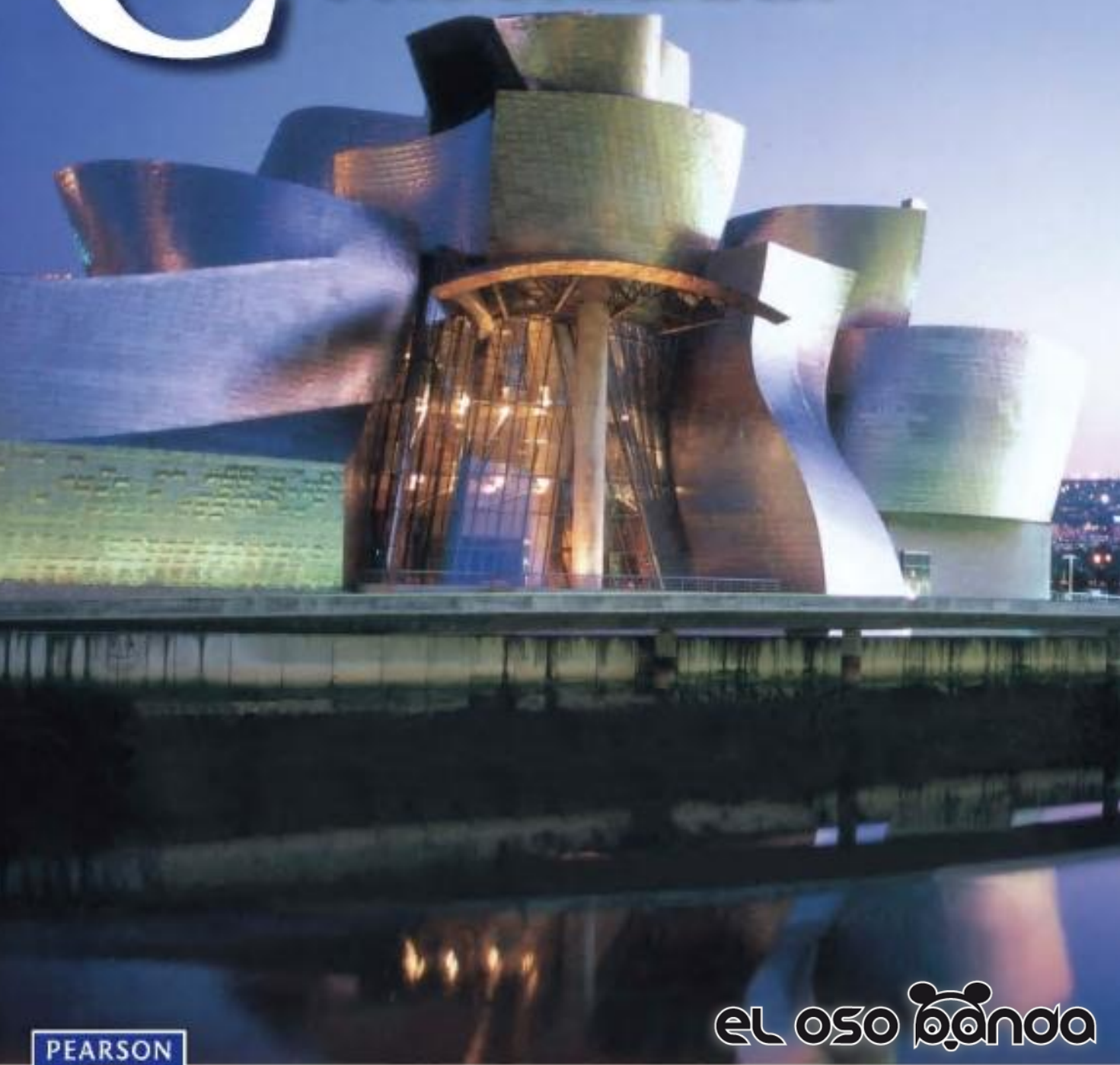


C#

Segunda edición

CÓMO PROGRAMAR



el oso **bánda**



HARVEY M. DEITEL y PAUL J. DEITEL

CUARTA EDICIÓN

CÓMO PROGRAMAR EN

C#

Harvey M. Deitel
Deitel & Associates, Inc.

Paul J. Deitel
Deitel & Associates, Inc.

TRADUCCIÓN

Alfonso Vidal Romero Elizondo
Ingeniero en Electrónica y Comunicación
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

REVISIÓN TÉCNICA

César Coutiño Gómez
Director del Departamento de Computación
Instituto Tecnológico y de Estudios Superiores
de Monterrey
Campus Ciudad de México

José Martín Molina Espinosa
Profesor investigador
Instituto Tecnológico y de Estudios Superiores
de Monterrey
Campus Ciudad de México

Rafael Lozano Espinosa
Profesor del Departamento de Computación
Instituto Tecnológico y de Estudios Superiores
de Monterrey
Campus Ciudad de México

Sergio Fuenlabrada Velázquez
Profesor investigador
Unidad Profesional Interdisciplinaria
de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

Mario Alberto Sesma Martínez
Profesor investigador
Unidad Profesional Interdisciplinaria
de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

Edna Martha Miranda Chávez
Profesora investigadora
Unidad Profesional Interdisciplinaria
de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional



México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

DEITEL, HARVEY M. y PAUL J. DEITEL

Cómo programar en C#. Segunda edición

PEARSON EDUCACIÓN, México 2007

ISBN: 978-970-26-1056-4

Área: Computación

Formato: 20 × 25.5 cm

Páginas: 1080

Authorized translation from the English language edition, entitled *C# for programmers, second edition*, by *Harvey M. Deitel* and *Paul J. Deitel*, published by Pearson Education, Inc., publishing as Prentice Hall, Inc., Copyright ©2006. All rights reserved.
ISBN 0-13-134591-5

Traducción autorizada de la edición en idioma inglés, titulada *C# for programmers, second edition*, por *Harvey M. Deitel* y *Paul J. Deitel*, publicada por Pearson Education, Inc., publicada como Prentice-Hall Inc., Copyright ©2006. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Enrique Trejo Hernández

SEGUNDA EDICIÓN, 2007

D.R. © 2007 por Pearson Educación de México, S.A. de C.V.
Atacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Edo. de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. Núm. 1031.

Prentice Hall es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN 10: 970-26-1056-7

ISBN 13: 978-970-26-1056-4

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 - 10 09 08 07

A Janie Schwark de Microsoft

*Con sincera gratitud por el privilegio y el placer de trabajar
de cerca contigo, durante tantos años.*

Harvey M. Deitel y Paul J. Deitel

Marcas registradas

Microsoft® Visual Studio® 2005 son marcas registradas o marcas registradas por Microsoft Corporation en Estados Unidos y/u otros países.

OMG, Object Management Group, UML, Lenguaje de Modelado Unificado, son marcas registradas de Object Management Group, Inc.

Contenido

Prefacio

xix

I Introducción a las computadoras, Internet y Visual C#

I

1.1	Introducción	2
1.2	Sistema operativo Windows® de Microsoft	2
1.3	C#	3
1.4	Internet y World Wide Web	3
1.5	Lenguaje de Marcado Extensible (XML)	4
1.6	Microsoft .NET	5
1.7	.NET Framework y Common Language Runtime	5
1.8	Prueba de una aplicación en C#	7
1.9	(Única sección obligatoria del caso de estudio) Caso de estudio de ingeniería de software: introducción a la tecnología de objetos y el UML	9
1.10	Conclusión	14
1.11	Recursos Web	14

2 Introducción al IDE de Visual C# 2005 Express Edition

17

2.1	Introducción	18
2.2	Generalidades del IDE de Visual Studio 2005	18
2.3	Barra de menús y barra de herramientas	22
2.4	Navegación por el IDE de Visual Studio 2005	25
2.4.1	Explorador de soluciones	25
2.4.2	Cuadro de herramientas	29
2.4.3	Ventana Propiedades	30
2.5	Uso de la Ayuda	31
2.6	Uso de la programación visual para crear un programa simple que muestra texto e imagen	34
2.7	Conclusión	45
2.8	Recursos Web	46

3 Introducción a las aplicaciones de C#

47

3.1	Introducción	48
3.2	Una aplicación simple en C#: mostrar una línea de texto	48
3.3	Cómo crear una aplicación simple en Visual C# Express	53
3.4	Modificación de su aplicación simple en C#	60
3.5	Formato del texto con Console.Write y Console.WriteLine	61
3.6	Otra aplicación en C#: suma de enteros	63
3.7	Conceptos sobre memoria	66

3.8	Aritmética	67
3.9	Toma de decisiones: operadores de igualdad y relacionales	69
3.10	(Opcional) Caso de estudio de ingeniería de software: análisis del documento de requerimientos del ATM	74
3.11	Conclusión	82

4 Introducción a las clases y los objetos 83

4.1	Introducción	84
4.2	Clases, objetos, métodos, propiedades y variables de instancia	84
4.3	Declaración de una clase con un método e instanciamiento del objeto de una clase	86
4.4	Declaración de un método con un parámetro	89
4.5	Variables de instancia y propiedades	92
4.6	Diagrama de clases de UML con una propiedad	96
4.7	Ingeniería de software con propiedades y los descriptores de acceso <code>set</code> y <code>get</code>	97
4.8	Comparación entre tipos por valor y tipos por referencia	98
4.9	Inicialización de objetos con constructores	99
4.10	Los números de punto flotante y el tipo <code>decimal</code>	102
4.11	(Opcional) Caso de estudio de ingeniería de software: identificación de las clases en el documento de requerimientos del ATM	107
4.12	Conclusión	114

5 Instrucciones de control: parte I 115

5.1	Introducción	116
5.2	Estructuras de control	116
5.3	Instrucción de selección simple <code>if</code>	118
5.4	Instrucción de selección doble <code>if...else</code>	119
5.5	Instrucción de repetición <code>while</code>	122
5.6	Cómo formular algoritmos: repetición controlada por un contador	123
5.7	Cómo formular algoritmos: repetición controlada por un centinela	127
5.8	Cómo formular algoritmos: instrucciones de control anidadas	130
5.9	Operadores de asignación compuestos	134
5.10	Operadores de incremento y decremento	134
5.11	Tipos simples	137
5.12	(Opcional) Caso de estudio de ingeniería de software: identificación de los atributos de las clases en el sistema ATM	137
5.13	Conclusión	141

6 Instrucciones de control: parte 2 143

6.1	Introducción	144
6.2	Fundamentos de la repetición controlada por un contador	144
6.3	Instrucción de repetición <code>for</code>	145
6.4	Ejemplos acerca del uso de la instrucción <code>for</code>	149
6.5	Instrucción de repetición <code>do...while</code>	153
6.6	Instrucción de selección múltiple <code>switch</code>	155
6.7	Instrucciones <code>break</code> y <code>continue</code>	162
6.8	Operadores lógicos	164
6.9	(Opcional) Caso de estudio de ingeniería de software: identificación de los estados y actividades de los objetos en el sistema ATM	168
6.10	Conclusión	173

7	Métodos: un análisis más detallado	175
7.1	Introducción	176
7.2	Empaquetamiento de código en C#	176
7.3	Métodos <code>static</code> , variables <code>static</code> y la clase <code>Math</code>	177
7.4	Declaración de métodos con múltiples parámetros	179
7.5	Notas acerca de cómo declarar y utilizar los métodos	183
7.6	La pila de llamadas a los métodos y los registros de activación	184
7.7	Promoción y conversión de argumentos	184
7.8	La Biblioteca de clases del .NET Framework	186
7.9	Caso de estudio: generación de números aleatorios	187
7.9.1	Escalar y desplazar números aleatorios	191
7.9.2	Repetitividad de números aleatorios para prueba y depuración	191
7.10	Caso de estudio: juego de probabilidad (introducción a las enumeraciones)	192
7.11	Alcance de las declaraciones	196
7.12	Sobrecarga de métodos	198
7.13	Recursividad	200
7.14	Paso de argumentos: diferencia entre paso por valor y paso por referencia	204
7.15	(Opcional) Caso de estudio de ingeniería de software: identificación de las operaciones de las clases en el sistema ATM	207
7.16	Conclusión	213
8	Arreglos	215
8.1	Introducción	216
8.2	Arreglos	216
8.3	Declaración y creación de arreglos	217
8.4	Ejemplos acerca del uso de los arreglos	218
8.5	Caso de estudio: simulación para barajar y repartir cartas	226
8.6	Instrucción <code>foreach</code>	230
8.7	Paso de arreglos y elementos de arreglos a los métodos	231
8.8	Paso de arreglos por valor y por referencia	233
8.9	Caso de estudio: la clase <code>LibroCalificaciones</code> que usa un arreglo para ordenar calificaciones	237
8.10	Arreglos multidimensionales	242
8.11	Caso de estudio: la clase <code>LibroCalificaciones</code> que usa un arreglo rectangular	246
8.12	Listas de argumentos de longitud variable	250
8.13	Uso de argumentos de línea de comandos	253
8.14	(Opcional) Caso de estudio de ingeniería de software: colaboración entre los objetos en el sistema ATM	254
8.15	Conclusión	260
9	Clases y objetos: un análisis más detallado	263
9.1	Introducción	264
9.2	Caso de estudio de la clase <code>Tiempo</code>	264
9.3	Control del acceso a los miembros	268
9.4	Referencias a los miembros del objeto actual mediante <code>this</code>	269
9.5	Indexadores	271
9.6	Caso de estudio de la clase <code>Tiempo</code> : constructores sobrecargados	274
9.7	Constructores predeterminados y sin parámetros	279
9.8	Composición	279
9.9	Recolección de basura y destructores	282
9.10	Miembros de clase <code>static</code>	283

9.11	Variables de instancia <code>readonly</code>	288
9.12	Reutilización de software	290
9.13	Abstracción de datos y encapsulamiento	291
9.14	Caso de estudio de la clase <code>Tiempo</code> : creación de bibliotecas de clases	292
9.15	Acceso <code>internal</code>	295
9.16	Vista de clases y Examinador de objetos	297
9.17	(Opcional) Caso de estudio de ingeniería de software: inicio de la programación de las clases del sistema ATM	298
9.18	Conclusión	304

10 Programación orientada a objetos: herencia 305

10.1	Introducción	306
10.2	Clases base y clases derivadas	307
10.3	Miembros <code>protected</code>	309
10.4	Relación entre las clases base y las clases derivadas	309
10.4.1	Creación y uso de una clase <code>EmpleadoPorComision</code>	310
10.4.2	Creación de una clase <code>EmpleadoBaseMasComision</code> sin usar la herencia	314
10.4.3	Creación de una jerarquía de herencia <code>EmpleadoPorComision-EmpleadoBaseMasComision</code>	319
10.4.4	La jerarquía de herencia <code>EmpleadoPorComision-EmpleadoBaseMasComision</code> mediante el uso de variables de instancia <code>protected</code>	321
10.4.5	La jerarquía de herencia <code>EmpleadoPorComision-EmpleadoBaseMasComision</code> mediante el uso de variables de instancia <code>private</code>	326
10.5	Los constructores en las clases derivadas	331
10.6	Ingeniería de software mediante la herencia	336
10.7	La clase <code>object</code>	337
10.8	Conclusión	339

11 Polimorfismo, interfaces y sobrecarga de operadores 341

11.1	Introducción	342
11.2	Ejemplos de polimorfismo	343
11.3	Demostración del comportamiento polimórfico	344
11.4	Clases y métodos abstractos	347
11.5	Caso de estudio: sistema de nómina utilizando polimorfismo	349
11.5.1	Creación de la clase base abstracta <code>Empleado</code>	350
11.5.2	Creación de la clase derivada concreta <code>EmpleadoAsalariado</code>	352
11.5.3	Creación de la clase derivada concreta <code>EmpleadoPorHoras</code>	353
11.5.4	Creación de la clase derivada concreta <code>EmpleadoPorComision</code>	355
11.5.5	Creación de la clase derivada concreta indirecta <code>EmpleadoBaseMasComision</code>	356
11.5.6	El procesamiento polimórfico, el operador <code>is</code> y la conversión descendente	357
11.5.7	Resumen de las asignaciones permitidas entre variables de la clase base y de la clase derivada	362
11.6	Métodos y clases <code>sealed</code>	362
11.7	Caso de estudio: creación y uso de interfaces	363
11.7.1	Desarrollo de una jerarquía <code>IPorPagar</code>	364
11.7.2	Declaración de la interfaz <code>IPorPagar</code>	365
11.7.3	Creación de la clase <code>Factura</code>	365
11.7.4	Modificación de la clase <code>Empleado</code> para implementar la interfaz <code>IPorPagar</code>	367
11.7.5	Modificación de la clase <code>EmpleadoAsalariado</code> para usarla en la jerarquía <code>IPorPagar</code>	369
11.7.6	Uso de la interfaz <code>IPorPagar</code> para procesar objetos <code>Factura</code> y <code>Empleado</code> mediante el polimorfismo	370

11.7.7	Interfaces comunes de la Biblioteca de clases del .NET Framework	372
11.8	Sobrecarga de operadores	372
11.9	(Opcional) Caso de estudio de ingeniería de software: incorporación de la herencia y el polimorfismo en el sistema ATM	376
11.10	Conclusión	383

12 Manejo de excepciones 385

12.1	Introducción	386
12.2	Generalidades acerca del manejo de excepciones	387
12.3	Ejemplo: división entre cero sin manejo de excepciones	387
12.4	Ejemplo: manejo de las excepciones <code>DivideByZeroException</code> y <code>FormatException</code>	390
12.4.1	Encerrar código en un bloque <code>try</code>	392
12.4.2	Atrapar excepciones	392
12.4.3	Excepciones no atrapadas	392
12.4.4	Modelo de terminación del manejo de excepciones	392
12.4.5	Flujo de control cuando ocurren las excepciones	394
12.5	Jerarquía <code>Exception</code> de .NET	394
12.5.1	Las clases <code>ApplicationException</code> y <code>SystemException</code>	394
12.5.2	Determinar cuáles excepciones debe lanzar un método	395
12.6	El bloque <code>finally</code>	395
12.7	Propiedades de <code>Exception</code>	402
12.8	Clases de excepciones definidas por el usuario	406
12.9	Conclusión	409

13 Conceptos de interfaz gráfica de usuario: parte I 411

13.1	Introducción	412
13.2	Formularios Windows Forms	413
13.3	Manejo de eventos	415
13.3.1	Una GUI simple controlada por eventos	416
13.3.2	Otro vistazo al código generado por Visual Studio	417
13.3.3	Delegados y el mecanismo de manejo de eventos	419
13.3.4	Otras formas de crear manejadores de eventos	419
13.3.5	Localización de la información de los eventos	420
13.4	Propiedades y distribución de los controles	420
13.5	Controles <code>Label</code> , <code>TextBox</code> y <code>Button</code>	425
13.6	Controles <code>GroupBox</code> y <code>Panel</code>	428
13.7	Controles <code>CheckBox</code> y <code>RadioButton</code>	430
13.8	Controles <code>PictureBox</code>	438
13.9	Controles <code>ToolTip</code>	440
13.10	Control <code>NumericUpDown</code>	442
13.11	Manejo de los eventos del ratón	444
13.12	Manejo de los eventos del teclado	446
13.13	Conclusión	449

14 Conceptos de interfaz gráfica de usuario: parte 2 451

14.1	Introducción	452
14.2	Menús	452
14.3	Control <code>MonthCalendar</code>	461
14.4	Control <code>DateTimePicker</code>	461

14.5	Control <code>LinkLabel</code>	465
14.6	Control <code>ListBox</code>	469
14.7	Control <code>CheckedListBox</code>	473
14.8	Control <code>ComboBox</code>	474
14.9	Control <code>TreeView</code>	479
14.10	Control <code>ListView</code>	484
14.11	Control <code>TabControl</code>	489
14.12	Ventanas de la interfaz de múltiples documentos (MDI)	494
14.13	Herencia visual	500
14.14	Controles definidos por el usuario	503
14.15	Conclusión	507

15 Subprocesamiento múltiple **509**

15.1	Introducción	510
15.2	Estados de los subprocesos: ciclo de vida de un subproceso	511
15.3	Prioridades y programación de subprocesos	513
15.4	Creación y ejecución de subprocesos	514
15.5	Sincronización de subprocesos y la clase <code>Monitor</code>	517
15.6	Relación productor/consumidor sin sincronización de subprocesos	519
15.7	Relación productor/consumidor con sincronización de subprocesos	525
15.8	Relación productor/consumidor: búfer circular	532
15.9	Subprocesamiento múltiple con GUIs	539
15.10	Conclusión	543

16 Cadenas, caracteres y expresiones regulares **545**

16.1	Introducción	546
16.2	Fundamentos de los caracteres y las cadenas	547
16.3	Constructores de <code>string</code>	547
16.4	Indexador de <code>string</code> , propiedad <code>Length</code> y método <code>CopyTo</code>	549
16.5	Comparación de objetos <code>string</code>	550
16.6	Localización de caracteres y subcadenas en objetos <code>string</code>	553
16.7	Extracción de subcadenas de objetos <code>string</code>	555
16.8	Concatenación de objetos <code>string</code>	556
16.9	Métodos varios de la clase <code>string</code>	557
16.10	La clase <code>StringBuilder</code>	558
16.11	Las propiedades <code>Length</code> y <code>Capacity</code> , el método <code>EnsureCapacity</code> y el indexador de la clase <code>StringBuilder</code>	558
16.12	Los métodos <code>Append</code> y <code>AppendFormat</code> de la clase <code>StringBuilder</code>	560
16.13	Los métodos <code>Insert</code> , <code>Remove</code> y <code>Replace</code> de la clase <code>StringBuilder</code>	563
16.14	Métodos de <code>char</code>	565
16.15	Simulación para barajar y repartir cartas	567
16.16	Expresiones regulares y la clase <code>Regex</code>	570
16.16.1	Ejemplo de expresión regular	571
16.16.2	Validación de la entrada del usuario con expresiones regulares	573
16.16.3	Los métodos <code>Replace</code> y <code>Split</code> de <code>Regex</code>	577
16.17	Conclusión	579

17 Gráficos y multimedia **581**

17.1	Introducción	582
17.2	Las clases de dibujo y el sistema de coordenadas	582

17.3	Contextos de gráficos y objetos Graphics	584
17.4	Control de los colores	585
17.5	Control de las fuentes	591
17.6	Dibujo de líneas, rectángulos y óvalos	595
17.7	Dibujo de arcos	597
17.8	Dibujo de polígonos y polilíneas	600
17.9	Herramientas de gráficos avanzadas	603
17.10	Introducción a multimedia	608
17.11	Carga, visualización y escalado de imágenes	608
17.12	Animación de una serie de imágenes	610
17.13	Reproductor de Windows Media	619
17.14	Microsoft Agent	621
17.15	Conclusión	632

18 Archivos y flujos 633

18.1	Introducción	634
18.2	Jerarquía de datos	634
18.3	Archivos y flujos	636
18.4	Las clases File y Directory	637
18.5	Creación de un archivo de texto de acceso secuencial	645
18.6	Lectura de datos desde un archivo de texto de acceso secuencial	654
18.7	Serialización	663
18.8	Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos	663
18.9	Lectura y deserialización de datos de un archivo de texto de acceso secuencial	669
18.10	Conclusión	673

19 Lenguaje de marcado extensible (XML) 675

19.1	Introducción	676
19.2	Fundamentos de XML	676
19.3	Estructuración de datos	679
19.4	Espacios de nombres de XML	685
19.5	Definiciones de tipo de documento (DTDs)	688
19.6	Documentos de esquemas XML del W3C	691
19.7	(Opcional) Lenguaje de hojas de estilos extensible y transformaciones XSL	697
19.8	(Opcional) Modelo de objetos de documento (DOM)	705
19.9	(Opcional) Validación de esquemas con la clase XmlReader	717
19.10	(Opcional) XSLT con la clase XsltCompiledTransform	720
19.11	Conclusión	722
19.12	Recursos Web	723

20 Bases de datos, SQL y ADO.NET 725

20.1	Introducción	726
20.2	Bases de datos relacionales	727
20.3	Generalidades acerca de las bases de datos relacionales: la base de datos Libros	728
20.4	SQL	731
20.4.1	Consulta SELECT básica	731
20.4.2	Cláusula WHERE	732
20.4.3	Cláusula ORDER BY	733

20.4.4	Mezcla de datos de varias tablas: INNER JOIN	734
20.4.5	Instrucción INSERT	737
20.4.6	Instrucción UPDATE	737
20.4.7	Instrucción DELETE	738
20.5	Modelo de objetos ADO.NET	739
20.6	Programación con ADO.NET: extraer información de una base de datos	740
20.6.1	Mostrar una tabla de base de datos en un control DataGridView	740
20.6.2	Cómo funciona el enlace de datos	747
20.7	Consulta de la base de datos Libros	751
20.8	Programación con ADO.NET: caso de estudio de libreta de direcciones	759
20.9	Uso de un objeto DataSet para leer y escribir XML	765
20.10	Conclusión	768
20.11	Recursos Web	769

21 ASP.NET 2.0, formularios Web Forms y controles Web **771**

21.1	Introducción	772
21.2	Transacciones HTTP simples	773
21.3	Arquitectura de aplicaciones multinivel	775
21.4	Creación y ejecución de un ejemplo de formulario Web Forms simple	776
21.4.1	Análisis de un archivo ASPX	777
21.4.2	Análisis de un archivo de código subyacente (code-behind)	778
21.4.3	Relación entre un archivo ASPX y un archivo de código subyacente	779
21.4.4	Cómo se ejecuta el código en una página Web ASP.NET	780
21.4.5	Análisis del XHTML generado por una aplicación ASP.NET	780
21.4.6	Creación de una aplicación Web ASP.NET	781
21.5	Controles Web	789
21.5.1	Controles de texto y gráficos	789
21.5.2	Control AdRotator	793
21.5.3	Controles de validación	798
21.6	Rastreo de sesiones	808
21.6.1	Cookies	809
21.6.2	Rastreo de sesiones con HttpSessionState	816
21.7	Caso de estudio: conexión a una base de datos en ASP.NET	824
21.7.1	Creación de un formulario Web Forms que muestra datos de una base de datos	825
21.7.2	Modificación del archivo de código subyacente para la aplicación Libro de visitantes	833
21.8	Caso de estudio: aplicación segura de la base de datos Libros	834
21.8.1	Análisis de la aplicación segura de la base de datos Libros completa	834
21.8.2	Creación de la aplicación segura de la base de datos Libros	838
21.9	Conclusión	861
21.10	Recursos Web	861

22 Servicios Web **863**

22.1	Introducción	864
22.2	Fundamentos de los servicios Web .NET	865
22.2.1	Creación de un servicio Web en Visual Web Developer	866
22.2.2	Descubrimiento de servicios Web	866
22.2.3	Determinación de la funcionalidad de un servicio Web	867
22.2.4	Prueba de los métodos de un servicio Web	868
22.2.5	Creación de un cliente para utilizar un servicio Web	870

22.3	Protocolo simple de acceso a objetos (SOAP)	871
22.4	Publicación y consumo de los servicios Web	872
22.4.1	Definición del servicio Web EnteroEnorme	872
22.4.2	Creación de un servicio Web en Visual Web Developer	877
22.4.3	Despliegue del servicio Web EnteroEnorme	879
22.4.4	Creación de un cliente para consumir el servicio Web EnteroEnorme	880
22.4.5	Consumo del servicio Web EnteroEnorme	883
22.5	Rastreo de sesiones en los servicios Web	887
22.5.1	Creación de un servicio Web de Blackjack	887
22.5.2	Consumo del servicio Web de Blackjack	890
22.6	Uso de formularios Web Forms y servicios Web	899
22.6.1	Agregar componentes de datos a un servicio Web	901
22.6.2	Creación de un formulario Web Forms para interactuar con el servicio Web de reservaciones de una aerolínea	903
22.7	Tipos definidos por el usuario en los servicios Web	906
22.8	Conclusión	915
22.9	Recursos Web	915

23 Redes: sockets basados en flujos y datagramas 917

23.1	Introducción	918
23.2	Comparación entre la comunicación orientada a la conexión y la comunicación sin conexión	918
23.3	Protocolos para transportar datos	919
23.4	Establecimiento de un servidor TCP simple (mediante el uso de sockets de flujo)	919
23.5	Establecimiento de un cliente TCP simple (mediante el uso de sockets de flujo)	921
23.6	Interacción entre cliente/servidor mediante conexiones de sockets de flujo	922
23.7	Interacción entre cliente/servidor sin conexión mediante datagramas	931
23.8	Juego de Tres en raya cliente/servidor mediante el uso de un servidor con subprocesamiento múltiple	936
23.9	Control WebBrowser	948
23.10	.NET Remoting	951
23.11	Conclusión	961

24 Estructuras de datos 963

24.1	Introducción	964
24.2	Tipos struct simples, boxing y unboxing	964
24.3	Clases autorreferenciadas	965
24.4	Listas enlazadas	967
24.5	Pilas	977
24.6	Colas	980
24.7	Árboles	984
24.7.1	Árbol binario de búsqueda de valores enteros	985
24.7.2	Árbol binario de búsqueda de objetos IComparable	991
24.8	Conclusión	997

25 Genéricos 999

25.1	Introducción	1000
25.2	Motivación para los métodos genéricos	1001
25.3	Implementación de un método genérico	1002

25.4	Restricciones de los tipos	1005
25.5	Sobrecarga de métodos genéricos	1007
25.6	Clases genéricas	1007
25.7	Observaciones acerca de los genéricos y la herencia	1016
25.8	Conclusión	1016

26 Colecciones **1017**

26.1	Introducción	1018
26.2	Generalidades acerca de las colecciones	1018
26.3	La clase <code>Array</code> y los enumeradores	1020
26.4	Colecciones no genéricas	1024
26.4.1	La clase <code>ArrayList</code>	1024
26.4.2	La clase <code>Stack</code>	1027
26.4.3	La clase <code>Hashtable</code>	1030
26.5	Colecciones genéricas	1034
26.5.1	La clase genérica <code>SortedDictionary</code>	1034
26.5.2	La clase genérica <code>LinkedList</code>	1036
26.6	Colecciones sincronizadas	1040
26.7	Conclusión	1040

Apéndices

(Incluidos en el CD-ROM que acompaña al libro)

A Tabla de precedencia de los operadores **1041**

B Sistemas numéricos **1043**

B.1	Introducción	1044
B.2	Abreviatura de los números binarios como números octales y hexadecimales	1046
B.3	Conversión de números octales y hexadecimales a binarios	1047
B.4	Conversión de un número binario, octal o hexadecimal a decimal	1048
B.5	Conversión de un número decimal a binario, octal o hexadecimal	1048
B.6	Números binarios negativos: notación de complemento a dos	1050

C Uso del depurador de Visual Studio® 2005 **1051**

C.1	Introducción	1052
C.2	Los puntos de interrupción y el comando Continuar	1052
C.3	Las ventanas Variables locales e Inspección	1057
C.4	Control de la ejecución mediante los comandos Paso a paso por instrucciones , Paso a paso por procedimientos , Paso a paso para salir y Continuar	1059
C.5	Otras características	1062
C.5.1	Editar y Continuar	1062
C.5.2	Ayudante de excepciones	1065
C.5.3	Depuración Sólo mi código (Just My Code™)	1065
C.5.4	Otras nuevas características del depurador	1065
C.6	Conclusión	1066

D Conjunto de caracteres ASCII **1067**

E	Unicode®	1069
E.1	Introducción	1070
E.2	Formatos de transformación de Unicode	1070
E.3	Caracteres y glifos	1071
E.4	Ventajas y desventajas de Unicode	1072
E.5	Uso de Unicode	1072
E.6	Rangos de caracteres	1074
F	Introducción a XHTML: parte 1	1077
F.1	Introducción	1078
F.2	Edición de XHTML	1078
F.3	El primer ejemplo en XHTML	1079
F.4	Servicio de validación de XHTML del W3C	1081
F.5	Encabezados	1083
F.6	Vínculos	1084
F.7	Imágenes	1086
F.8	Caracteres especiales y más saltos de línea	1090
F.9	Listas desordenadas	1091
F.10	Listas anidadas y ordenadas	1092
F.11	Recursos Web	1094
G	Introducción a XHTML: parte 2	1095
G.1	Introducción	1096
G.2	Tablas básicas en XHTML	1096
G.3	Tablas intermedias en XHTML y formatos	1098
G.4	Formularios básicos en XHTML	1101
G.5	Formularios más complejos en XHTML	1103
G.6	Vínculos internos	1110
G.7	Creación y uso de mapas de imágenes	1112
G.8	Elementos meta	1115
G.9	Elemento frameset	1116
G.10	Elementos frameset anidados	1120
G.11	Recursos Web	1122
H	Caracteres especiales de HTML/XHTML	1123
I	Colores en HTML/XHTML	1125
J	Código del caso de estudio del ATM	1129
J.1	Implementación del caso de estudio del ATM	1129
J.2	La clase ATM	1130
J.3	La clase Pantalla	1135
J.4	La clase Teclado	1135
J.5	La clase DispensadorEfectivo	1136
J.6	La clase RanuraDeposito	1137

J.7	La clase Cuenta	1138
J.8	La clase BaseDatosBanco	1140
J.9	La clase Transaccion	1142
J.10	La clase SolicitudSaldo	1144
J.11	La clase Retiro	1144
J.12	La clase Deposito	1148
J.13	La clase CasoEstudioATM	1150
J.14	Conclusión	1151

K UML 2: tipos de diagramas adicionales 1153

K.1	Introducción	1153
K.2	Tipos de diagramas adicionales	1153

L Los tipos simples 1155

Índice 1157

Prefacio

“Ya no vivas en fragmentos, conéctate.”

—Edgar Morgan Foster

Bienvenido a C# y al mundo de Windows, Internet y la programación Web con Visual Studio 2005 y la plataforma .NET 2.0. En este libro se presentan las tecnologías de computación más recientes e innovadoras para los desarrolladores de software y profesionales de TI.

En Deitel & Associates escribimos libros de texto de ciencias computacionales para estudiantes universitarios, y libros profesionales para desarrolladores de software. Además, utilizamos este material para impartir seminarios industriales a nivel mundial.

Fue muy divertido crear este libro. Para empezar, analizamos cuidadosamente la edición anterior:

- Auditamos nuestra presentación de C# en comparación con las especificaciones más recientes del lenguaje C# de Microsoft y Ecma, las cuales se encuentran en los sitios msdn.microsoft.com/vcsharp/programming/language y www.ecma-international.org/publications/standards/Ecma-334.html, respectivamente.
- Hemos actualizado y mejorado en forma considerable todos los capítulos.
- Cambiamos de pedagogía y ahora se analizarán antes las clases y los objetos. Es decir, podrá crear clases reutilizables a partir del capítulo 4.
- Actualizamos nuestra presentación orientada a objetos para acoger la versión más reciente del *UML (Lenguaje Unificado de Modelado)*: *UML™ 2.0*, el lenguaje gráfico estándar en la industria para modelar sistemas orientados a objetos.
- En los capítulos 1, 3 a 9 y 11 agregamos un caso de estudio sobre un cajero automático (ATM) de DOO/UML. Este caso de estudio incluye una implementación completa del ATM en código de C#.
- Agregamos diversos casos de estudio de programación orientada a objetos con varias secciones.
- Incorporamos nuevas características clave de la versión más reciente de C# de Microsoft: Visual C# 2005; además, agregamos análisis sobre los elementos genéricos, el trabajo remoto y la depuración en .NET.
- Mejoramos de manera considerable el manejo sobre XML, ADO.NET, ASP.NET y los servicios Web.

Todo esto ha sido revisado con detalle por un equipo de distinguidos desarrolladores de .NET en la industria, profesionales académicos y miembros del equipo de desarrollo de Microsoft C#.

Quién debe leer este libro

Tenemos varias publicaciones acerca de C#, para distintas audiencias.

Cómo programar en C#, 2ª edición forma parte de la *Serie Deitel® para desarrolladores*, diseñada para desarrolladores profesionales de software que desean analizar a profundidad la nueva tecnología, sin dedicar demasiado tiempo al material introductorio. Este libro se concentra en lograr una claridad en los programas a través de las técnicas demostradas: programación estructurada, programación orientada a objetos (POO) y programación controlada por eventos. Continúa con temas de nivel superior como XML, ASP.NET 2.0, ADO.NET 2.0 y los servicios Web.

Cómo programar en C#, 2ª edición presenta muchos programas completos y funcionales en C#, y describe sus entradas y salidas mediante capturas de pantalla reales de los programas en ejecución. Éste es nuestro método distintivo al que llamamos código activo (LIVE-CODE™); presentamos los conceptos dentro del contexto de programas funcionales completos. El código fuente del libro (en inglés) puede descargarlo sin costo de los siguientes sitios Web:

www.deitel.com/books/csharpforprogrammers2/
www.pearsoneducacion.com/deitel

En nuestras instrucciones de prueba del capítulo 1 asumimos que usted extraerá estos ejemplos en la carpeta C:\ de su computadora.

Si surgen dudas o preguntas, a medida que lea este libro, puede enviar un correo electrónico a deitel@deitel.com; le responderemos a la mayor brevedad. Para actualizaciones sobre este libro y el estado del software de C#, asimismo, obtendrá las noticias más recientes acerca de todas las publicaciones y servicios Deitel, visite, con regularidad, el sitio www.deitel.com e inscribise en el boletín de correo electrónico gratuito *Deitel® Buzz Online* en www.deitel.com/newsletter/subscribe.html.

Cómo descargar el software Microsoft Visual C# 2005 Express Edition

Microsoft pone a disposición del público una versión gratuita de su herramienta de desarrollo en C# llamada Visual C# 2005 Express Edition. Puede utilizarla para compilar los programas de ejemplo que vienen en el libro. Para descargar el software Visual C# 2005 Express Edition y Visual Web Developer 2005 Express Edition visite el sitio:

www.microsoft.com/spanish/msdn/vstudio/express/VCS/default.aspx

Microsoft cuenta con un foro dedicado, en idioma inglés, para obtener ayuda acerca del uso del software Express Edition:

forums.microsoft.com/msdn/showForum.aspx?ForumID=24

En nuestro sitio Web www.deitel.com y en nuestro boletín de correo electrónico sin costo www.deitel.com/newsletter/subscribe.html proporcionamos actualizaciones sobre el estado de este software.

Características en *Cómo programar en C#, 2ª edición*

Esta edición contiene muchas características nuevas y mejoradas.

Actualización de Visual Studio 2005, C# 2.0 y .NET 2.0

Actualizamos todo el libro para reflejar la versión más reciente de Microsoft Visual C# 2005. Las nuevas características son:

- Se actualizaron las capturas de pantalla para el IDE Visual Studio 2005.
- Accesorios de propiedades con distintos modificadores.
- Uso del Asistente de excepciones (Exception Assistant) para ver los datos de las excepciones (una nueva característica del depurador de Visual Studio 2005).
- Uso de las técnicas de “arrastrar y soltar” para crear formularios de ventanas enlazados a datos en ADO.NET 2.0.
- Uso de la ventana **Orígenes de datos** (Data Sources) para crear conexiones de datos a nivel de aplicación.
- Uso de la clase `BindingSource` para simplificar el proceso de enlazar controles a un origen de datos subyacente.
- Uso de la clase `BindingNavigator` para habilitar las operaciones simples de navegación, inserción, eliminación y edición de los datos de una base de datos en un formulario de Windows.
- Uso de la página Diseñador de páginas principales para crear una apariencia visual común para las páginas Web de ASP.NET.

- Uso de los menús de etiqueta inteligente (smart tag) de Visual Studio 2005 para realizar muchas de las tareas de programación más comunes cuando se arrastran nuevos controles en un formulario de Windows o en una página Web de ASP.NET.
- Uso del servidor Web integrado de Visual Web Developer para probar las aplicaciones y servicios Web de ASP.NET 2.0.
- Uso de la clase `Xm1DataSource` para enlazar orígenes de datos de XML a un control.
- Uso de la clase `Sq1DataSource` para enlazar una base de datos de SQL Server a un control o a un conjunto de controles.
- Uso de la clase `ObjectDataSource` para enlazar un control a un objeto que sirve como origen de datos.
- Uso del control de inicio de sesión (login) de ASP.NET 2.0 y creación de nuevos controles de usuarios para personalizar el acceso a las aplicaciones Web.
- Uso de genéricos y colecciones genéricas para crear modelos generales de métodos y clases que pueden declararse una vez y utilizarse con muchos tipos de datos.
- Uso de colecciones genéricas del espacio de nombres `Systems.Collections.Generic`.

Nuevo diseño interior

Para rediseñar el estilo interior de nuestros libros trabajamos con el equipo de creativos en Prentice Hall. En respuesta a las peticiones de los lectores, ahora colocamos los términos clave en ***negritas cursivas***, para facilitar su referencia. Enfatizamos los componentes en pantalla mediante el tipo de letra **Helvetica en negrita** (por ejemplo, el menú **Archi-vo**) y el texto de los programas de C# mediante otro tipo de letra **Lucida** (por ejemplo, `int x = 5`).

Sombreado de la sintaxis

Sombreamos la sintaxis de todo el código de C#, de manera similar al código de color de sintaxis que utilizan casi todos los entornos de desarrollo integrados y editores de código de C#. Esto mejora en forma considerable la legibilidad del código; una meta en especial importante, dado que este libro contiene más de 16,800 líneas de código. Nuestras convenciones de sombreado de sintaxis son las siguientes:

los comentarios aparecen en cursiva
las palabras clave aparecen en negrita, cursiva
los errores y los delimitadores de scriptlets de JSP aparecen en color negro y negrita
las constantes y los valores literales aparecen en color gris y negrita
 todos los demás elementos del código aparecen en letra normal y color negro

Resaltado del código

El uso extensivo de resaltado del código facilita a los lectores detectar los segmentos de código que se presentan en cada programa; colocamos rectángulos color gris claro alrededor del código clave.

Método para presentar clases y objetos de manera anticipada

En el capítulo 1 seguimos presentando los conceptos básicos de la tecnología orientada a objetos y la terminología relacionada. En el capítulo 9 de la edición anterior desarrollamos clases personalizadas, pero en esta edición empezamos a hacerlo en el capítulo 4. Hemos reescrito con mucho cuidado los capítulos 5 a 8, con base en nuestro “método de presentación de clases y objetos de manera anticipada”.

El cuidadoso proceso para poner a punto la programación orientada a objetos en los capítulos del 9 al 11

Hemos realizado una actualización muy precisa de *Cómo programar en C#, 2ª edición*. Esta edición es sin duda más clara y accesible; en especial si es la primera vez que trabaja con la programación orientada a objetos (POO). Reescribimos por completo los capítulos sobre la POO, en los cuales integramos un caso de estudio sobre la jerarquía de clases de una nómina de empleados y motivamos las interfaces con una jerarquía de cuentas por pagar.

Casos de estudio

Incluimos muchos casos de estudio, algunos de los cuales abarcan varias secciones y capítulos:

- La clase `RegistroCalificaciones` en los capítulos 4, 5, 6 y 8.
- El sistema ATM opcional de DOO/UML en las secciones de ingeniería de software de los capítulos 1, 3 a 9 y 11.
- La clase `Tiempo` en varias secciones del capítulo 9.
- La aplicación de nómina de `Empleado` en los capítulos 10 y 11.
- La aplicación `LibroVisitantes` de ASP.NET en el capítulo 21.
- La aplicación segura de base de datos de libros de ASP.NET en el capítulo 21.
- El servicio Web de reservación en una aerolínea en el capítulo 22.

Caso de estudio integrado: RegistroCalificaciones

Para reforzar nuestra presentación anticipada de las clases, en los capítulos 4 a 6 y 8 presentamos un caso de estudio integrado en el que se utilizan clases y objetos. En este caso de estudio crearemos una clase llamada `RegistroCalificaciones`, a la que iremos agregando elementos. Esta clase representa el registro de calificaciones de un instructor y realiza varios cálculos con base en un conjunto de calificaciones de los estudiantes: encontrar el promedio, la calificación máxima y la mínima e imprimir un gráfico de barras. Nuestro objetivo es que usted se familiarice con los conceptos importantes de los objetos y las clases, a través de un ejemplo real de una clase sustancial. Desarrollamos esta clase desde cero, construyendo métodos a partir de instrucciones de control y de algoritmos desarrollados con mucho cuidado; se agregaron variables de instancia y arreglos según fue necesario para mejorar la funcionalidad de la clase.

Lenguaje Unificado de Modelado (UML): uso del UML 2.0 para desarrollar un diseño orientado a objetos de un ATM

El Lenguaje Unificado de Modelado (UML™) se ha convertido en el lenguaje de modelado gráfico preferido para diseñar sistemas orientados a objetos. Todos los diagramas de UML en el libro cumplen con la especificación UML 2.0. Utilizamos los diagramas de clases de UML para representar visualmente las clases y sus relaciones de herencia, además, usamos los diagramas de actividades para demostrar el flujo de control en cada una de las diversas instrucciones de control de C#.

Esta *segunda edición* incluye un nuevo caso de estudio opcional (pero que recomendamos ampliamente) sobre el diseño orientado a objetos mediante el uso de UML, y su revisión estuvo a cargo de un distinguido equipo de profesionales académicos y de la industria relacionados con el DOO/UML, incluyendo líderes en el campo por parte de Rational (los creadores del UML, que ahora son una división de IBM) y el Grupo de Administración de Objetos (responsable del mantenimiento y la evolución del UML). En este caso de estudio diseñamos e implementamos en su totalidad el software para un cajero automático (ATM) simple. Las secciones tituladas Caso de estudio de ingeniería de software al final de los capítulos 1, 3 a 9 y 11 presentan una introducción que lo lleva paso a paso en el diseño orientado a objetos mediante el uso de UML. Primero presentamos un subconjunto conciso y simplificado del UML 2.0, después lo guiamos a través de su primera experiencia de diseño, que está enfocada al diseñador/programador orientado a objetos novato. El caso de estudio no es un ejercicio, sino una experiencia de aprendizaje de extremo a extremo, la cual concluye con un paseo detallado por todo el código de C#. Las secciones tituladas Caso de estudio de ingeniería de software lo ayudan a desarrollar un diseño orientado a objetos para complementar los conceptos de programación orientada a objetos que comenzará a aprender en el capítulo 1 y que implementará en el capítulo 4. En la primera de estas secciones al final del capítulo 1, presentamos los conceptos básicos del DOO y su terminología relacionada. En las secciones opcionales de Caso de estudio de ingeniería de software al final de los capítulos 3 a 6 consideramos cuestiones más sustanciales al resolver un problema avanzado con las técnicas del DOO. Analizamos un documento de requerimientos común, en el cual se especifica la construcción de un sistema, se determinan las clases necesarias para implementar ese sistema, se determinan los atributos que deben tener esas clases, los comportamientos que deben exhibir y se especifica cómo deben interactuar las clases entre sí para cumplir con los requerimientos del sistema. En el apéndice J incluimos una implementación completa en C# del

sistema orientado a objetos que diseñamos en los capítulos anteriores. Empleamos un proceso de diseño orientado a objetos incremental, desarrollado con mucho cuidado, para producir un modelo en UML para nuestro sistema ATM. Con base en este diseño producimos una robusta implementación funcional en C# mediante el uso de las nociones clave de programación, incluyendo clases, objetos, encapsulamiento, visibilidad, composición, herencia y polimorfismo.

Formularios Web, controles Web y ASP.NET 2.0

La plataforma .NET permite a los desarrolladores crear aplicaciones basadas en Web robustas y escalables. La tecnología del lado del servidor .NET de Microsoft, conocida como Páginas Activas de Servidor (ASP) .NET, permite a los programadores crear documentos Web que respondan a las peticiones de los clientes. Para habilitar las páginas Web interactivas, los programas del lado del servidor procesan información que los usuarios introducen en formularios en HTML. ASP.NET provee las capacidades de programación visual mejoradas, similares a las que se utilizan en la creación de formularios de Windows para programas de escritorio. Los programadores pueden crear páginas Web en forma visual, arrastrando y soltando controles en formularios Web. En el capítulo 21, ASP.NET, Formularios Web y Controles Web, introducimos estas poderosas tecnologías.

Servicios Web y ASP.NET 2.0

La estrategia .NET de Microsoft integra a Internet y la Web en el desarrollo y el despliegue de software. La tecnología de servicios Web permite compartir información, el comercio electrónico y demás interacciones mediante el uso de protocolos y tecnologías estándar de Internet, como el Protocolo de Transferencia de Hipertexto (HTTP), el Lenguaje de Marcado Extensible (XML) y el Protocolo Simple de Acceso a Objetos (SOAP). Los servicios Web permiten a los programadores encapsular la funcionalidad de una aplicación de forma que convierta a la Web en una biblioteca de componentes de software reutilizables. En el capítulo 22 presentamos un servicio Web que permite a los usuarios manipular enteros enormes (huge); enteros demasiado grandes como para representarlos con los tipos de datos integrados en C#. En este ejemplo, un usuario introduce dos enteros enormes y oprime botones para invocar servicios Web que suman, restan y comparan ambos enteros. También presentamos un servicio Web de Blackjack y un sistema de reservación de una aerolínea, controlado por una base de datos.

Programación orientada a objetos

La programación orientada a objetos es la técnica más utilizada a nivel mundial para desarrollar software robusto y reutilizable. Este libro ofrece un extenso tratamiento de las características de programación orientada a objetos de C#. En el capítulo 4 se proporciona una introducción a la creación de clases y objetos. Estos conceptos se extienden en el capítulo 9. El capítulo 10 muestra cómo crear nuevas y poderosas clases con rapidez, mediante el uso de la herencia para “absorber” las capacidades de las clases existentes. El capítulo 11 lo familiariza con los conceptos cruciales del polimorfismo, las clases abstractas, las clases concretas y las interfaces, para facilitar manipulaciones poderosas entre objetos que pertenecen a una jerarquía de herencia.

XML

El uso del Lenguaje de Marcado Extensible (XML) es cada vez más amplio en la industria del desarrollo de software y en la comunidad del comercio electrónico (e-business), además de ser predominante en la plataforma .NET. Como XML es una tecnología independiente de la plataforma para describir datos y crear lenguajes de marcado, su portabilidad de datos se integra a la perfección con las aplicaciones y servicios portables basados en C#. En el capítulo 19 se introduce el XML, el mercado del XML y las tecnologías como DTDs y Schema, las cuales se utilizan para validar el contenido de los documentos de XML. También explicamos cómo manipular documentos de XML mediante programación, utilizando el Modelo de Objetos de Documento (DOM™) y cómo transformar documentos de XML en otros tipos de documentos a través de la tecnología de Transformación del Lenguaje de Hojas de estilo Extensible (XSLT).

ADO.NET 2.0

Las bases de datos almacenan enormes cantidades de información a las que deben tener acceso los individuos y las organizaciones para realizar sus actividades comerciales. Como evolución de la tecnología de Objetos de Datos ActiveX de Microsoft (ADO), ADO.NET representa un nuevo enfoque para la creación de aplicaciones que interactúan con bases de datos. ADO.NET utiliza XML y un modelo de objetos mejorado para proporcionar

a los desarrolladores las herramientas que necesitan para acceder a las bases de datos y manipularlas a través de aplicaciones multinivel, de misión específica, extensibles y de gran escala. En el capítulo 20 se introducen las capacidades de ADO.NET y el Lenguaje de Consulta Estructurado (SQL) para manipular bases de datos.

Depurador de Visual Studio 2005

En el apéndice C explicamos cómo utilizar las características clave del depurador, como establecer “puntos de interrupción” (breakpoints) e “inspecciones” (watches), entrar y salir de los métodos y examinar la pila de llamadas a los métodos.

Enfoque de enseñanza

Cómo programar en C#, 2ª edición contiene una vasta colección de ejemplos que se han probado en Windows 2000 y Windows XP. El libro se concentra en los principios de la buena ingeniería de software, haciendo hincapié en la claridad de los programas. Evitamos la terminología arcana y las especificaciones sintácticas para favorecer la enseñanza mediante ejemplos. Somos educadores que enseñamos temas de vanguardia, sobre la materia, en salones de clases alrededor del mundo. El doctor Harvey M. Deitel tiene 20 años de experiencia de enseñanza universitaria y 15 en la enseñanza en la industria. Paul Deitel tiene 12 años de experiencia de enseñanza en la industria, además de ser un experimentado instructor corporativo que ha impartido cursos de todos los niveles a clientes del gobierno, la industria, la milicia y académicos de Deitel & Associates.

Aprenda C# a través del método de código activo (Live-Code)

Cómo programar en C#, 2ª edición está lleno de ejemplos de código activo; cada nuevo concepto se presenta en el contexto de una aplicación completa y funcional en C#, seguida inmediatamente por una o más ejecuciones que muestran las entradas y salidas del programa. Este estilo ejemplifica la manera en que enseñamos y escribimos acerca de la programación. A este método de enseñar y escribir le llamamos método del código activo (o método LIVE-CODE™). *Utilizamos lenguajes de programación para enseñar lenguajes de programación.*

Acceso a World Wide Web

Todos los ejemplos de código fuente para esta segunda edición (y demás publicaciones de los autores) están disponibles (en inglés) en Internet, de manera que puedan descargarse de los siguientes sitios Web:

www.deitel.com/books/csharpforprogrammers2
www.pearsoneducacion.net/deitel

Puede registrarse en forma rápida y sencilla, las descargas son gratuitas. Baje todos los ejemplos y después ejecute cada programa a medida que lea las correspondientes discusiones en el libro. Hacer cambios a los ejemplos e inmediatamente ver los efectos de esos cambios es una excelente manera de mejorar su experiencia de aprendizaje de C#.

Objetivos

Cada capítulo comienza con una declaración de los objetivos.

Frases

Los objetivos de aprendizaje van seguidos de una o más frases. Algunas son graciosas, otras filosóficas y algunas más ofrecen ideas interesantes.

Plan general

El plan general de cada capítulo le permitirá abordar el material de manera ordenada jerárquicamente, para que se pueda anticipar a lo que está por venir y así establecer un ritmo de aprendizaje cómodo y efectivo.

16,785 líneas de código en 213 programas de ejemplo (con la salida de los programas)

Nuestros programas de código activo varían en tamaño, desde unas cuantas líneas de código hasta ejemplos importantes con cientos de líneas de código (por ejemplo, nuestra implementación del sistema de ATM contiene 655 líneas de código). Cada programa va seguido por una ventana que contiene los resultados que se producen al ejecutar dicho programa; de esta manera usted podrá confirmar que los programas se ejecuten como se esperaba.

Nuestros programas demuestran las diversas características de C#. La sintaxis del código va sombreada; las palabras clave de C#, los comentarios y demás texto del programa se enfatizan con variaciones de texto en negrita, cursiva y de color gris. Esto facilita la lectura del código, en especial cuando se leen los programas más grandes.

700 Ilustraciones/Figuras

Incluimos una gran cantidad de gráficas, tablas, dibujos lineales, programas y salidas de programa. Modelamos el flujo de control en las instrucciones de control mediante diagramas de actividad en UML. Los diagramas de clases en UML modelan los campos, constructores y métodos de las clases. Utilizamos tipos adicionales de diagramas en UML a lo largo de nuestro caso de estudio opcional del ATM de DOO/UML.

316 Tips de programación

Hemos incluido tips de programación para enfatizar los aspectos importantes del desarrollo de programas. Resaltamos estos tips como *Buenas prácticas de programación*, *Errores comunes de programación*, *Tips para prevenir errores*, *Observaciones de apariencia visual*, *Tips de rendimiento*, *Tips de portabilidad* y *Observaciones de ingeniería de software*. Estos tips y prácticas representan lo mejor que hemos recabado a lo largo de seis décadas (combinadas) de experiencia en la programación y la enseñanza. Este método es similar al de resaltar axiomas, teoremas y corolarios en los libros de matemáticas; proporciona una base sólida sobre la cual se puede construir buen software.



Buena práctica de programación

Las buenas prácticas de programación *llaman la atención hacia todas las técnicas que ayudan a los desarrolladores a producir programas más legibles, más fáciles de entender y mantener.*



Error común de programación

Los desarrolladores que están aprendiendo un lenguaje a menudo cometen ciertos tipos de errores. Al poner atención en estos Errores comunes de programación se reduce la probabilidad de cometerlos.



Tip de prevención de errores

Cuando diseñamos por primera vez este tipo de tips, pensamos que contendrían sugerencias estrictamente para exponer errores y eliminarlos de los programas. De hecho, muchos de los tips describen aspectos de C# que evitan de antemano la aparición de errores en los programas, con lo cual se simplifican los procesos de prueba y depuración.



Observación de apariencia visual

Ofrecemos las Observaciones de apariencia visual para resaltar las convenciones de la interfaz gráfica de usuario. Estas observaciones le ayudarán a diseñar interfaces gráficas de usuario atractivas y amigables para el usuario en conformidad con las normas de la industria.



Tip de rendimiento

A los desarrolladores les gusta “turbo cargar” sus programas. Incluimos Tips de rendimiento que resaltan las oportunidades para mejorar el rendimiento de los programas: hacer que se ejecuten más rápido o minimizar la cantidad de memoria que ocupan.



Tip de portabilidad

Incluimos Tips de portabilidad para ayudarle a escribir código portable y para explicar cómo C# logra su alto grado de portabilidad.



Observación de ingeniería de software

El paradigma de la programación orientada a objetos requiere de una completa reconsideración de la manera en que construimos los sistemas de software. C# es un lenguaje efectivo para lograr una buena ingeniería de software. Las Observaciones de ingeniería de software resaltan los asuntos de arquitectura y diseño, lo cual afecta la construcción de sistemas de software, en especial los de gran escala.

Sección de conclusión

Cada capítulo termina con una breve sección de conclusión, en la cual se recapitula el contenido del capítulo y las transiciones hacia el siguiente capítulo.

Aproximadamente 5500 entradas en el índice

Hemos incluido un índice extenso, el cual es en especial útil para los desarrolladores que utilizan el libro como una referencia.

“Doble indexado” de ejemplos de código activo en C#

Cómo programar en C#, 2ª edición cuenta con 213 ejemplos de código activo, los cuales hemos indexado dos veces. Para cada programa de código fuente en el libro, indexamos la leyenda de la figura en forma alfabética y también como un elemento de subíndice bajo “Ejemplos”. Esto facilita la búsqueda de ejemplos en los que se utilizan características específicas.

Un paseo por el caso de estudio opcional sobre diseño orientado a objetos con UML

En esta sección daremos un paseo por el caso de estudio opcional del libro, el cual trata sobre el diseño orientado a objetos con UML. En este paseo echaremos un vistazo al contenido de las nueve secciones de Caso de estudio de ingeniería de software (en los capítulos 1, 3 a 9 y 11). Después de completar estas secciones, usted estará muy familiarizado con el diseño y la implementación orientados a objetos para una aplicación importante en C#.

El diseño presentado en el caso de estudio del ATM se desarrolló en Deitel & Associates, Inc.; varios profesionales de la industria lo analizaron con detalle. Nuestro principal objetivo a lo largo del proceso de diseño fue crear un diseño simple que fuera más claro para los desarrolladores que incursionan por primera vez en el DOO y el UML, y que al mismo tiempo se siguieran demostrando los conceptos clave del DOO y las técnicas relacionadas de modelado en UML.

Sección 1.9 (la única sección requerida para el caso de estudio), Caso de estudio de ingeniería de software: introducción a la tecnología de objetos y al UML. En esta sección presentamos el caso de estudio de diseño orientado a objetos con UML. También presentamos los conceptos básicos y la terminología relacionada con la tecnología de los objetos, incluyendo: clases, objetos, encapsulamiento, herencia y polimorfismo. Hablamos sobre la historia de UML. Ésta es la única sección requerida del caso de estudio.

Sección 3.10 (opcional), Caso de estudio de ingeniería de software: análisis del documento de requerimientos del ATM. En esta sección hablamos sobre un *documento de requerimientos*, en el cual se especifican los requerimientos para un sistema que diseñaremos e implementaremos: el software para un cajero automático simple (ATM). Investigamos la estructura y el comportamiento de los sistemas orientados a objetos en general. Hablamos sobre cómo el UML facilitará el proceso de diseño en las subsiguientes secciones del Caso de estudio de ingeniería de software, al proporcionar diversos diagramas para modelar nuestro sistema. Incluimos una lista de URLs y referencias a libros sobre diseño orientado a objetos con UML. Hablamos sobre la interacción entre el sistema ATM especificado por el documento de requerimientos y su usuario. En especial, investigamos los escenarios que pueden ocurrir entre el usuario y el sistema en sí; a éstos se les llama *casos de uso*. Modelamos estas interacciones mediante el uso de los *diagramas de caso de uso* en UML.

Sección 4.11 (opcional), Caso de estudio de ingeniería de software: identificación de las clases en el documento de requerimientos del ATM. En esta sección comenzamos a diseñar el sistema ATM. Identificamos sus clases mediante la extracción de los sustantivos y las frases nominales del documento de requerimientos. Ordenamos estas clases en un diagrama de clases de UML, el cual describe la estructura de las clases de nuestra simulación. Este diagrama de clases también describe las relaciones, conocidas como *asociaciones*, entre las clases.

Sección 5.12 (opcional), Caso de estudio de ingeniería de software: identificación de los atributos de las clases en el sistema ATM. En esta sección nos enfocamos en los atributos de las clases descritas en la sección 3.10. Una clase contiene tanto *atributos* (datos) como *operaciones* (comportamientos). Como veremos en secciones posteriores, los cambios en los atributos de un objeto a menudo afectan el comportamiento de ese objeto. Para determinar los atributos para las clases en nuestro caso de estudio, extraemos los adjetivos que describen los sustantivos y las frases nominales (que definieron nuestras clases) del documento de requerimientos, y después colocamos los atributos en el diagrama de clases que creamos en la sección 3.10.

Sección 6.9 (opcional), Caso de estudio de ingeniería de software: identificación de los estados y las actividades de los objetos en el sistema ATM. En esta sección veremos cómo un objeto, en un tiempo dado, ocupa una condición específica conocida como *estado*. Una *transición de estado* ocurre cuando ese objeto recibe un mensaje para cambiar de estado. UML cuenta con el *diagrama de máquina de estado*, el cual identifica el conjunto de posibles estados que puede ocupar un objeto y modela las transiciones de estado de ese objeto. Un objeto también tiene una *actividad*: el trabajo que desempeña durante su tiempo de vida. UML cuenta con el *diagrama de actividades*: un diagrama de flujo que modela la actividad de un objeto. En esta sección utilizamos ambos tipos de diagramas para comenzar a modelar los aspectos específicos del comportamiento de nuestro sistema ATM, como la forma en que el ATM lleva a cabo una transacción de retiro de dinero y cómo responde cuando el usuario es autenticado.

Sección 7.15 (opcional), Caso de estudio de ingeniería de software: identificación de las operaciones de las clases en el sistema ATM. En esta sección identificamos las operaciones, o servicios, de nuestras clases. Del documento de requerimientos extraemos los verbos y las frases verbales que especifican las operaciones para cada clase. Después modificamos el diagrama de clases de la sección 3.10 para incluir cada operación con su clase asociada. En este punto del caso de estudio ya hemos recopilado toda la información posible del documento de requerimientos. No obstante y a medida que los futuros capítulos introduzcan temas como la herencia, modificaremos nuestras clases y diagramas.

Sección 8.14 (opcional), Caso de estudio de ingeniería de software: colaboración entre los objetos en el sistema ATM. En esta sección hemos creado un “borrador” del modelo para nuestro sistema ATM. Aquí vemos cómo funciona. Investigamos el comportamiento de la simulación al hablar sobre las *colaboraciones*: mensajes que los objetos se envían entre sí para comunicarse. Las operaciones de las clases que hemos descubierto en la sección 6.9 resultan ser las colaboraciones entre los objetos de nuestro sistema. Determinamos las colaboraciones y después las reunimos en un *diagrama de comunicación*: el diagrama de UML para modelar colaboraciones. Este diagrama revela qué objetos colaboran y cuándo lo hacen. Presentamos un diagrama de comunicación de las colaboraciones entre los objetos para realizar una solicitud de saldo en el ATM. Después presentamos el *diagrama de secuencia* de UML para modelar las interacciones en un sistema. Este diagrama enfatiza el ordenamiento cronológico de los mensajes. Un diagrama de secuencia modela la manera en que interactúan los objetos en el sistema para llevar a cabo transacciones de retiro y depósito.

Sección 9.17 (opcional), Caso de estudio de ingeniería de software: cómo empezar a programar las clases del sistema ATM. En esta sección tomamos un descanso, dejando a un lado el diseño del comportamiento de nuestro sistema. Comenzamos el proceso de implementación para enfatizar el material descrito en el capítulo 8. Utilizando el diagrama de clases de UML de la sección 3.10 y los atributos y operaciones descritas en las secciones 4.11 y 6.9, mostramos cómo implementar una clase en C# a partir de un diseño. No implementamos todas las clases, ya que no hemos completado el proceso de diseño. Basándonos en nuestros diagramas de UML, creamos código para la clase Retiro.

Sección 11.9 (opcional), Caso de estudio de ingeniería de software: cómo incorporar la herencia y el polimorfismo al sistema ATM. En esta sección continuamos con nuestra discusión acerca de la programación orientada a objetos. Consideramos la herencia: las clases que comparten características comunes pueden heredar atributos y operaciones de una clase “base”. En esta sección investigamos cómo nuestro sistema ATM puede beneficiarse del uso de la herencia. Documentamos nuestros descubrimientos en un diagrama de clases que modela las relaciones de herencia; UML se refiere a estas relaciones como *generalizaciones*. Después modificamos el diagrama de clases de la sección 3.10 mediante el uso de la herencia para agrupar clases con características similares. En esta sección concluimos el diseño de la porción correspondiente al modelo de nuestra simulación. En el apéndice J implementamos el modelo en forma de código en C#.

Apéndice J: Código del caso de estudio del ATM. La mayor parte del caso de estudio hace hincapié en el diseño del modelo (es decir, los datos y la lógica) del sistema ATM. En este apéndice implementamos ese modelo en C#. Utilizando todos los diagramas de UML que creamos, presentamos las clases de C# necesarias para implementar el modelo. Aplicamos los conceptos del diseño orientado a objetos con el UML y la programación orientada a objetos en C# que usted aprendió a lo largo del libro. Al final de este apéndice usted habrá completado el diseño y la implementación de un sistema real, por lo que deberá sentir la confianza de lidiar con sistemas más grandes, como los que construyen los ingenieros profesionales de software.

Apéndice K, UML 2: tipos de diagramas adicionales. En este apéndice presentamos las generalidades sobre los tipos de diagramas de UML que no se incluyen en el caso de estudio de DOO/UML.

Boletín de correo electrónico gratuito DEITEL® Buzz Online

Nuestro boletín de correo electrónico gratuito *Deitel® Buzz Online* incluye, sin costo alguno, comentarios sobre las tendencias y desarrollos en la industria, vínculos hacia artículos y recursos de nuestros libros publicados y de las próximas publicaciones, itinerarios de liberación de productos, erratas, retos, anécdotas, información sobre nuestros cursos de capacitación corporativa impartidos por instructores, y mucho más. También es una buena forma para mantenerse actualizado acerca de las cuestiones relacionadas con este libro. Para suscribirse, visite el sitio Web:

www.deitel.com/newsletter/subscribe.html

Nota: sitio administrado por los autores del libro.

Agradecimientos

Es un gran placer para nosotros agradecer los esfuerzos de muchas personas cuyos nombres tal vez no aparezcan en la portada, pero cuyo arduo trabajo, cooperación, amistad y comprensión fueron cruciales para la producción del libro. Muchas personas en Deitel & Associates, Inc., dedicaron largas horas a este proyecto.

- Andrew B. Goldberg, graduado de Ciencias Computacionales por la Amherst College, contribuyó en la actualización de los capítulos 19 a 22; también colaboró en el diseño y fue coautor en el nuevo caso de estudio opcional del ATM de DOO/UML. Además fue coautor del apéndice K.
- Su Zhang posee títulos de B.Sc. y una Maestría en Ciencias Computacionales por la Universidad McGill. Su Zhang contribuyó en los capítulos 25 y 26, así como también en el apéndice J.
- Cheryl Yeager, graduado de la Universidad de Boston como licenciado en Ciencias Computacionales, ayudó a actualizar los capítulos 13-14.
- Barbara Deitel, Directora en Jefe de Finanzas en Deitel & Associates, Inc., aplicó las correcciones al manuscrito del libro.
- Abbey Deitel, Presidente de Deitel & Associates, Inc., licenciado en Administración Industrial por la Universidad Carnegie Mellon, fue coautor del capítulo 1.
- Christi Kelsey, graduada de la Universidad Purdue con título en comercio y especialización secundaria en sistemas de información, fue coautora del capítulo 2, el prefacio y el apéndice C. Ella editó el índice y paginó todo el manuscrito. También trabajó en estrecha colaboración con el equipo de producción en Prentice Hall, coordinando prácticamente todos los aspectos relacionados con la producción de este libro.

También queremos agradecer a los tres participantes de nuestro Programa de pasantía para estudiantes sobresalientes (Honors Internship) y los programas de educación cooperativa que contribuyeron a esta publicación: Nick Santos, con especialidad en Ciencias Computacionales en el Dartmouth Collage; Jeffrey Peng, estudiante de Ciencias Computacionales en la Universidad Cornell y William Chen, estudiante de Ciencias Computacionales en la Universidad Cornell.

Nos sentimos afortunados de haber trabajado en este proyecto con el talentoso y dedicado equipo de editores profesionales en Pearson Education/PTG. En especial, apreciamos los extraordinarios esfuerzos de Mark Taub, Editor asociado de Prentice Hall/PTR, y Marcia Horton, Directora editorial de la División de Ciencias computacionales e ingeniería de Prentice Hall. Noreen Regina y Jennifer Cappello hicieron un extraordinario trabajo al reclutar los equipos de revisión para el libro y administrar el proceso de revisión. Sandra Schroeder hizo un maravilloso trabajo al rediseñar por completo la portada del libro. Vince O'Brien, Bob Engelhardt, Donna Crilly y Marta Samsel hicieron un estupendo trabajo al administrar la producción del libro.

Nos gustaría agradecer de manera especial a Dan Fernandez, Gerente de producto de C#, y a Janie Schwark, Gerente Comercial en Jefe de la División de Mercadotecnia para el Desarrollador, ambos de Microsoft, por su esfuerzo especial al trabajar con nosotros en este proyecto. También agradecemos a los muchos otros miembros del equipo de Microsoft que se tomaron el tiempo para responder a nuestras preguntas a lo largo de este proceso:

Anders Hejlsburg, Ayudante técnico (C#)
 Brad Abrams, Gerente de Programa en Jefe (.NET Framework)
 Jim Miller, Arquitecto de Software (.NET Framework)
 Joe Duffy, Gerente de Programa (.NET Framework)
 Joe Stegman, Gerente de Programa en Jefe (Windows Forms)
 Kit George, Gerente de Programa (.NET Framework)
 Luca Bolognese, Gerente de Programa en Jefe (C#)

Luke Hoban, Gerente de Programa (C#)
 Mads Torgersen, Gerente de Programa (C#)
 Peter Hallam, Ingeniero de Diseño de Software (C#)
 Scout Nonnenberg, Gerente de Programa (C#)
 Shamez Rajan, Gerente de Programa (Visual Basic)

Deseamos agradecer también los esfuerzos de nuestros revisores. Al adherirse a un apretado itinerario de trabajo, escudriñaron el texto y los programas, proporcionando innumerables sugerencias para mejorar la precisión y finalización de la presentación.

Revisores de Microsoft

George Bullock, Gerente de Programa en Microsoft, Equipo Comunitario Microsoft.com
 Darmesh Chauhan, Microsoft
 Shon Katzenberger, Microsoft
 Matteo Taveggia, Microsoft
 Matt Tavis, Microsoft

Revisores de la Industria

Alex Bondarev, Investor's Bank and Trust
 Peter Bromberg, Arquitecto en Jefe de Merrill Lynch y MVP de C#
 Vijay Cinnakonda, TrueCommerce, Inc.
 Jay Cook, Alcon Laboratorios
 Jeff Cowan, Magenic, Inc.
 Ken Cox, Consultor independiente, escritor y desarrollador, y MVP de ASP.NET
 Stochio Goutsev, Consultor independiente, escritor y desarrollador, y MVP de C#
 James Huddleston, Consultor independiente
 Rex Jaeschke, Consultor independiente
 Saurabh Nandú, AksTech Solutions Pvt. Ltd.
 Simon North, Quintiq BV
 Mike O'Brien, Departamento de Desarrollo de Empleos del Estado de California
 José Antonio González Seco, Parlamento de Andalucía
 Devan Shepard, Xmalpha Technologies
 Pavel Tsekov, Caesar BSC
 John Varghese, UBS
 Stacey Yassenka, Desarrollador de software en Hyland Software y MVP de C#

Revisores académicos

Rekha Bhowmik, Universidad Luterana de California
 Ayad Boudiab, Georgia Perimeter Collage
 Harlan Brewer, Universidad de Cincinnati
 Sam Gill, Universidad Estatal de San Francisco
 Gavin Osborne, Saskatchewan Institute of Applied Science and Technology
 Catherine Wyman, DeVry-Phoenix

Bueno, eso es todo. C# es un poderoso lenguaje de programación que le ayudará a escribir programas con rapidez y efectividad. C# se adapta perfectamente al ámbito del desarrollo de sistemas a nivel empresarial, para ayudar a las organizaciones a construir sus sistemas críticos de información para los negocios y de misión crítica. Le agradeceremos sinceramente que, a medida que lea el libro, nos haga saber sus comentarios, críticas, correcciones y sugerencias para mejorar el texto. Por favor dirija toda su correspondencia a:

deitel@deitel.com

Le responderemos oportunamente y publicaremos las correcciones y aclaraciones en nuestro sitio Web:

www.deitel.com

¡Esperamos que disfrute leyendo *Cómo programar en C#, 2ª edición* tanto como nosotros disfrutamos el escribirlo!

Dr. Harvey M. Deitel
Paul J. Deitel

Acerca de los autores

El Dr. Harvey M. Deitel, Presidente y Jefe de Estrategias de Deitel & Associates, Inc., tiene 44 años de experiencia en el campo de la computación, esto incluye un amplio trabajo académico y en la industria. El Dr. Deitel tiene una licenciatura y una maestría por el Massachusetts Institute of Technology y un doctorado de la Boston University. Trabajó en los proyectos pioneros de sistemas operativos de memoria virtual en IBM y el MIT, con los cuales se desarrollaron técnicas que ahora se implementan ampliamente en sistemas como UNIX, Linux y Windows XP. Tiene 20 años de experiencia como profesor universitario, incluyendo un puesto vitalicio y el haber sido presidente del Departamento de Ciencias Computacionales en el Boston College antes de fundar, con su hijo Paul J. Deitel, Deitel & Associates, Inc. Él y Paul son coautores de varias docenas de libros y paquetes multimedia, y piensan escribir muchos más. Los textos de los Deitel se han ganado el reconocimiento internacional y han sido traducidos al japonés, alemán, ruso, español, chino tradicional, chino simplificado, coreano, francés, polaco, italiano, portugués, griego, urdú y turco. El Dr. Deitel ha impartido cientos de seminarios profesionales para grandes empresas, instituciones académicas, organizaciones gubernamentales y diversos sectores del ejército.

Paul J. Deitel, CEO y Jefe Técnico de Deitel & Associates, Inc., es egresado del Sloan School of Management del Massachusetts Institute of Technology, en donde estudió Tecnología de la Información. A través de Deitel & Associates, Inc., ha impartido cursos de Java, C, C++, Internet y World Wide Web a clientes de la industria, incluyendo IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, la NASA en el Centro Espacial Kennedy, el Nacional Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems y muchos más. Paul es uno de los instructores de Java más experimentados, ya que ha impartido cerca de 100 cursos de Java profesionales. También ha ofrecido conferencias sobre C++ y Java para la Boston Chapter de la Association for Computing Machinery. Él y su padre, el Dr. Harvey M. Deitel, son autores de los libros de texto de lenguajes de programación con más ventas en el mundo.

Acerca de Deitel & Associates, Inc.

Deitel & Associates, Inc. es una empresa reconocida a nivel mundial, dedicada al entrenamiento corporativo y la creación de contenido, con especialización en lenguajes de programación de computadoras, tecnología de software para Internet/World Wide Web, educación en tecnología de objetos y desarrollo de comercios en Internet. La empresa proporciona cursos impartidos por instructores sobre los principales lenguajes y plataformas de programación, como Java, Java Avanzado, C, C++, los lenguajes de programación .NET, XML, Perl, Python; tecnología de objetos y programación en Internet y World Wide Web. Los fundadores de Deitel & Associates, Inc. son el Dr. Harvey M. Deitel y Paul J. Deitel. Sus clientes incluyen muchas de las empresas de cómputo más grandes del mundo, agencias gubernamentales, sectores del ejército y empresas comerciales. A lo largo de su sociedad editorial de 29 años con Prentice Hall, Deitel & Associates, Inc. ha publicado libros de texto de vanguardia sobre programación, libros profesionales, multimedia interactiva en CD como los *Cyber Classrooms*, cursos de capacitación basados en Web y contenido electrónico para sistemas de administración de cursos populares como WebCT, BlackBoard y CourseCompass de Pearson. Deitel & Associates, Inc. y los autores pueden ser contactados mediante correo electrónico en:

deitel@deitel.com

Para conocer más acerca de Deitel & Associates, Inc., sus publicaciones y su currículo de Capacitación corporativa mundial de la serie *DIVE INTO™*, consulte las últimas páginas de este libro o visite el sitio Web:

www.deitel.com

y suscríbase al boletín de correo electrónico gratuito *Deitel® Buzz Online* en:

www.deitel.com/newsletter/subscribe.html

Las personas que deseen comprar libros de Deitel, Cyber Classrooms, Cursos de capacitación completos y cursos de capacitación basados en Web pueden hacerlo a través de la siguiente página Web:

www.deitel.com/books/index.html

Las empresas e instituciones académicas que deseen hacer pedidos en grandes cantidades pueden hacerlo directamente con Pearson Educación.

1

Introducción a las computadoras, Internet y Visual C#

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- La historia del lenguaje de programación Visual C#.
- Algunos fundamentos sobre la tecnología de los objetos.
- La historia del UML: el lenguaje de modelado de sistemas orientado a objetos estándar en la industria.
- La historia de Internet y World Wide Web.
- La motivación detrás de la iniciativa .NET de Microsoft y sus generalidades, en la cual se involucra a Internet en el desarrollo y uso de sistemas de software.
- Cómo probar una aplicación de Visual C# 2005 que le permitirá dibujar en la pantalla.

El principal mérito del lenguaje es la claridad.

—Galen

Los pensamientos elevados deben tener un lenguaje elevado.

—Aristófanes

Nuestra vida se malgasta con los detalles... Simplifica, simplifica.

—Henry David Thoreau

Mi objetivo sublime lo lograré con el tiempo.

—W. S. Gilbert

El hombre es aún la computadora más extraordinaria de todas.

—John F. Kennedy

- 1.1 Introducción
- 1.2 Sistema operativo Windows® de Microsoft
- 1.3 C#
- 1.4 Internet y World Wide Web
- 1.5 Lenguaje de Marcado Extensible (XML)
- 1.6 Microsoft .NET
- 1.7 .NET Framework y Common Language Runtime
- 1.8 Prueba de una aplicación en C#
- 1.9 (Única sección obligatoria del caso de estudio) Caso de estudio de ingeniería de software: Introducción a la tecnología de objetos y el UML
- 1.10 Conclusión
- 1.11 Recursos Web

1.1 Introducción

¡Bienvenido a Visual C# 2005! Nos hemos esforzado mucho para ofrecerle información precisa y completa relacionada con este poderoso lenguaje de programación, que de ahora en adelante llamaremos simplemente C#. Este lenguaje es apropiado para crear sistemas de información complejos. Esperamos que su trabajo con este libro sea una experiencia informativa, retadora y divertida.

El principal objetivo de este libro es lograr la legibilidad de los programas a través de las técnicas comprobadas de la programación orientada a objetos (POO) y la programación controlada por eventos. Tal vez lo más importante de este libro es que presenta cientos de programas completos y funcionales, además de la descripción de sus entradas y salidas. A esto le llamamos *método de código activo*. Puede descargar todos los ejemplos del libro del sitio web www.deitel.com/books/csharpforprogrammers2/index.html y de www.prenhall.com/deitel.

Esperamos que disfrute la experiencia de aprendizaje con este libro. Está a punto de emprender un camino lleno de retos y recompensas. Si tiene dudas a medida que avance, no dude en enviarnos un correo electrónico a

deitel@deitel.com

Para mantenerse actualizado con los avances sobre C# en Deitel & Associates, y recibir actualizaciones en relación con este libro, suscríbase a nuestro boletín electrónico gratuito *Deitel® Buzz Online* en:

www.deitel.com/newsletter/subscribe.html

1.2 Sistema operativo Windows® de Microsoft

Durante las décadas de 1980 y 1990, Microsoft Corporation se convirtió en la compañía dominante de software. En 1981 sacó al mercado la primera versión del sistema operativo DOS para la computadora personal IBM. A mediados de la década de 1980, desarrolló el *sistema operativo Windows*, una interfaz gráfica de usuario creada para ejecutarse sobre el DOS. En 1990, Microsoft liberó el sistema Windows 3.0, esta nueva versión contaba con una interfaz amigable para el usuario y una funcionalidad robusta. El sistema operativo Windows se hizo increíblemente popular después de la liberación, en 1992, de Windows 3.1, cuyos sucesores (Windows 95 y Windows 98) prácticamente atajaron el mercado de los sistemas operativos de escritorio a finales de la década de 1990. Estos sistemas operativos, que tomaron muchos conceptos (como los iconos, los menús y las ventanas) popularizados por los primeros sistemas operativos Apple Macintosh, permitían a los usuarios navegar por varias aplicaciones al mismo tiempo. En 1993, Microsoft entró al mercado de los sistemas operativos corporativos con la presentación de Windows NT®. Windows XP está basado en el sistema operativo Windows NT y se presentó en el 2001. Este sistema operativo combina las líneas de sistemas operativos corporativos y para el consumidor de Microsoft. Hasta ahora, Windows es el sistema operativo más utilizado en el mundo.

El mayor competidor del sistema operativo Windows es Linux. Este nombre se deriva de Linus (por Linus Torvalds, quien desarrolló Linux) y UNIX (el sistema operativo en el cual está basado Linux); este último se

desarrolló en los Laboratorios Bell y está escrito en el lenguaje de programación C. Linux es un sistema operativo gratuito, de *código fuente abierto*. A diferencia de Windows, que es propietario (Microsoft posee los derechos de propiedad y el control); el código fuente de Linux está disponible sin costo para los usuarios, y pueden modificarlo para ajustarlo a sus necesidades.

1.3 C#

El avance de las herramientas de programación y los dispositivos electrónicos para el consumidor (por ejemplo, los teléfonos celulares y los PDAs) ha creado problemas y nuevos requerimientos. La integración de componentes de software de diversos lenguajes fue difícil y los problemas de instalación eran comunes, ya que las nuevas versiones de los componentes compartidos eran incompatibles con el software anterior. Los desarrolladores también descubrieron que necesitaban aplicaciones basadas en Web, que pudieran accesarse y utilizarse a través de Internet. Como resultado de la popularidad de los dispositivos electrónicos móviles, los desarrolladores de software se dieron cuenta que sus clientes ya no estaban restringidos sólo a las computadoras de escritorio. Reconocieron que hacía falta software accesible para todos y que estuviera disponible a través de casi cualquier tipo de dispositivo. Para satisfacer estas necesidades, en el año 2000, Microsoft anunció el lenguaje de programación **C#**. Este lenguaje, desarrollado en Microsoft por un equipo dirigido por Anders Heljsberg y Scott Wiltamuth, se diseñó en específico para la plataforma .NET (de la cual hablaremos en la sección 1.6) como un lenguaje que permitiera a los programadores migrar con facilidad hacia .NET. Tiene sus raíces en C, C++ y Java; adapta las mejores características de cada uno de estos lenguajes y agrega nuevas características propias. C# está orientado a objetos y contiene una poderosa *biblioteca de clases*, que consta de componentes preconstruidos que permiten a los programadores desarrollar aplicaciones con rapidez; C# y Visual Basic comparten la Biblioteca de Clases Framework (FCL), de la cual hablaremos en la sección 1.6. C# es apropiado para las tareas de desarrollo de aplicaciones demandantes, en especial para crear las aplicaciones populares basadas en la Web actual.

La *plataforma .NET* es la infraestructura sobre la cual pueden distribuirse aplicaciones basadas en Web a una gran variedad de dispositivos (incluso los teléfonos celulares) y computadoras de escritorio. La plataforma ofrece un nuevo modelo de desarrollo de software, que permite crear aplicaciones en distintos lenguajes de programación, de manera que se comuniquen entre sí.

C# es un lenguaje de programación visual, controlado por eventos, en el cual se crean programas mediante el uso de un *Entorno de Desarrollo Integrado (IDE)*. Con un IDE, un programador puede crear, ejecutar, probar y depurar programas en C# de manera conveniente, con lo cual se reduce el tiempo requerido para producir un programa funcional en una fracción del tiempo que se llevaría sin utilizar el IDE. La plataforma .NET permite la interoperabilidad de los lenguajes: los componentes de software de distintos lenguajes pueden interactuar como nunca antes se había hecho. Los desarrolladores pueden empaquetar incluso hasta el software antiguo para que trabaje con nuevos programas en C#. Además, las aplicaciones en C# pueden interactuar a través de Internet mediante el uso de estándares industriales como XML, del cual hablaremos en el capítulo 19, y el Protocolo Simple de Acceso a Objetos (SOAP) basado en XML, del cual hablaremos en el capítulo 22, Servicios Web.

El lenguaje de programación C# original se estandarizó a través de Ecma International (www.ecma-international.org) en diciembre del 2002, como *Estándar ECMA-334: Especificación del lenguaje C#* (ubicado en www.ecma-international.org/publications/standards/Ecma-334.htm). Desde entonces, Microsoft propuso varias extensiones del lenguaje que se han adoptado como parte del estándar Ecma C# revisado. Microsoft hace referencia al lenguaje C# completo (incluyendo las extensiones adoptadas) como **C# 2.0**.

[Nota: A lo largo de este libro proporcionamos referencias a secciones determinadas de la *Especificación del lenguaje C#*. Utilizamos los números de sección señalados en la versión de Microsoft de esa especificación, la cual está compuesta de dos documentos: la *Especificación del lenguaje C# 1.2* y la *Especificación del lenguaje C# 2.0* (una extensión del documento 1.2 que contiene las mejoras del lenguaje C# 2.0). Ambos documentos se encuentran en msdn.microsoft.com/vcsharp/programming/language/.]

1.4 Internet y World Wide Web

Internet (una red global de computadoras) se inició hace casi cuatro décadas, con fondos suministrados por el Departamento de Defensa de los Estados Unidos. Aunque originalmente se diseñó para conectarse con los principales sistemas computacionales de aproximadamente una docena de universidades y organizaciones de investigación,

su principal beneficio ha sido, desde ese entonces, la capacidad de comunicarse en forma fácil y rápida a través de lo que llegó a conocerse como *correo electrónico (e-mail)*. Esto es cierto incluso en la actualidad, en donde el correo electrónico, la mensajería instantánea y la transferencia de archivos facilitan las comunicaciones entre cientos de millones de personas en todo el mundo. Internet se ha convertido en uno de los mecanismos de comunicación más importantes a nivel mundial, y continúa creciendo con rapidez.

La *World Wide Web* permite a los usuarios de computadora localizar y ver, a través de Internet, documentos basados en multimedia, sobre cualquier tema. A pesar de que Internet se desarrolló hace varias décadas, la introducción de la Web es un evento relativamente reciente. En 1989, Tim Berners-Lee de CERN (la Organización Europea de Investigación Nuclear) comenzó a desarrollar una tecnología para compartir información a través de documentos de texto con hipervínculos. Berners-Lee llamó a su invención el *Lenguaje de Marcado de Hipertexto (HTML)*. También escribió protocolos de comunicación para formar las bases de su nuevo sistema de información, al cual denominó World Wide Web.

En el pasado, la mayoría de las aplicaciones se ejecutaba en computadoras que no estaban conectadas entre sí. Las aplicaciones actuales pueden escribirse de manera que se comuniquen con computadoras de todo el mundo. Internet mezcla las tecnologías de la computación y la comunicación, facilitando nuestro trabajo. Hace que la información esté accesible de manera instantánea y conveniente en todo el mundo, y permite a las personas y pequeñas empresas tener una presencia que no tendrían de otra manera. Está cambiando la forma en que se hacen los negocios. La gente puede buscar los mejores precios de casi cualquier producto o servicio, mientras que los miembros de las comunidades de interés especial pueden permanecer en contacto unos con otros, y los investigadores pueden estar actualizados de inmediato sobre los descubrimientos más recientes. Sin duda, Internet y World Wide Web se encuentran entre las creaciones más importantes de la humanidad. En los capítulos del 19 al 22 aprenderá a crear aplicaciones basadas en Internet y Web.

En 1994, Tim Berners-Lee fundó una organización llamada *Consorcio World Wide Web* (o *W3C*), dedicada a desarrollar tecnologías no propietarias e interoperables para World Wide Web. Una de las principales metas de la W3C es hacer que la Web sea accesible de manera universal, sin importar las discapacidades, el lenguaje o la cultura.

La W3C (www.w3.org) es también una organización de estandarización. Las tecnologías Web estandarizadas por la W3C se conocen como Recomendaciones; y las más actuales incluyen el *Lenguaje de Marcado Extensible (XML)*. En la sección 1.5 presentaremos el XML y en el capítulo 19, Lenguaje de Marcado Extensible (XML), lo veremos con detalle. Este lenguaje es la tecnología clave detrás de la próxima versión de World Wide Web, que algunas veces se le denomina “Web semántica”. Es también una de las tecnologías clave detrás de los servicios Web, de los cuales hablaremos en el capítulo 22.

1.5 Lenguaje de Marcado Extensible (XML)

A medida que fue creciendo la popularidad de la Web, sus limitaciones del HTML se hicieron aparentes. La falta de *extensibilidad* (la habilidad de cambiar o agregar características) frustró a los desarrolladores, y su definición ambigua permitía la proliferación de HTML con errores. Era inminente la necesidad de un lenguaje estandarizado, estricto en estructura y completamente extensible. Como resultado, la W3C desarrolló XML.

La *independencia de datos*, la separación del contenido de su presentación, es la característica esencial del XML. Debido a que los documentos de XML describen los datos en forma independiente de la máquina, es concebible que cualquier aplicación pueda procesarlos. Los desarrolladores de software están integrando XML en sus aplicaciones para mejorar la funcionalidad e interoperabilidad de la Web.

XML no se limita a las aplicaciones Web. Por ejemplo, cada vez se está utilizando más en bases de datos; la estructura de un documento XML le permite integrarse fácilmente con las aplicaciones de bases de datos. A medida que las aplicaciones incluyan más funcionalidad para la Web, es probable que XML se convierta en la tecnología universal para la representación de datos. Todas las aplicaciones que empleen XML podrán comunicarse entre sí, siempre y cuando puedan comprender sus esquemas de marcado de XML respectivos, a los cuales se les conoce como *vocabularios*.

El *Protocolo Simple de Acceso a Objetos (SOAP)* es una tecnología para la transmisión de objetos (marcados como XML) a través de Internet. Las tecnologías .NET de Microsoft (que veremos en las siguientes dos secciones) utilizan XML y SOAP para marcar y transferir datos a través de Internet. XML y SOAP se encuentran en el núcleo de .NET; permiten la interoperación de los componentes de software (es decir, que se comuniquen fácilmente unos con otros). Como las bases de SOAP son en XML y *HTTP (Protocolo de Transferencia de*

Hipertexto; el protocolo de comunicación clave de la Web), la mayoría de los tipos de sistemas computacionales lo soporta. En el capítulo 19, Lenguaje de marcado extensible (XML), hablaremos sobre el XML y en el capítulo 22, Servicios Web, hablaremos sobre SOAP.

1.6 Microsoft .NET

En el año 2000, Microsoft anunció su *iniciativa .NET* (www.microsoft.com/net en inglés, www.microsoft.com/latam/net en español), una nueva visión para incluir Internet y Web en el desarrollo y uso de software. Un aspecto clave de .NET es su independencia de un lenguaje o plataforma específicos. En vez de estar forzados a utilizar un solo lenguaje de programación, los desarrolladores pueden crear una aplicación .NET en cualquier lenguaje compatible con .NET. Los programadores pueden contribuir al mismo proyecto de software, escribiendo código en los lenguajes .NET (como Microsoft Visual C#, Visual C++, Visual Basic y muchos otros) en los que sean más competentes. Parte de la iniciativa incluye la tecnología *ASP.NET* de Microsoft, la cual permite a los programadores crear aplicaciones para Web. En el capítulo 21, ASP.NET 2.0, Formularios y Controles Web, hablaremos sobre ASP.NET. En el capítulo 22 utilizaremos la tecnología ASP.NET para crear aplicaciones que utilicen servicios Web.

La arquitectura .NET puede existir en varias plataformas, no sólo en los sistemas basados en Microsoft Windows, con lo cual se extiende la portabilidad de los programas .NET. Un ejemplo es Mono (www.mono-project.com/Main_Page), un proyecto de software libre de Novell. Otro ejemplo es .NET Portable de DotGNU (www.dotgnu.org).

Un componente clave de la arquitectura .NET son los *servicios Web*, que son componentes de software reutilizables que pueden usarse a través de Internet. Los clientes y otras aplicaciones pueden usar los servicios Web como bloques de construcción reutilizables. Un ejemplo de un servicio Web es el sistema de reservación de Dollar Rent a Car (<http://www.microsoft.com/casestudies/casestudy.aspx?casestudyid=50318>). Un socio de una aerolínea deseaba permitir que los clientes hicieran reservaciones de autos de renta desde el sitio Web de la aerolínea. Para ello, la aerolínea necesitaba acceder al sistema de reservaciones de Dollar. En respuesta, Dollar creó un servicio Web que permitía a la aerolínea el acceso a su base de datos para hacer reservaciones. Los servicios Web permitían que dos computadoras de las dos compañías se comunicaran a través de Web, aun y cuando la aerolínea utiliza sistemas UNIX y Dollar utiliza Microsoft Windows. Dollar pudo haber creado una solución específica para esa aerolínea en particular, pero nunca hubiera podido reutilizar dicho sistema personalizado. El servicio Web de Dollar permite que muchas aerolíneas, hoteles y agencias de viaje utilicen su sistema de reservaciones sin necesidad de crear un programa personalizado para cada relación.

La estrategia .NET extiende el concepto de reutilización de software hasta Internet, con lo cual permite que los programadores y las compañías se concentren en sus especialidades sin tener que implementar cada componente de cada aplicación. En vez de ello, las compañías pueden comprar los servicios Web y dedicar sus recursos a desarrollar sus propios productos. Por ejemplo, una aplicación individual que utilice los servicios Web de varias compañías podría administrar los pagos de facturas, devoluciones de impuestos, préstamos e inversiones. Un comerciante de aerolínea podría comprar servicios Web para los pagos de tarjeta de crédito en línea, la autenticación de los usuarios, la seguridad de la red y bases de datos de inventario para crear un sitio Web de comercio electrónico.

1.7 .NET Framework y Common Language Runtime

El *.NET Framework* de Microsoft es el corazón de la estrategia .NET. Este marco de trabajo administra y ejecuta aplicaciones y servicios Web, contiene una biblioteca de clases (llamada Biblioteca de clases del .NET Framework, o FCL), impone la seguridad y proporciona muchas otras herramientas de programación. Los detalles del .NET Framework se encuentran en la *Infraestructura de Lenguaje Común (CLI)*, la cual contiene información acerca del almacenamiento de los tipos de datos (es decir, datos que tienen características predefinidas como fecha, porcentaje o una cantidad monetaria), los objetos y demás. Ecma International (conocida en un principio como la Asociación Europea de Fabricantes de Computadoras) ha estandarizado la CLI, con lo cual se facilita la creación del .NET Framework para otras plataformas. Esto es como publicar los planos del marco de trabajo; cualquiera puede construirlo con sólo seguir las especificaciones.

El entorno en tiempo de ejecución *Common Language Runtime (CLR)* es otra parte central del .NET Framework: ejecuta los programas de .NET. Los programas se compilan en instrucciones específicas para la máquina

en dos pasos. Primero, el programa se compila en *Lenguaje Intermedio de Microsoft (MSIL)*, el cual define las instrucciones para el CLR. El CLR puede unir el código convertido en MSIL proveniente de otros lenguajes y fuentes. El MSIL para los componentes de una aplicación se coloca en el archivo ejecutable de la aplicación (lo que se conoce como *ensamblaje*). Cuando la aplicación se ejecuta, otro compilador (conocido como *compilador justo a tiempo* o *compilador JIT*) en el CLR traduce el MSIL del archivo ejecutable en código de lenguaje máquina (para una plataforma específica) y después el código de lenguaje máquina se ejecuta en esa plataforma. [Nota: MSIL es el nombre de Microsoft para lo que la especificación del lenguaje C# denomina *Lenguaje Intermedio Común (CIL)*.]

Si existe el .NET Framework (y está instalado) para una plataforma, esa plataforma puede ejecutar cualquier programa .NET. La habilidad de un programa para ejecutarse (sin modificaciones) en varias plataformas se conoce como *independencia de la plataforma*. De esta forma, el código escrito puede usarse en otro tipo de computadora sin modificación, con lo cual se ahorra tiempo y dinero. Además, el software puede abarcar una audiencia mayor; antes las compañías tenían que decidir si valía la pena el costo de convertir sus programas para usarlos en distintas plataformas (lo que algunas veces se conoce como *portar*). Con .NET, el proceso de portar los programas ya no representa un problema (una vez que .NET esté disponible en esas plataformas).

El .NET Framework también proporciona un alto nivel de *interoperabilidad de lenguajes*. Los programas escritos en distintos lenguajes se compilan en MSIL; las diferentes partes pueden combinarse para crear un solo programa unificado. MSIL permite que el .NET Framework sea *independiente del lenguaje*, ya que los programas .NET no están atados a un lenguaje de programación en especial. Cualquier lenguaje que pueda compilarse en MSIL se denomina *lenguaje compatible con .NET*. La figura 1.1 lista muchos de los lenguajes de programación disponibles para la plataforma .NET (msdn.microsoft.com/netframework/technologyinfo/overview/default.aspx).

La interoperabilidad de lenguajes ofrece muchos beneficios a las compañías de software. Por ejemplo, los desarrolladores en C#, Visual Basic y Visual C++ pueden trabajar lado a lado en el mismo proyecto, sin tener que aprender otro lenguaje de programación; todo su código se compila en MSIL y se enlaza para formar un solo programa.

Cualquier lenguaje .NET puede utilizar la Biblioteca de Clases del .NET Framework (FCL). Esta biblioteca contiene una variedad de componentes reutilizables, con lo cual los programadores se ahorran el problema de crear nuevos componentes. En este libro le explicaremos cómo desarrollar software .NET con C#.

Lenguajes de programación .NET	
APL	Mondrian
C#	Oberon
COBOL	Oz
Component Pascal	Pascal
Curriculum	Perl
Eiffel	Python
Forth	RPG
Fortran	Scheme
Haskell	Smalltalk
Java	Standard ML
JScript	Visual Basic
Mercury	Visual C++

Figura 1.1 | Los lenguajes de programación .NET.

I.8 Prueba de una aplicación en C#

En esta sección usted “probará” una aplicación, en C#, que le permite dibujar en la pantalla mediante el uso del ratón. Ejecutará una aplicación funcional e interactuará con ella. En el capítulo 13, Conceptos de interfaz gráfica de usuario: parte 1 creará una aplicación similar.

La aplicación **Drawing** le permite dibujar con distintos tamaños de brocha y diversos colores. Los elementos y la funcionalidad que se pueden ver en esta aplicación son muestras de lo que usted aprenderá a programar en este texto. Utilizamos tipos de letras para diferenciar las características del IDE (como los nombres y los elementos de los menús) y otros elementos que aparecen en el IDE. Nuestra convención es enfatizar las características del IDE (como el menú **Archivo**) en un tipo de letra **sans-serif Helvética** seminegrita y los demás elementos, como los nombres de archivo (por ejemplo, Form1.cs) en un tipo de letra **sans-serif Lucida**. Los siguientes pasos le mostrarán cómo probar la aplicación.

1. *Compruebe su instalación.* Confirme que ha instalado Visual C# 2005 Express o Visual Studio 2005, como vimos en el *Prefacio*.
2. *Localice el directorio de aplicaciones.* Abra el Explorador de Windows y navegue hasta el directorio C:\examples\Ch01\Drawing.
3. *Ejecute la aplicación **Drawing**.* Ahora que se encuentra en el directorio correcto, haga doble clic sobre el nombre de archivo Drawing.exe para ejecutar la aplicación.

En la figura 1.2 se etiquetan varios elementos gráficos (llamados *controles*). En esta aplicación se incluyen dos controles **GroupBox** (en este caso, **Color** y **Size**), siete controles **RadioButton** y un control **Panel** (más adelante hablaremos sobre estos controles de forma detallada). La aplicación **Drawing** le permite dibujar con una brocha color rojo, azul, verde o negro de un tamaño pequeño, mediano o grande. En esta prueba explorará todas estas opciones.

Puede utilizar controles existentes (que son objetos) para crear poderosas aplicaciones que se ejecuten en C# con mucha más rapidez que si tuviera que escribir todo el código por su cuenta. En este libro aprenderá a utilizar muchos de los controles preexistentes, así como también a escribir su propio código de programa para personalizar sus aplicaciones.

Las propiedades de la brocha, que se seleccionan mediante los controles **RadioButton** (los pequeños círculos en donde para seleccionar una opción se hace clic sobre ella con el ratón) etiquetados como **Black** y **Small** son opciones predeterminadas que constituyen la configuración inicial que se ve cuando se ejecuta la aplicación por primera vez. Los programadores incluyen opciones predeterminadas para ofrecer selecciones razonables que utilizan la aplicación en caso de que el usuario no desee modificar la configuración. Ahora usted elegirá sus propias opciones.

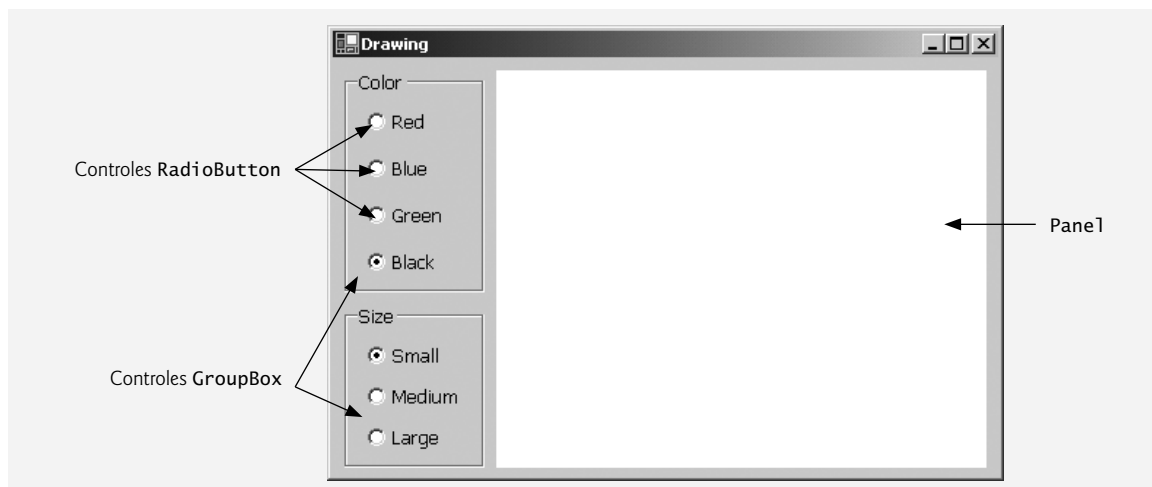


Figura 1.2 | Aplicación **Drawing** en Visual C#.

4. *Cambie el color de la brocha.* Haga clic en el control `RadioButton` con la etiqueta **Red** para cambiar el color de la brocha. Mantenga oprimido el botón del ratón y el puntero del mismo posicionado en cualquier parte del control `Panel`, después arrastre el ratón para dibujar con la brocha. Dibuje los pétalos de una flor, como se muestra en la figura 1.3. Después haga clic en el control `RadioButton` con la etiqueta **Green** para cambiar el color de la brocha de nuevo.
5. *Cambie el tamaño de la brocha.* Haga clic en el control `RadioButton` con la etiqueta **Large** para cambiar el tamaño de la brocha. Dibuje pasto y el tallo de una flor, como se muestra en la figura 1.4.
6. *Termine el dibujo.* Haga clic en el control `RadioButton` con la etiqueta **Blue**. Después haga clic en el control `RadioButton` con la etiqueta **Medium**. Dibuje gotas de lluvia, como se muestra en la figura 1.5 para completar el dibujo.
7. *Cierre la aplicación.* Haga clic en el **cuadro cerrar**, , para cerrar la aplicación en ejecución.

Aplicaciones adicionales incluidas en C# para Programadores, 2a. edición

La figura 1.6 lista unas cuantas aplicaciones de ejemplo que introducen algunas de las poderosas y divertidas herramientas de C#. Le recomendamos que practique ejecutando algunas de ellas. La carpeta de ejemplos del

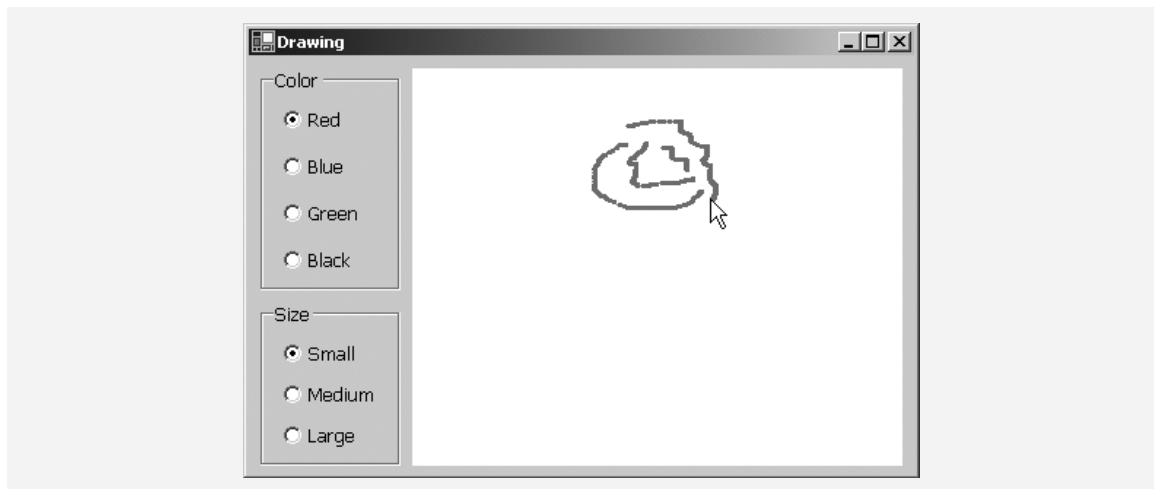


Figura 1.3 | Dibuje con un nuevo color de brocha.

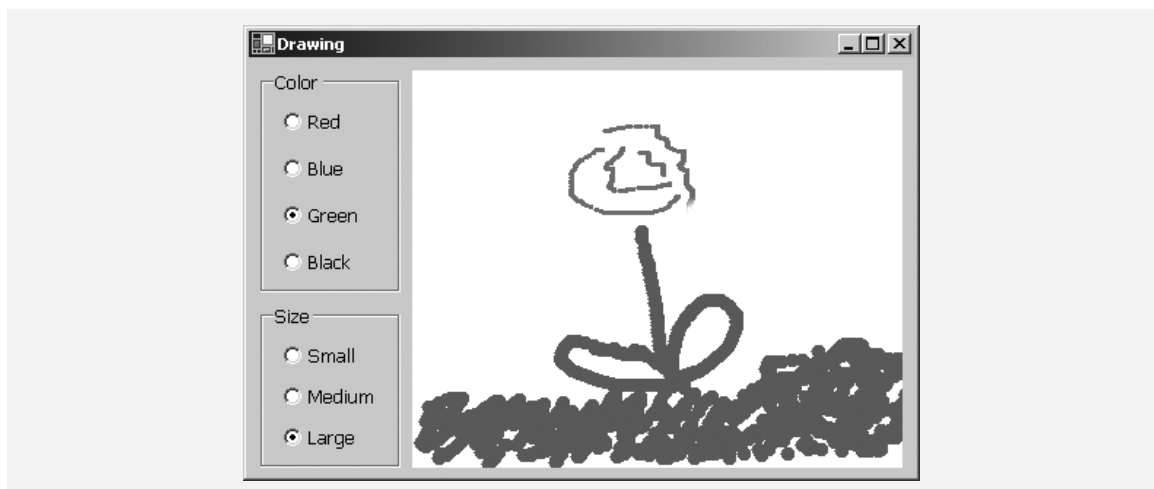


Figura 1.4 | Dibuje con un nuevo tamaño de brocha.

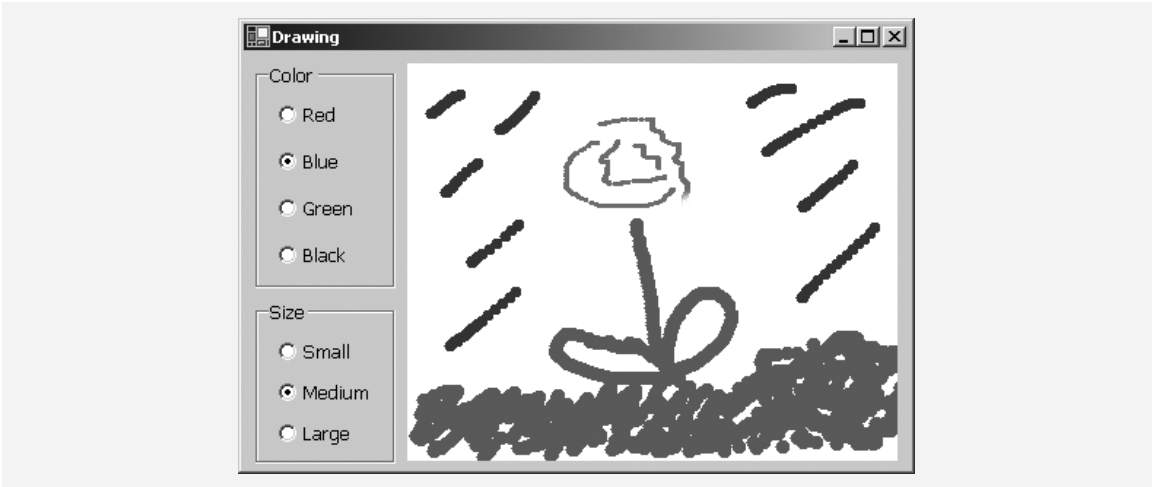


Figura I.5 | Termine el dibujo.

capítulo 1 contiene todos los archivos requeridos para ejecutar cada una de las aplicaciones que se listan en la figura 1.6. Sólo haga doble clic en el nombre de archivo de cualquier aplicación que desee ejecutar. [Nota: la aplicación Garage.exe asume que el usuario introduce un valor entre 0 y 24.]

1.9 (Única sección obligatoria del caso de estudio)

Caso de estudio de ingeniería de software: introducción a la tecnología de objetos y el UML

Ahora comenzaremos nuestra introducción anticipada a la orientación a los objetos, una manera natural de pensar acerca del mundo y de escribir programas de computadora. En un momento en el que hay cada vez más demanda por software nuevo y más poderoso, la habilidad de crear software con rapidez, economía y sin errores sigue siendo una meta difícil. Este problema puede tratarse en parte a través del uso de los *objetos*, los cuales son *componentes* de software reutilizables que modelan elementos en el mundo real. Un enfoque modular orientado a objetos para el diseño y la implementación puede hacer que los grupos de desarrollo de software sean más productivos de lo que es posible mediante el uso de las primeras técnicas de programación. Lo que es más, los programas orientados a objetos son más fáciles de comprender, corregir y modificar.

Los capítulos 1, 3-9 y 11 terminan con una sección breve titulada “Caso de estudio de ingeniería de software”, en la cual presentamos, a un ritmo cómodo, una introducción a la orientación a objetos. Nuestro objetivo aquí es ayudarlo a desarrollar una manera de pensar orientada a objetos y de presentarle el *Lenguaje Unificado de*

Nombre de la aplicación	Archivo a ejecutar
Cuotas de estacionamiento	Garage.exe
Tic Tac Toe	TicTacToe.exe
Dibujar estrellas	DrawStars.exe
Dibujar figuras	DrawShapes.exe
Dibujar polígonos	DrawPolygons.exe

Figura I.6 | Ejemplos de programas en C# incluidos en este libro.

Modelado™ (UML™): un lenguaje gráfico que permite a las personas que diseñan sistemas de software orientado a objetos utilizar una notación estándar en la industria para representarlos.

En esta sección, que es la única obligatoria del caso de estudio, presentaremos los conceptos básicos de la orientación a objetos y su terminología relacionada. Las secciones opcionales en los capítulos 3-9 y 11 presentan un diseño orientado a objetos y la implementación del software para un sistema de cajero automático (ATM) simple. Las secciones tituladas “Caso de estudio de ingeniería de software” al final de los capítulos 3-9:

- analizan un documento de requerimientos típico, el cual describe cómo se creará un sistema de software (el ATM).
- determinan los objetos requeridos para implementar el sistema.
- determinan los atributos que tendrán los objetos.
- determinan los comportamientos que exhibirán los objetos.
- especifican cómo interactuarán los objetos entre sí para cumplir con los requerimientos del sistema.

Las secciones tituladas “Caso de estudio de ingeniería de software” al final de los capítulos 9 y 11 modifican y mejoran el diseño presentado en los capítulos 3-8. El apéndice J contiene una implementación completa y funcional en C# del sistema ATM orientado a objetos.

Aunque nuestro caso de estudio es una versión simplificada de un problema de nivel industrial, de todas formas cubrimos muchas prácticas comunes en la industria. Usted experimentará una sólida introducción al diseño orientado a objetos con el UML. Además, agudizará sus habilidades de lectura de código al dar un vistazo a una implementación completa, directa y bien documentada en C# del ATM.

Conceptos básicos de la tecnología de los objetos

Comenzaremos nuestra introducción a la orientación a objetos con cierta terminología clave. Cada vez que voltee a su alrededor encontrará objetos: personas, animales, plantas, autos, aviones, edificios, computadoras, etcétera. Los humanos piensan en términos de los objetos. Los teléfonos, las casas, los semáforos, los hornos de microondas y los enfriadores de agua son sólo algunos objetos más que vemos a nuestro alrededor todos los días.

En ocasiones dividimos a los objetos en dos categorías: animados e inanimados. Los objetos animados están “vivos” en cierto sentido; se mueven a su alrededor y hacen cosas. Los objetos inanimados no se mueven por cuenta propia. No obstante, ambos tipos de objetos tienen cosas en común. Todos tienen *atributos* (por ejemplo: tamaño, forma, color y peso) y todos exhiben *comportamientos* (es decir, una pelota rueda, rebota, se infla y se desinfla; un bebé llora, duerme, gatea, camina y parpadea; un auto acelera, frena y da vuelta; una toalla absorbe agua). Estudiamos los tipos de atributos y comportamientos que tienen los objetos de software.

Los humanos aprenden acerca de los objetos al estudiar sus atributos y observar sus comportamientos. Distintos objetos pueden tener atributos similares y exhibir comportamientos similares. Por ejemplo, pueden hacerse comparaciones entre los bebés y los adultos, y entre los humanos y los chimpancés.

El *diseño orientado a objetos (DOO)* modela el software en términos similares a los que utilizan las personas para modelar los objetos del mundo real. Aprovecha las relaciones de las clases, en donde los objetos de cierta clase, como una clase de vehículos, tienen las mismas características: los autos, los camiones, las pequeñas vagonetas rojas y los patines tienen mucho en común. El DOO aprovecha las relaciones de *herencia*, en donde se derivan nuevas clases de objetos al absorber las características de las clases existentes y agregar características únicas para cada objeto. Sin duda un objeto de la clase “convertible” tiene las características de la clase “automóvil” que es más general, pero el techo se quita y se pone de manera específica.

El diseño orientado a objetos ofrece una manera natural e intuitiva de ver el proceso de diseño del software; a saber, se modelan los objetos por sus atributos, comportamientos e interrelaciones, de igual forma que describimos los objetos del mundo real. El DOO también modela la comunicación entre los objetos. Así como las personas se envían mensajes unas con otras (por ejemplo, un sargento ordena a un soldado que se ponga en posición firme, o un adolescente envía mensajes de texto a un amigo para verse en el cine), los objetos también se comunican a través de mensajes. Un objeto tipo cuenta de banco puede recibir un mensaje para reducir su saldo por cierta cantidad, debido a que el cliente ha retirado esa cantidad de dinero.

El DOO *encapsula* (es decir, envuelve) los atributos y las *operaciones* (comportamientos) en los objetos; los atributos y las operaciones de un objeto están enlazadas íntimamente. Los objetos tienen la propiedad conocida como *ocultamiento de información*. Esto significa que pueden saber cómo comunicarse unos con otros a través de *interfaces* bien definidas, pero por lo general no se les permite saber cómo se implementan otros objetos; los detalles de implementación se ocultan dentro de los mismos objetos. Por ejemplo, usted puede conducir un auto a la perfección sin necesidad de conocer los detalles de cómo funcionan los motores, las transmisiones, los frenos y los sistemas de escape de manera interna; lo importante es que sepa cómo utilizar el pedal del acelerador, el del freno, el volante y lo demás. Como veremos más adelante, el ocultamiento de la información es imprescindible para la buena ingeniería de software.

Los lenguajes como C# están *orientados a objetos*. La programación en un lenguaje de este tipo se llama *programación orientada a objetos* (POO), y permite a los programadores de computadoras implementar en forma conveniente un diseño orientado a objetos como un sistema de software funcional. Por otro lado, los lenguajes como C son *por procedimientos*, de manera que la programación tiende a estar *orientada a las acciones*. En C, la unidad de programación es la *función*. En C#, la unidad de programación es la *clase*, a partir de la cual los objetos se *instancian* (un término del DOO para indicar la “creación”). Las clases en C# contienen *métodos* (el equivalente de C# para las funciones de C) que implementan las operaciones y datos que implementan los atributos.

Clases, miembros de datos y métodos

Los programadores de C# se concentran en crear sus propios *tipos definidos por el usuario*, a los cuales se les llama clases. Cada clase contiene datos y también el conjunto de métodos que manipulan esos datos y proporcionan servicios a los *clientes* (es decir, otras clases que utilizan esa clase). Los componentes de datos de una clase se llaman atributos, o *campos*. Por ejemplo, una clase tipo cuenta bancaria podría incluir un número de cuenta y un balance. Los componentes de operación de la clase se llaman métodos. Por ejemplo, una clase tipo cuenta bancaria podría incluir métodos para hacer un depósito (aumentar el saldo), hacer un retiro (reducir el saldo) y solicitar el saldo actual. El programador utiliza tipos integrados (y otros tipos definidos por el usuario) como “bloques de construcción” para crear nuevos tipos definidos por el usuario (clases). Los *sustantivos en la especificación de un sistema* ayudan al programador de C# a determinar el conjunto de clases a partir de las cuales se crean los objetos que trabajan en conjunto para implementar el sistema.

Las clases son para los objetos como los planos para las casas; una clase es un “plano” para crear objetos de esa clase. Así como podemos construir muchas casas a partir de un plano, también podemos instanciar (crear) muchos objetos de una clase. Usted no puede preparar alimentos en la cocina de un plano, pero sí puede prepararlos en la cocina de una casa. No puede dormir en la recámara de un plano, pero sí puede hacerlo en la recámara de una casa.

Las clases pueden tener relaciones con otras clases. Por ejemplo, en un diseño orientado a objetos de un banco, la clase “cajero” necesita relacionarse con otras clases, como la clase “cliente”, la clase “caja registradora”, la clase “caja fuerte”, y así en lo sucesivo. A estas relaciones se les conoce como *asociaciones*.

El empaquetamiento de software en forma de clases hace posible que los futuros sistemas de software *reutilicen* esas clases. A menudo los grupos de clases relacionadas se empaquetan como *componentes* reutilizables. Así como los agentes de bienes raíces dicen con frecuencia que los tres factores más importantes que afectan al precio de los bienes raíces son “la ubicación, la ubicación y la ubicación”, también algunas personas en la comunidad de desarrollo de software dicen a menudo que los tres factores más importantes que afectan al futuro del desarrollo del software son “la reutilización, la reutilización y la reutilización”.



Observación de ingeniería de software I.1

La reutilización de las clases existentes al crear nuevas clases y programas ahorra tiempo, dinero y esfuerzo. La reutilización también ayuda a los programadores a crear sistemas más confiables y efectivos, ya que por lo general las clases y los componentes existentes han pasado por un proceso exhaustivo de prueba, depuración y puesta a punto para el rendimiento.

Sin duda, con la tecnología de objetos usted podrá crear la mayor parte del nuevo software que necesite mediante la combinación de clases existentes, de igual manera que los fabricantes de automóviles combinan las piezas intercambiables. Cada nueva clase que usted cree tendrá el potencial de convertirse en un valioso patrimonio de software que usted y otros programadores podrán reutilizar para agilizar y mejorar la calidad de los esfuerzos de desarrollo del software a futuro.

Introducción al análisis y diseño orientados a objetos (A/DOO)

Pronto usted estará escribiendo programas en C#. ¿Cómo creará el código para sus programas? Tal vez, como muchos programadores novatos, sólo encenderá su computadora y empezará a escribir. Este método puede funcionar para programas pequeños (como los que presentaremos en los primeros capítulos del libro), pero ¿qué tal si se le pidiera que creara un sistema de software para controlar miles de cajeros automáticos para un banco importante? O ¿qué pasaría si se le pidiera que trabajara como parte de un equipo de 1,000 desarrolladores de software para crear la siguiente generación del sistema de control de tráfico aéreo de Estados Unidos? Para proyectos tan grandes y complejos no puede tan sólo sentarse y empezar a escribir programas.

Para crear las mejores soluciones es conveniente que siga un proceso detallado para *analizar* los *requerimientos* de su proyecto (es decir, determinar *qué* es lo que hará el sistema) y desarrollar un *diseño* que lo satisfaga (es decir, debe decidir *cómo* lo hará su sistema). En teoría, usted pasaría por este proceso y revisaría con cuidado el diseño (y haría que otros profesionales de software revisaran también su diseño) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a los objetos, se llama *análisis y diseño orientados a objetos (A/DOO)*. Los programadores experimentados saben que un proceso apropiado de análisis y diseño puede ahorrarles muchas horas, ya que ayuda a evitar un método de diseño del sistema mal planeado que tendría que abandonarse a mitad de su implementación, con la posibilidad de desperdiciar una cantidad considerable de tiempo, dinero y esfuerzo.

A/DOO es el término genérico para el proceso de analizar un problema y desarrollar un método para resolverlo. Los pequeños problemas como los que veremos en los primeros capítulos de este libro no requieren de un proceso exhaustivo de A/DOO. Podría ser suficiente, antes de empezar a escribir código en C#, con escribir *seudocódigo*: un medio informal basado en texto para expresar la lógica de un programa. En realidad no es un lenguaje de programación, pero puede utilizarlo como un tipo de plan general para guiarlo a la hora que escriba su código. En el capítulo 5 hablaremos sobre el pseudocódigo.

A medida que los problemas y los grupos de personas que los tratan de resolver aumentan en tamaño, el A/DOO se vuelve con rapidez más apropiado que el pseudocódigo. En teoría, un grupo debería acordar un proceso definido de manera estricta para resolver su problema y una manera uniforme de comunicarse unos a otros los resultados de ese proceso. Aunque existen muchos procesos de A/DOO distintos, hay un solo lenguaje gráfico para comunicar los resultados de *cualquier* proceso de A/DOO que se ha empezado a utilizar mucho. Este lenguaje, conocido como Lenguaje Unificado de Modelado (UML), se desarrolló a mediados de la década de 1990 bajo la dirección inicial de tres metodologistas de software: Grady Booch, James Rumbaugh e Ivar Jacobson.

Historia del UML

En la década de 1980, aumentó el número de organizaciones que comenzaron a utilizar la POO para crear sus aplicaciones, y se desarrolló la necesidad de un proceso estándar de A/DOO. Muchos metodologistas (incluyendo a Grady Booch, James Rumbaugh e Ivar Jacobson) produjeron y promovieron por su cuenta procesos separados para satisfacer esta necesidad. Cada proceso tenía su propia notación, o “lenguaje” (en forma de diagramas gráficos), para conllevar los resultados del análisis (es decir, determinar *qué* es lo que se supone que debe hacer un sistema propuesto) y el diseño (es decir, determinar *cómo* debe implementarse un sistema propuesto para que haga lo que tiene que hacer).

A principios de 1990, varias organizaciones utilizaban sus propios procesos y notaciones únicas. Al mismo tiempo, estas organizaciones también querían utilizar herramientas de software que tuvieran soporte para sus procesos específicos. Los distribuidores de software se dieron cuenta que era difícil proveer herramientas para tantos procesos. Se necesitaban tanto una notación como un proceso estándar.

En 1994, James Rumbaugh se unió con Grady Booch en Rational Software Corporation (ahora una división de IBM) y ambos comenzaron a trabajar para unificar sus populares procesos. Poco después se les unió Ivar Jacobson. En 1996, el grupo publicó varias versiones del UML para la comunidad de ingeniería de software y solicitó retroalimentación. Casi al mismo tiempo, una organización conocida como el Grupo de Administración de Objetos™ (OMG™) solicitó presentaciones para un lenguaje de modelado común. La OMG (www.omg.org) es una organización sin fines de lucro que promueve la estandarización de las tecnologías orientadas a objetos mediante la publicación de lineamientos y especificaciones, como el UML. Varias empresas (entre ellas HP, IBM, Microsoft, Oracle y Rational Software) ya habían reconocido la necesidad de un lenguaje de modelado común.

En respuesta a la petición de proposiciones por parte de la OMG, estas compañías formaron el consorcio UML Partners; este consorcio desarrolló la versión 1.1 del UML y la envió a la OMG, quien aceptó la proposición y, en 1997, asumió la responsabilidad del mantenimiento y la revisión continuos del UML. A lo largo de este libro presentamos la terminología y la notación del UML 2, que se adoptaron recientemente.

¿Qué es el UML?

El *Lenguaje Unificado de Modelado* (UML) es el esquema de representación gráfica más utilizado para modelar sistemas orientados a objetos. Sin duda ha unificado los diversos esquemas de notación populares. Las personas que diseñan sistemas utilizan el lenguaje (en forma de diagramas, muchos de los cuales veremos a lo largo de nuestro caso de estudio del ATM) para modelar sus sistemas. En este libro utilizaremos diversos tipos populares de diagramas de UML.

Una característica atractiva del UML es su flexibilidad. El UML es *extensible* (es decir, capaz de mejorar mediante nuevas características) e independiente de cualquier proceso de A/DOO particular. Los modeladores de UML tienen la libertad de utilizar varios procesos al diseñar sistemas, pero todos los desarrolladores pueden ahora expresar sus diseños mediante un conjunto estándar de notaciones gráficas.

UML es un lenguaje gráfico lleno de características. En las subsiguientes (y opcionales) secciones de “Caso de estudio de ingeniería de software” que tratan sobre el desarrollo del software para un cajero automático (ATM), presentaremos un subconjunto conciso de estas características. Después utilizaremos este subconjunto para guiarlo a través de su primera experiencia de diseño con el UML. Utilizaremos algunas notaciones de C# en nuestros diagramas de UML para evitar la confusión y mejorar la legibilidad. En la práctica industrial, en especial con herramientas de UML que generan código de manera automática (una conveniente característica de muchas herramientas de UML), es probable que uno se adhiera más a las palabras clave y las convenciones de nomenclatura de UML para los atributos y las operaciones.

Este caso de estudio se desarrolló con mucho cuidado bajo la guía de varios revisores académicos y profesionales distinguidos. Es nuestro más sincero deseo que usted se divierta al trabajar con él. Si tiene dudas o preguntas, por favor envíe un correo electrónico a deitel@deitel.com y le responderemos a la brevedad.

Recursos del UML en Internet y Web

Para más información acerca de UML, consulte los siguientes sitios Web. Para sitios de UML adicionales, consulte los recursos de Internet y Web que se listan al final de la sección 3.10.

www.uml.org

Este sitio de recursos de UML del Grupo de Administración de Objetos (OMG) contiene los documentos de las especificaciones para el UML y otras tecnologías orientadas a objetos.

www.ibm.com/software/rational/uml

Ésta es la página de recursos de UML para IBM Rational, el sucesor de Rational Software Corporation (la compañía que creó UML).

Lecturas recomendadas

Se han publicado muchos libros acerca de UML. Los siguientes libros que recomendamos proporcionan información acerca del diseño orientado a objetos con UML.

- Arlow, J. e I. Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design, Second Edition*. Londres: Addison-Wesley, 2005.
- Fowler, M. *UML Distilled, Third Edition: Applying the Standard Object Modeling Language*. Boston: Addison-Wesley, 2004.
- Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language User Guide, Second Edition*. Upper Saddle River, NJ: Addison-Wesley, 2005.

Para libros adicionales acerca de UML, consulte las lecturas recomendadas que se listan al final de la sección 3.10 o visite www.amazon.com, www.bn.com y www.informIT.com. IBM Rational, antes Rational Software Corporation, también proporciona una lista de lecturas recomendadas para libros de UML en <http://www-306.ibm.com/software/rational/sw-library/#Books>.

Sección 1.9 Ejercicios de autoevaluación

1.1 Liste tres ejemplos de objetos del mundo real que no mencionamos. Para cada objeto, liste varios atributos y comportamientos.

- 1.2 El pseudocódigo es _____.
 a) otro término para A/DOO
 b) un lenguaje de programación que se utiliza para mostrar diagramas de UML
 c) un medio informal para expresar la lógica del programa
 d) un esquema de representación gráfica para modelar sistemas orientados a objetos
- 1.3 El UML se utiliza principalmente para _____.
 a) evaluar sistemas orientados a objetos
 b) diseñar sistemas orientados a objetos
 c) implementar sistemas orientados a objetos
 d) las opciones a y b

Respuestas a los ejercicios de autoevaluación de la sección 1.9

1.1 [Nota: las respuestas pueden variar.] a) Los atributos de una televisión son: el tamaño de la pantalla, el número de colores que puede visualizar y el canal y volumen actuales. Una televisión se enciende y se apaga, cambia de canal, muestra video y reproduce sonidos. b) Los atributos de una cafetera son: el volumen máximo de agua que puede contener, el tiempo requerido para elaborar una jarra de café y la temperatura del plato calentador debajo de la jarra de café. Una cafetera se enciende y se apaga, elabora café y lo calienta. c) Los atributos de una tortuga son: su edad, el tamaño de su concha y su peso. Una tortuga se arrastra, se mete en su concha, emerge de la misma y come vegetación.

- 1.2 c.
 1.3 b.

1.10 Conclusión

En este capítulo se presentaron los conceptos básicos de la tecnología de objetos, incluyendo: clases, objetos, atributos y comportamientos. Presentamos una breve historia de los sistemas operativos, como Microsoft Windows. Hablamos sobre la historia de Internet y Web. Presentamos la historia de la programación en C# y la iniciativa .NET de Microsoft, la cual le permite programar aplicaciones basadas en Internet y Web mediante el uso de C# (y otros lenguajes). Conoció los pasos para ejecutar una aplicación en C#. Probó una aplicación de C# de ejemplo, similar a los tipos de aplicaciones que aprenderá a programar en este libro. Aprendió acerca de la historia y el propósito de UML (el lenguaje gráfico estándar en la industria para modelar sistemas de software). Iniciamos nuestra presentación anticipada de los objetos y las clases con la primera de nuestras secciones “Caso de estudio de ingeniería de software” (y la única que es obligatoria). El resto de las secciones (todas opcionales) del caso de estudio utilizan el diseño orientado a objetos y el UML para diseñar el software para nuestro sistema de cajero automático simplificado. En el apéndice J presentamos la implementación completa del código en C# del sistema ATM.

En el siguiente capítulo utilizará el IDE (Entorno de Desarrollo Integrado) de Visual Studio para crear su primera aplicación en C# mediante el uso de las técnicas de la programación visual. También aprenderá acerca de las características de ayuda de Visual Studio.

1.11 Recursos Web

Sitio Web de Deitel & Associates

www.deitel.com/books/csharpforprogrammers2/index.html

El sitio Web de Deitel & Associates para este libro incluye vínculos a los ejemplos y otros recursos del mismo.

www.deitel.com

Visite este sitio para actualizaciones, correcciones y recursos adicionales para todas las publicaciones de Deitel.

www.deitel.com/newsletter/subscribe.html

Visite este sitio para suscribirse al boletín de correo electrónico gratuito *Deitel® Buzz Online*; aquí podrá seguir el programa de publicación de Deitel & Associates y recibirá actualizaciones sobre C# y este libro.

www.prenhall.com/deitel

El sitio Web de Prentice Hall para las publicaciones de Deitel. Incluye información detallada sobre los productos, capítulos de muestra y *Sitios Web complementarios (Companion Web Sites)* que contienen recursos específicos para cada libro y cada capítulo, tanto para los estudiantes como para los instructores.

Sitios Web de Microsoft

msdn.microsoft.com/vcsharp/default.aspx

El sitio del Centro de desarrollo de Microsoft Visual C# incluye información sobre el producto, descargas, tutoriales, grupos de Chat y mucho más. Incluye casos de estudio de las compañías que utilizan C# en sus negocios.

msdn2.microsoft.com/es-es/library/kx37x362.aspx

Sitio Web que muestra una descripción de Microsoft Visual C# en español. Incluye una introducción a Visual C#, cómo utilizar el IDE y escribir aplicaciones, y otras cosas más.

msdn.microsoft.com/vstudio/default.aspx

Visite este sitio para que aprenda más acerca de los productos y recursos de Microsoft Visual Studio.

msdn2.microsoft.com/es-es/library/ms310242.aspx

Sitio que habla sobre Visual Studio 2005 en español.

www.gotdotnet.com

Sitio Web para la Comunidad del .NET Framework de Microsoft. Incluye tableros de mensajes, un centro de recursos, programas de ejemplo y demás.

www.thespoke.net

Los estudiantes pueden chatear, publicar su código, calificar el código de otros estudiantes, crear centros y publicar preguntas en este sitio Web.

Recursos

www.ecma-international.org/publications/standards/Ecma-334.html

La página de Ecma International para la Especificación del lenguaje C#.

www.w3.org

El Consorcio World Wide Web (W3C) desarrolla tecnologías para Internet y Web. Este sitio incluye vínculos a las tecnologías del W3C, noticias y preguntas frecuentes (FAQs).

www.error-bank.com

El Banco de errores es una colección de errores, excepciones y soluciones de .NET.

www.csharp-station.com

Este sitio Web ofrece noticias, vínculos, tutoriales, ayuda y otros recursos de C#.

www.mentores.net

Este sitio Web es una comunidad en línea sobre .NET, C#, ASP.NET, Visual Basic .NET, Programación y noticias de tecnología en español, que incita el compartir conocimientos entre “mentores” o colaboradores y usuarios de todo el mundo.

www.csharphelp.com

Este sitio incluye un foro de ayuda de C#, tutoriales y artículos.

www.codeproject.com/index.asp?cat=3

Esta página de recursos incluye código, artículos, noticias y tutoriales sobre C#.

www.dotnetpowered.com/languages.aspx

Este sitio Web proporciona una lista de los lenguajes implementados en .NET.

Recursos de UML

www.uml.org

Esta página de recursos de UML del Grupo de administración de objetos (OMG) proporciona los documentos de las especificaciones para el UML y otras tecnologías de orientación a objetos.

www.ibm.com/software/rational/uml

Ésta es la página de recursos de UML para IBM Rational: el sucesor de Rational Software Corporation (la compañía que creó UML).

Juegos en C#

www.c-sharpcorner.com/Games.asp

En este sitio Web encontrará una variedad de juegos desarrollados con C#. También puede enviar sus propios juegos para publicarlos en el sitio.

www.gamespp.com/cgi-bin/index.cgi?csharpsourcecode

Este sitio Web de recursos incluye juegos en C#, código fuente y tutoriales.

2

Introducción al IDE de Visual C# 2005 Express Edition

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Los fundamentos del Entorno Integrado de Desarrollo (IDE) de Visual Studio, el cual le ayudará a escribir, ejecutar y depurar sus programas en Visual C#.
- Las características de ayuda de Visual Studio.
- Los comandos clave contenidos en los menús y las barras de herramientas del IDE.
- El propósito de los diversos tipos de ventanas en el IDE de Visual Studio 2005.
- Qué es la programación visual y cómo simplifica y agiliza el desarrollo de programas.
- Cómo crear, compilar y ejecutar un programa simple en Visual C# para visualizar texto y una imagen mediante el uso del IDE de Visual Studio y la técnica de programación visual.

Hay que ver para creer.

—Proverbio

La forma siempre sigue a la función.

—Louis Henri Sullivan

La inteligencia ... es la facultad de crear objetos artificiales, en especial herramientas para crear herramientas.

—Henri-Louis Bergson

- 2.1 Introducción
- 2.2 Generalidades del IDE de Visual Studio 2005
- 2.3 Barra de menús y barra de herramientas
- 2.4 Navegación por el IDE de Visual Studio 2005
 - 2.4.1 Explorador de soluciones
 - 2.4.2 Cuadro de herramientas
 - 2.4.3 Ventana Propiedades
- 2.5 Uso de la Ayuda
- 2.6 Uso de la programación visual para crear un programa simple que muestra texto e imagen
- 2.7 Conclusión
- 2.8 Recursos Web

2.1 Introducción

Visual Studio® 2005 es el Entorno de Desarrollo Integrado (IDE) de Microsoft para crear, ejecutar y depurar programas (también conocidos como *aplicaciones*) escritos en una variedad de lenguajes de programación .NET. En este capítulo veremos las generalidades del IDE Visual Studio 2005 y demostraremos cómo crear un programa simple en Visual C# mediante el procedimiento de arrastrar y colocar bloques de construcción predefinidos en el lugar adecuado; a esta técnica se le llama *programación visual*. Este capítulo se dedica en específico a Visual C#, la implementación de Microsoft del estándar C# de Ecma.

2.2 Generalidades del IDE de Visual Studio 2005

Existen diversas versiones disponibles de Visual Studio; pero para este libro utilizamos *Microsoft Visual C# 2005 Express Edition*, la cual sólo tiene soporte para el lenguaje de programación Visual C#. Microsoft también ofrece una versión completa de Visual Studio 2005, que incluye soporte para otros lenguajes además de Visual C#, como Visual Basic y Visual C++. Nuestras capturas de pantalla y discusiones se enfocan en el IDE de Visual C# 2005 Express Edition. Supondremos que usted tiene cierta familiaridad con Windows.

Utilizaremos varios tipos de letra para diferenciar las características del IDE (como nombres y elementos de menús) y otros elementos que aparecen en el IDE. Enfatizaremos las características del IDE en un tipo de letra **Helvetica sans-serif negrita** (por ejemplo, menú **Archivo**) y resaltaremos los demás elementos, como los nombres de archivo (por ejemplo, `Form1.cs`) y los nombres de las propiedades (que veremos en la sección 2.4) en un tipo de letra **Lucida sans-serif**.

Introducción a Microsoft Visual C# 2005 Express Edition

Para iniciar Microsoft Visual C# 2005 Express Edition en Windows XP, seleccione **Inicio > Todos los programas > Microsoft Visual C# 2005 Express Edition**. Si utiliza Windows 2000, seleccione **Inicio > Programas > Microsoft Visual C# 2005 Express Edition**. Una vez que comience la ejecución de esta aplicación, aparecerá la **Página de inicio** (figura 2.1). Dependiendo de su versión de Visual Studio, su **Página de inicio** podría ser distinta. Para los nuevos programadores que no estén familiarizados con Visual C#, la **Página de inicio** contiene una lista de vínculos a recursos en el IDE de Visual Studio 2005 y en Internet. A partir de aquí nos referiremos al IDE de Visual Studio 2005 como “Visual Studio” o “el IDE”. Para los desarrolladores experimentados, esta página proporciona vínculos a los desarrollos más recientes en Visual C# (como actualizaciones y correcciones de errores) y a información sobre temas avanzados de programación. Una vez que empiece a explorar el IDE, para regresar a la **Página de inicio** sólo tiene que seleccionar la página del menú desplegable de la barra de dirección (figura 2.2), o también puede seleccionar **Ver > Otras ventanas > Página de inicio** o hacer clic en el icono **Página de inicio** de la **Barra de herramientas** del IDE (figura 2.9). En la sección 2.3 hablaremos sobre la **Barra de herramientas** y sus diversos iconos. Utilizamos el carácter **>** para indicar la selección de un comando de un menú. Por ejemplo, utilizamos la notación **Archivo > Abrir archivo** para indicar que debe seleccionar el comando **Abrir archivo** del menú **Archivo**.

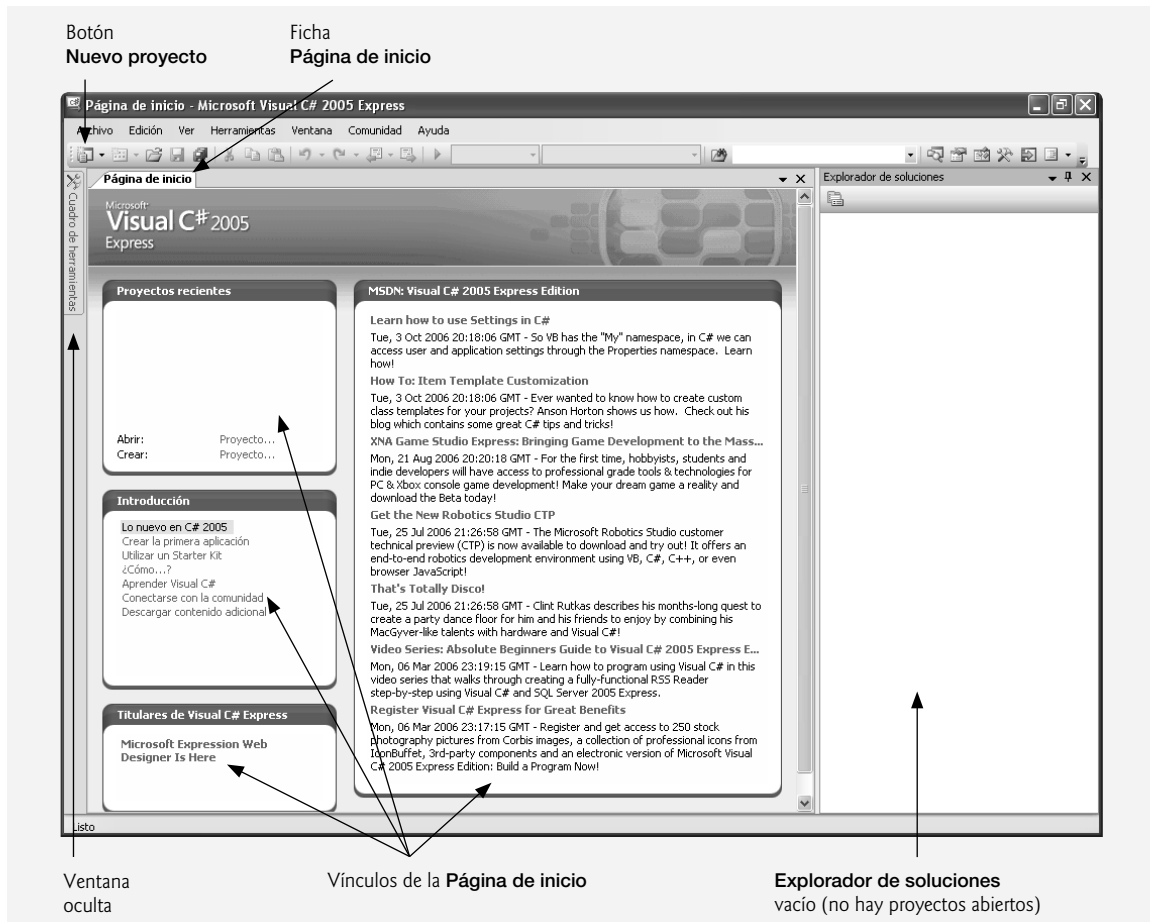


Figura 2.1 | La **Página de inicio** en Visual C# 2005 Express Edition.

Vínculos en la Página de inicio

Los vínculos de la **Página de inicio** se organizan en secciones: **Proyectos recientes**, **Introducción**, **Encabezados de Visual C# Express** y **MSDN: Visual C# 2005 Express Edition**; estas secciones contienen vínculos hacia recursos de programación útiles. Al hacer clic sobre cualquier vínculo de la **Página de inicio** se muestra la información asociada con ese vínculo específico. Para referirnos a la acción de *hacer un solo clic* con el botón izquierdo del ratón utilizaremos los términos *seleccionar* o *hacer clic*; para referirnos a la acción de hacer doble clic con el botón izquierdo del ratón usaremos el término *hacer doble clic*.

La sección **Proyectos recientes** contiene información sobre los proyectos que usted ha creado o modificado recientemente. También puede abrir los proyectos existentes o crear nuevos proyectos al hacer clic en los vínculos de esa sección. La sección **Introducción** se concentra en cómo utilizar el IDE para crear programas, aprender Visual C#, conectarse a la comunidad de desarrollo de Visual C# (es decir, otros desarrolladores de software con los cuales usted se puede comunicar a través de grupos de noticias y sitios Web) y proporcionar varias herramientas de desarrollo como los Starter Kits. Por ejemplo, al hacer clic en el vínculo **Utilizar un Starter Kit** obtendrá recursos y vínculos para crear una aplicación de protector de pantalla simple o una aplicación de colección de películas. La aplicación de protector de pantalla crea un protector de pantalla que visualiza artículos de noticias recientes. El Starter Kit de la colección de películas crea una aplicación que le permite mantener un catálogo de sus películas en DVD y VHS, o puede modificar la aplicación para que lleve el registro de cualquier otra cosa que usted desee coleccionar (por ejemplo, CDs, videojuegos).

Las secciones **Encabezados de Visual C# Express** y **MSDN: Visual C# 2005 Express Edition** proporcionan vínculos hacia información acerca de la programación en Visual C#, incluyendo un paseo por el lenguaje, nuevas



Figura 2.2 | Visualización de una página Web en Visual Studio.

características de Visual C# 2005 y cursos en línea. Para acceder a más información sobre Visual Studio, puede navegar por la biblioteca en línea *MSDN (Microsoft Developer Network)* en msdn.microsoft.com (inglés) y www.microsoft.com/spanish/msdn/latam (español). El sitio de MSDN contiene artículos, descargas y tutoriales sobre las tecnologías de interés para los desarrolladores de Visual Studio. También puede navegar a través de la Web desde el IDE, mediante el uso de Internet Explorer. Para obtener información detallada sobre C#, visite el Centro de desarrollo de C# en msdn.microsoft.com/vcsharp (inglés) y http://www.microsoft.com/spanish/msdn/centro_recursos/csharp/default.aspx (español). Para solicitar una página Web, escriba la URL en la barra de dirección (figura 2.2) y oprima *Intro*; desde luego que su computadora debe tener conexión a Internet. (Si la barra de dirección no se encuentra presente de antemano en el IDE, seleccione **Ver > Otras ventanas > Explorador Web**.) La página Web que desea ver aparecerá como otra *ficha*, la cual puede seleccionar dentro del IDE de Visual Studio (figura 2.2).

Personalización del IDE y creación de un nuevo proyecto

Para comenzar a programar en Visual C#, debe crear un nuevo proyecto o abrir uno existente; existen dos formas para hacerlo. Puede seleccionar **Archivo > Nuevo proyecto...** o **Archivo > Abrir proyecto...** en el menú **Archivo**, con lo cual se crea un nuevo proyecto o se abre uno existente, en forma respectiva. Desde la **Página de inicio**, bajo la sección **Proyectos recientes** puede también hacer clic en los vínculos **Crear: Proyecto** o **Abrir: Proyecto/Solución**. Un *proyecto* es un grupo de archivos relacionados, como el código de Visual C# y cualquier imagen que pueda conformar un programa. Visual Studio 2005 organiza los programas en proyectos y *soluciones*, que pueden contener uno o más proyectos. Las soluciones con múltiples proyectos se utilizan para crear programas de mayor escala. Cada uno de los programas que crearemos en este libro consiste de un solo proyecto.

Seleccione **Archivo > Nuevo proyecto...** o el vínculo **Crear: Proyecto...** en la **Página de inicio** para que aparezca el **cuadro de diálogo Nuevo proyecto** (figura 2.3). Los **cuadros de diálogo** son ventanas que facilitan la comunicación entre el usuario y la computadora. En unos momentos hablaremos sobre el proceso detallado para crear nuevos proyectos.

Visual Studio cuenta con plantillas para diversos tipos de proyectos (figura 2.3). Las **plantillas** son los tipos de proyectos que los usuarios pueden crear en Visual C#: aplicaciones para Windows, aplicaciones de consola y otras (en este libro de texto utilizaremos principalmente aplicaciones Windows y aplicaciones de consola). Los usuarios también pueden utilizar o crear plantillas de aplicación personalizadas. En este capítulo nos concentraremos en las aplicaciones para Windows. En el capítulo 3, Introducción a las aplicaciones de C#, hablaremos sobre la **Aplicación de consola**. Una **aplicación para Windows** es un programa que se ejecuta dentro de un sistema operativo Windows (por ejemplo, Windows 2000 o Windows XP) y por lo general tiene una **interfaz gráfica de usuario (GUI)**: la parte visual del programa con la que interactúa el usuario. Las aplicaciones para Windows incluyen productos de software de Microsoft como Microsoft Word, Internet Explorer y Visual Studio; los productos de software creados por otros distribuidores y el software personalizado que usted y otros programadores crean. En este libro creará muchas aplicaciones para Windows. [Nota: Novell patrocina un proyecto de código fuente abierto llamado **Mono**, el cual permite a los desarrolladores crear aplicaciones .NET para Linux, Windows y Mac OS X. Mono está basado en los estándares de Ecma para C# y la Infraestructura de lenguaje común (CLI). Para obtener más información sobre Mono, visite el sitio Web www.mono-project.com].

Visual Studio asigna de manera predeterminada el nombre **WindowsApplication1** al nuevo proyecto y solución (figura 2.3). Usted cambiará el nombre del proyecto y la ubicación en la que se almacena. Haga clic en **Aceptar** para mostrar el IDE en **vista de diseño** (figura 2.4), la cual contiene todas las características necesarias para que usted comience a crear programas. La porción de la vista de diseño del IDE se conoce también como **Diseñador de formularios de Windows**.

El rectángulo de color gris titulado **Form1** (al cual se le conoce como **Formulario**) representa la ventana principal de la aplicación Windows que usted creará. Las aplicaciones de C# pueden tener varios Formularios (ventanas); no obstante, la mayoría de las aplicaciones que usted creará en este texto sólo utilizan uno. Más adelante en este capítulo aprenderá a personalizar el Formulario mediante el proceso de agregar controles (es decir, componentes reutilizables) a un programa; en este caso son **Label** (etiqueta) y **PictureBox** (cuadro de imagen) (como se puede ver en la figura 2.27). Por lo general, un control **Label** contiene texto descriptivo (por ejemplo, "Bienvenido a Visual C#!") y un control **PictureBox** muestra imágenes como el insecto mascota de Deitel. Visual Studio cuenta con más de 65 controles preexistentes que usted puede utilizar para crear y personalizar sus programas. Muchos de

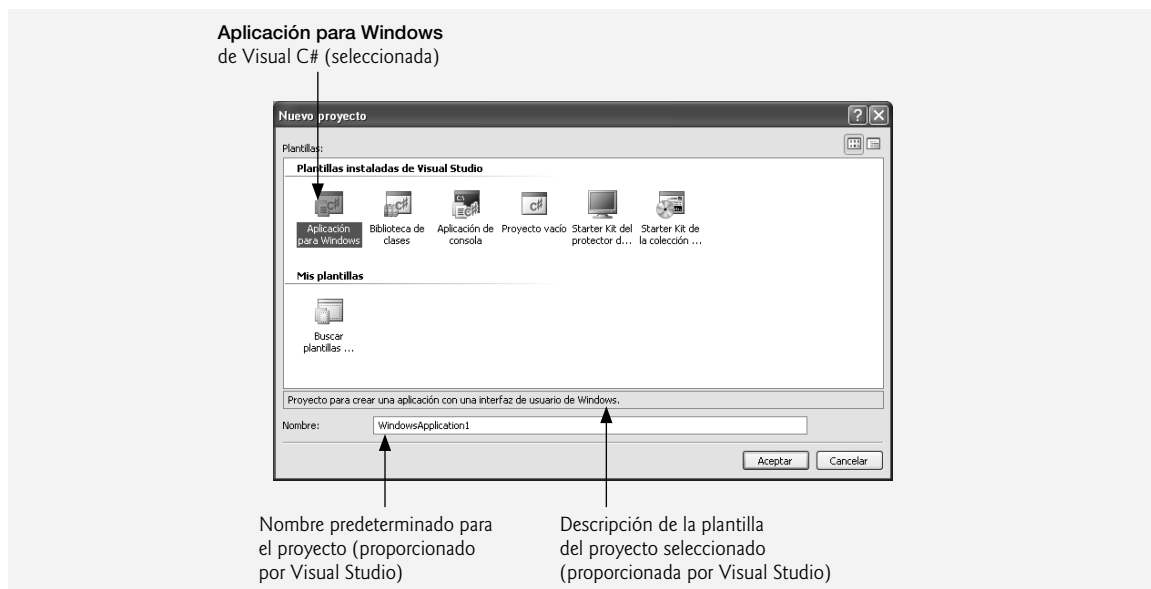


Figura 2.3 | Cuadro de diálogo **Nuevo proyecto**.

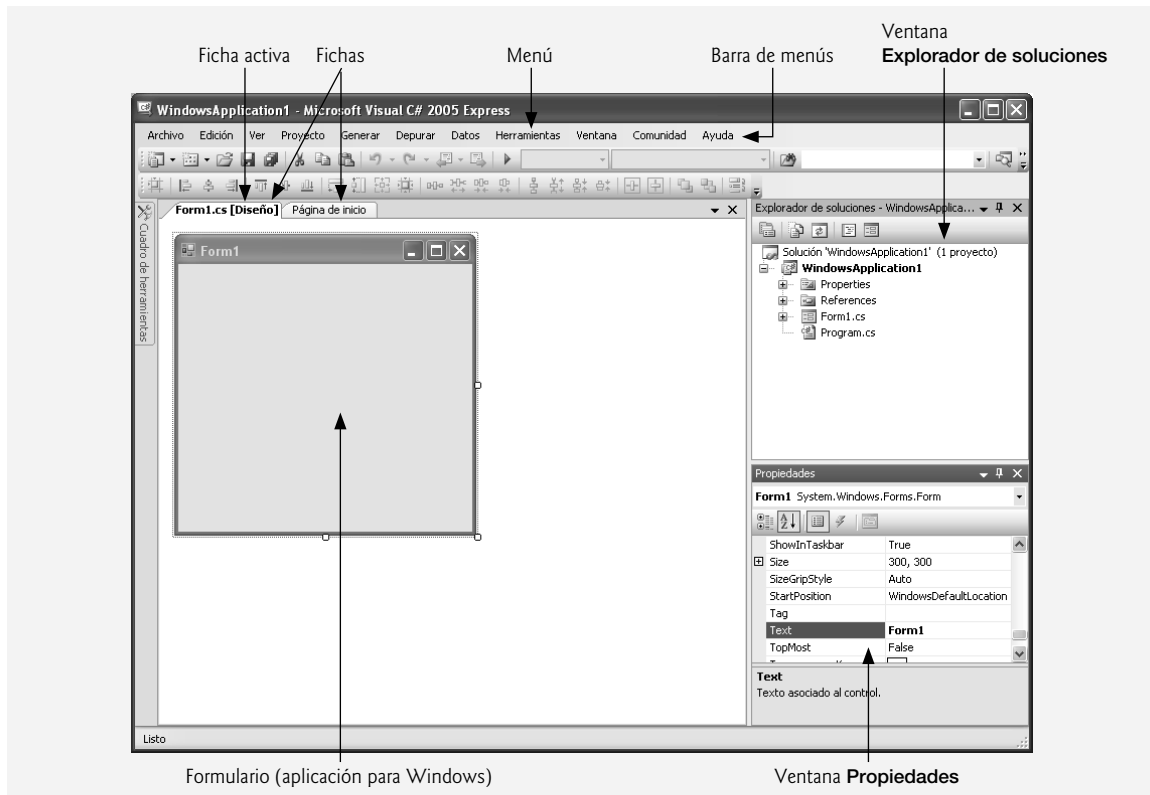


Figura 2.4 | Vista de **diseño** del IDE.

estos controles se definen y utilizan a lo largo de este libro. Además de los controles incluidos con Visual C# 2005 Express o Visual Studio 2005, hay muchos otros controles de terceros disponibles; puede descargarlos de la página Web msdn.microsoft.com/vcsharp/downloads/components/default.aspx.

Cuando comience a programar, trabajará con controles que no necesariamente formarán parte de su programa, sino que son parte de la FCL. A medida que coloque controles en el formulario, podrá modificar sus propiedades (que veremos con detalle en la sección 2.4) al introducir texto alternativo en un cuadro de texto (figura 2.5) o seleccionar opciones y después oprimir un botón, como **Aceptar** o **Cancelar**, como se muestra en la figura 2.6.

En conjunto, el formulario y los controles constituyen la GUI del programa. Los usuarios introducen datos (*entrada*) en el programa a través de una variedad de formas, como escribir en el teclado, hacer clic en los botones del ratón y escribir en los controles de la GUI como los cuadros de texto (TextBox). Los programas utilizan la GUI para mostrar instrucciones y demás información (*salida*) para que la lean los usuarios. Por ejemplo, el cuadro de diálogo **Nuevo proyecto** de la figura 2.3 presenta una GUI en la que el usuario hace clic en el botón del ratón para seleccionar una plantilla de proyecto y después escribe un nombre para el proyecto por medio del teclado (observe que la figura sigue mostrando el nombre predeterminado del proyecto **WindowsApplication1**, asignado por Visual Studio).

El nombre de cada documento abierto se lista en una ficha; en la figura 2.4, los documentos abiertos son **Form1.cs [Diseño]** y la **Página de inicio**. Para ver un documento, haga clic en su ficha. Las fichas facilitan el acceso cuando hay varios documentos abiertos. La *ficha activa* (la ficha del documento que se muestra en el IDE en un momento dado) se muestra con el texto en negrita (por ejemplo, **Form1.cs [Diseño]** en la figura 2.4) y se posiciona en frente de todas las demás fichas.

2.3 Barra de menús y barra de herramientas

Los comandos para administrar el IDE y para desarrollar, mantener y ejecutar programas están contenidos en *menús*, los cuales se encuentran en la *barra de menús* del IDE (figura 2.7). Observe que el conjunto de menús que se muestra en la figura 2.7 cambia en base al contexto.

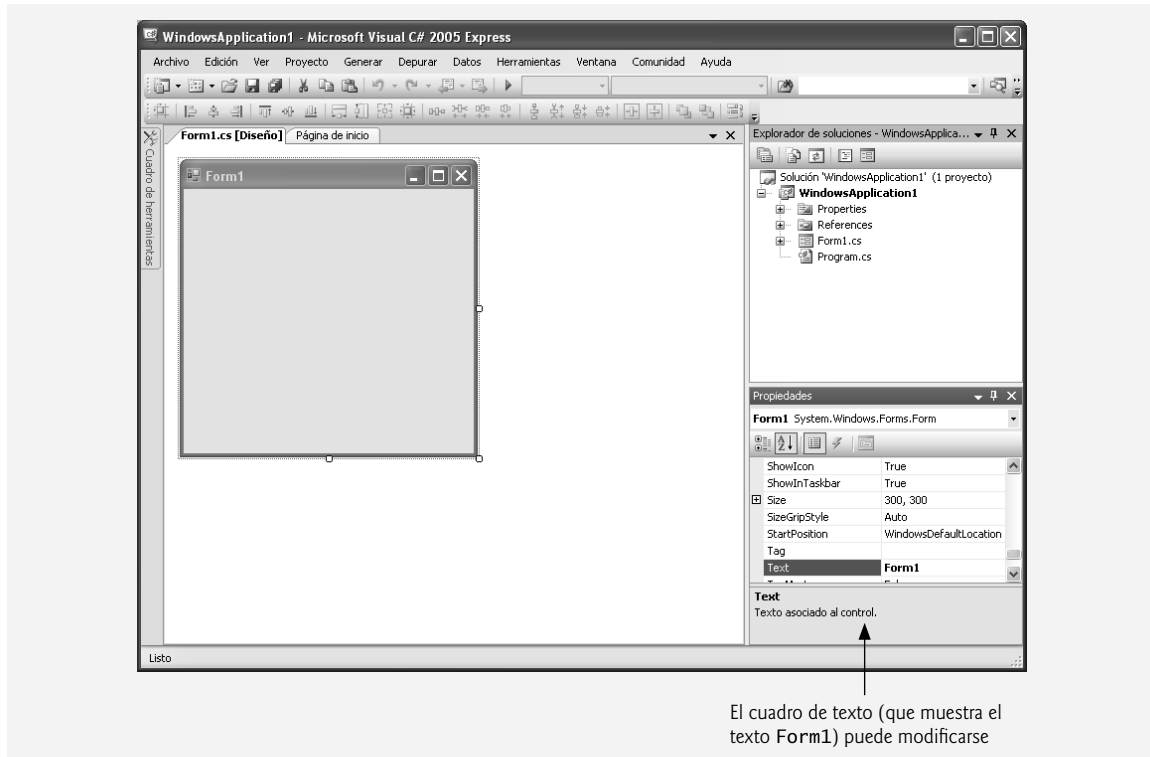


Figura 2.5 | Ejemplo de un control de cuadro de texto en el IDE de Visual Studio.

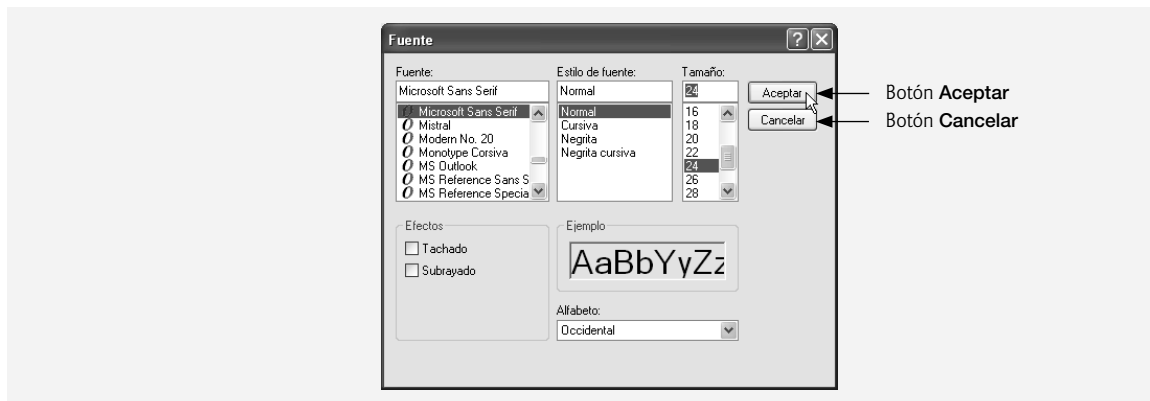


Figura 2.6 | Ejemplos de botones en el IDE de Visual Studio.

Los menús contienen grupos de comandos relacionados (también conocidos como *elementos de menú*) que, cuando se seleccionan, hacen que el IDE realice acciones específicas (es decir, abrir una ventana, almacenar un archivo, imprimir un proyecto y ejecutar un programa). Por ejemplo, para crear nuevos proyectos puede seleccionar **Archivo > Nuevo proyecto....** Los menús que se muestran en la figura 2.7 se sintetizan en la figura 2.8. En el capítulo 14, Conceptos de interfaz gráfica de usuario: parte 2, hablaremos sobre cómo crear y agregar menús y elementos de menú a sus programas.

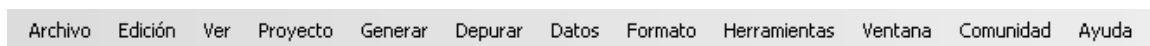


Figura 2.7 | Barra de menú de Visual Studio.

Menú	Descripción
Archivo	Contiene comandos para abrir, cerrar y guardar proyectos, así como para imprimir los datos de un proyecto y salir de Visual Studio.
Edición	Contiene comandos para editar programas, como cortar, copiar, pegar, deshacer, rehacer, eliminar, buscar y seleccionar.
Ver	Contiene comandos para mostrar ventanas (por ejemplo, Explorador de Windows , Cuadro de herramientas , ventana Propiedades) y para agregar barras de herramientas al IDE.
Proyecto	Contiene comandos para administrar proyectos y sus archivos.
Generar	Contiene comandos para compilar un programa.
Depurar	Contiene comandos para depurar (es decir, identificar y corregir los problemas en un programa) y ejecutar un programa. En el apéndice C hablaremos detalladamente sobre el proceso de depuración.
Datos	Contiene comandos para interactuar con bases de datos (es decir, colecciones organizadas de datos que se almacenan en las computadoras), que veremos en el capítulo 20, Bases de datos, SQL y ADO.NET.
Formato	Contiene comandos para organizar y modificar los controles de un formulario. El menú Formato sólo aparece cuando se selecciona un componente de la GUI en vista Diseño .
Herramientas	Contiene comandos para acceder a las herramientas adicionales del IDE (por ejemplo, el Cuadro de herramientas) y opciones que le permiten personalizar el IDE.
Ventana	Contiene comandos para organizar y visualizar ventanas.
Comunidad	Contiene comandos para enviar preguntas en forma directa a Microsoft, verificar el estado de las preguntas, enviar retroalimentación acerca de Visual C# y buscar en el centro de desarrollo CodeZone y en el sitio comunitario de desarrollo de Microsoft.
Ayuda	Contiene comandos para acceder a las características de ayuda del IDE.

Figura 2.8 | Resumen de los menús del IDE de Visual Studio 2005.

En vez de navegar por los menús de la barra de menús, puede acceder a muchos de los comandos más comunes desde la *barra de herramientas* (figura 2.9), la cual contiene gráficos llamados *iconos*, que representan comandos en forma gráfica. [Nota: la figura 2.9 divide la barra de herramientas en dos partes, para que podamos ilustrar los gráficos con mayor claridad; la barra de herramientas aparece en una línea dentro del IDE.] La barra de herramientas estándar se muestra de manera predeterminada cuando ejecutamos Visual Studio por primera vez; esta barra contiene iconos para los comandos que se utilizan con más frecuencia, como abrir un archivo, agregar un elemento a un proyecto, guardar y ejecutar (figura 2.9). Algunos comandos están deshabilitados al principio (es decir, no se pueden utilizar). Estos comandos, que en un principio aparecen en color gris, Visual Studio los habilita sólo cuando son necesarios. Por ejemplo, habilita el comando para guardar un archivo una vez que usted comienza a editarlo.

Puede personalizar el IDE agregando más barras de herramientas. Para ello, seleccione **Ver > Barras de herramientas** (figura 2.10). Cada barra de herramienta que seleccione se mostrará con las demás en la parte superior de la ventana de Visual Studio (figura 2.10). Otra forma en la que puede agregar barras de herramientas a su IDE (que no mostraremos en este capítulo) es a través de **Herramientas > Personalizar**. Después, en la ficha **Barras de herramientas** seleccione las que desea que aparezcan en el IDE.

Para ejecutar un comando por medio de la barra de herramientas, haga clic sobre su icono. Algunos iconos contienen una flecha hacia abajo que, cuando se hace clic sobre ella, muestra un comando o comandos relacionados, como se ilustra en la figura 2.11.

Es difícil recordar qué representa cada uno de los iconos en la barra de tareas. Puede posicionar el puntero del ratón sobre un icono para resaltarlo y que, después de unos instantes, aparezca una descripción, a lo cual se le conoce como *cuadro de información sobre herramientas* (*tool tip*) (figura 2.12). Estos cuadros ayudan a los programadores novatos a familiarizarse con las características del IDE, y sirven como recordatorios útiles de la funcionalidad de cada icono de la barra de herramientas.

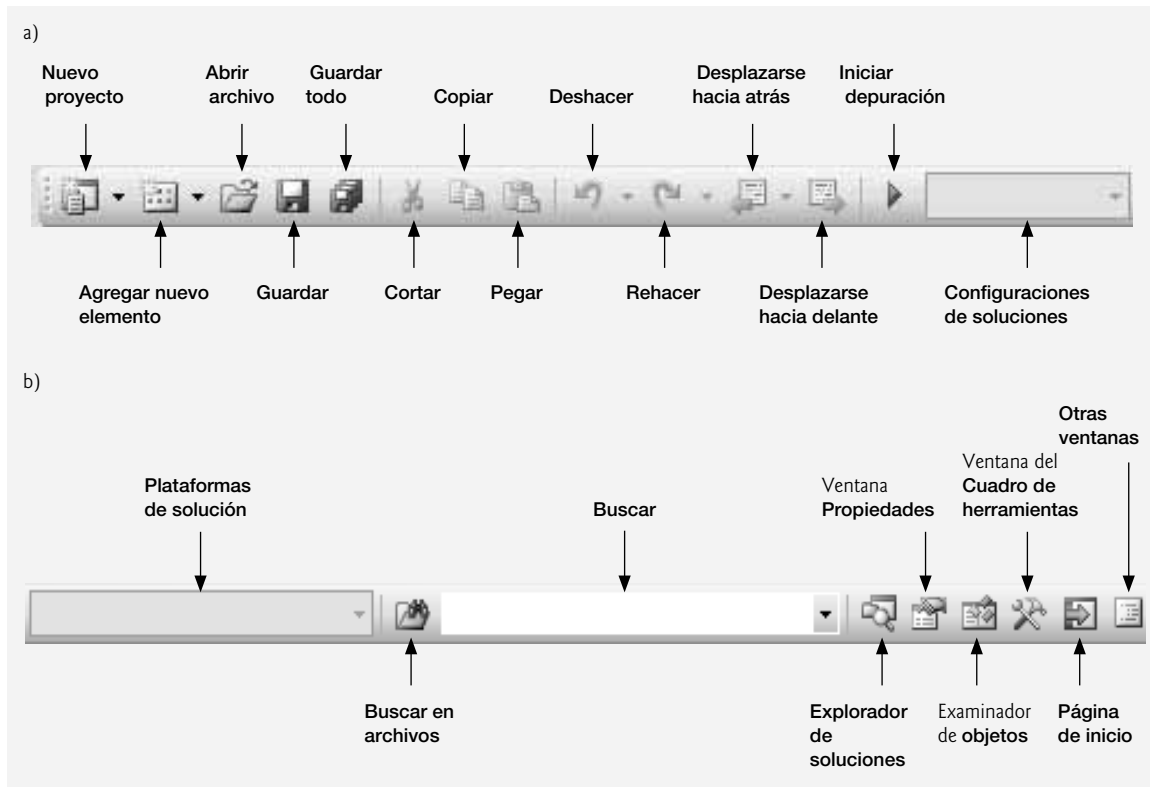


Figura 2.9 | La barra de herramientas estándar en Visual Studio.

2.4 Navegación por el IDE de Visual Studio 2005

El IDE ofrece ventanas para acceder a los archivos de proyecto y personalizar controles. En esta sección presentaremos algunas de las que usted utilizará con frecuencia al desarrollar programas en Visual C#. Puede acceder a estas ventanas a través de los iconos de la barra de herramientas (figura 2.13) o seleccionando el nombre de la ventana deseada en el menú **Ver**.

Visual Studio cuenta con una característica de ahorro de espacio llamada *ocultar automáticamente*. Cuando se habilita esta característica, aparece una ficha a lo largo del borde izquierdo o derecho de la ventana del IDE (figura 2.14). Esta ficha contiene uno o más iconos, cada uno de los cuales identifica a una ventana oculta. Si coloca el puntero del ratón encima de uno de estos iconos, se muestra esa ventana (figura 2.15). La ventana se oculta nuevamente cuando se mueve el ratón fuera de su área. Para “marcar” una ventana (es decir, para deshabilitar la característica de ocultarse automáticamente y mantenerla abierta), haga clic sobre el icono del marcador. Observe que cuando se habilita el ocultamiento automático, el icono del marcador está en posición horizontal (figura 2.15), mientras que cuando una ventana se “marca”, el icono del marcador está en posición vertical (figura 2.16).

En las siguientes tres secciones veremos las generalidades sobre tres de las principales ventanas que se utilizan en Visual Studio: el **Explorador de soluciones**, el **Cuadro de herramientas** y la ventana **Propiedades**. Estas ventanas muestran información acerca del proyecto e incluyen herramientas que le ayudarán a generar sus programas.

2.4.1 Explorador de soluciones

La ventana del **Explorador de soluciones** (figura 2.17) proporciona acceso a todos los archivos en una solución. Si esta ventana no aparece en el IDE, seleccione **Ver > Explorador de soluciones** o haga clic en el icono **Explorador de soluciones** (figura 2.13). La primera vez que abre Visual Studio, el **Explorador de soluciones** está vacío; no hay archivos que mostrar. Una vez que abre una solución, el **Explorador de soluciones** muestra el contenido de la solución y sus proyectos o cuando usted crea un nuevo proyecto, se muestra su contenido.



Figura 2.10 | Agregue la barra de herramientas **Generar** al IDE.

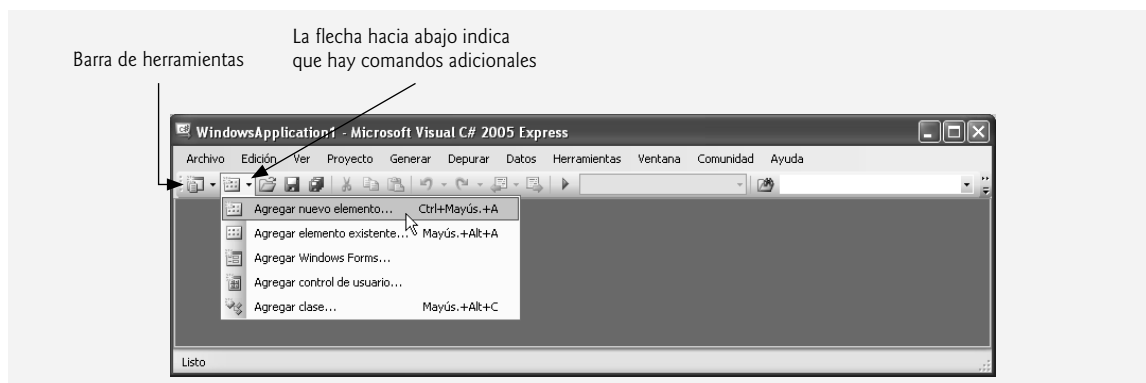


Figura 2.11 | Icono de la barra de herramientas del IDE que muestra comandos adicionales.

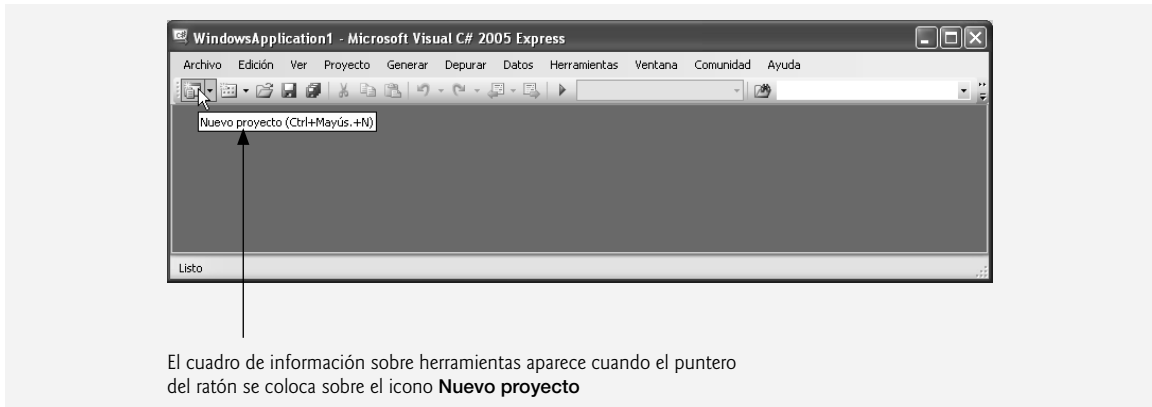


Figura 2.12 | Demostración del cuadro de información sobre herramientas (*tool tip*).

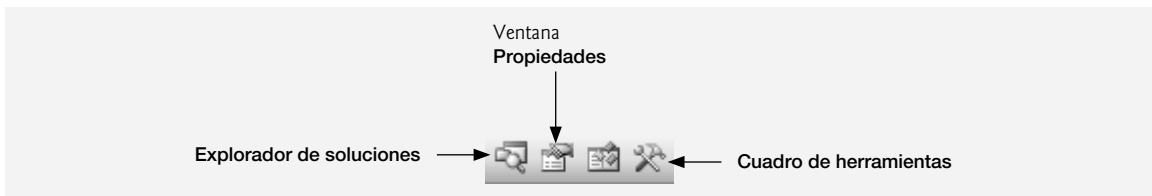


Figura 2.13 | Iconos de la barra de herramientas para tres ventanas de Visual Studio.

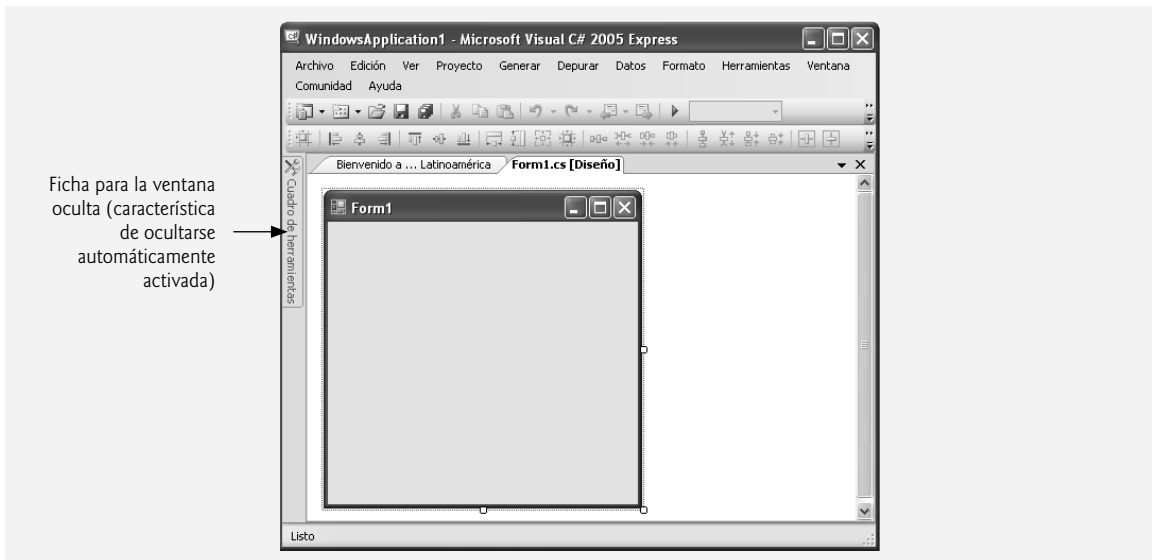


Figura 2.14 | Demostración de la característica de ocultarse automáticamente.

El *proyecto de inicio* de la solución es el proyecto que se ejecuta cuando corre el programa. Si usted tiene varios proyectos en una solución dada, puede especificar cuál de ellos será el de inicio, esto se logra al hacer clic con el botón derecho del ratón sobre el nombre del proyecto de la ventana del **Explorador de soluciones**; después seleccione la opción **Establecer como proyecto de inicio**. Para una solución con un solo proyecto, el proyecto de inicio es el único (en este caso, **WindowsApplication1**) y su nombre aparece en negrita en la ventana del **Explorador de soluciones**. Todos los programas que veremos en este libro son soluciones con un solo proyecto.

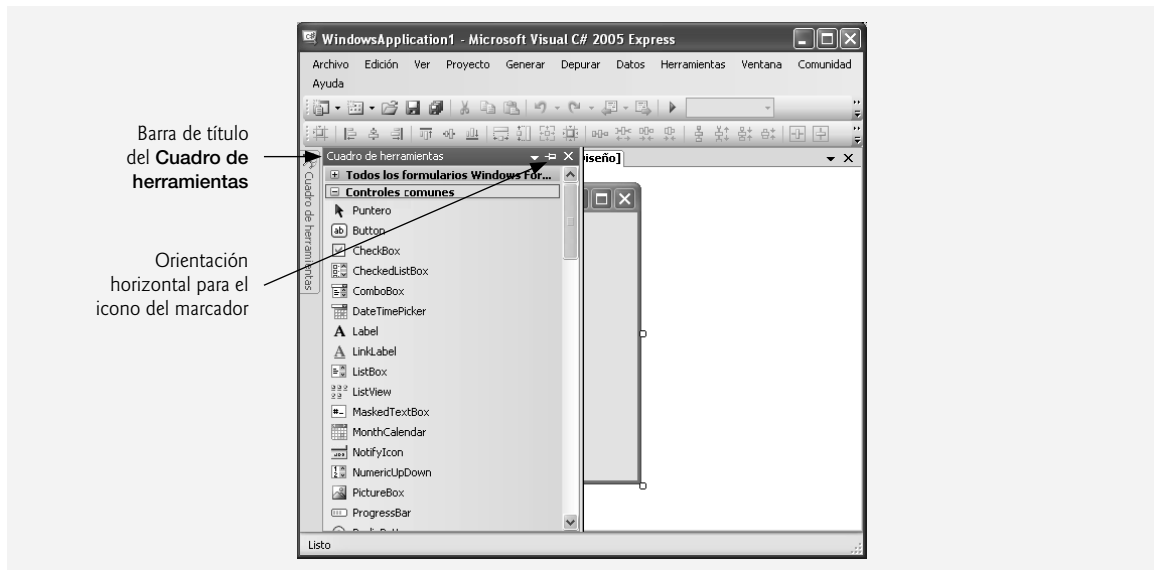


Figura 2.15 | Mostrar una ventana oculta cuando se habilita la característica de ocultarse automáticamente.

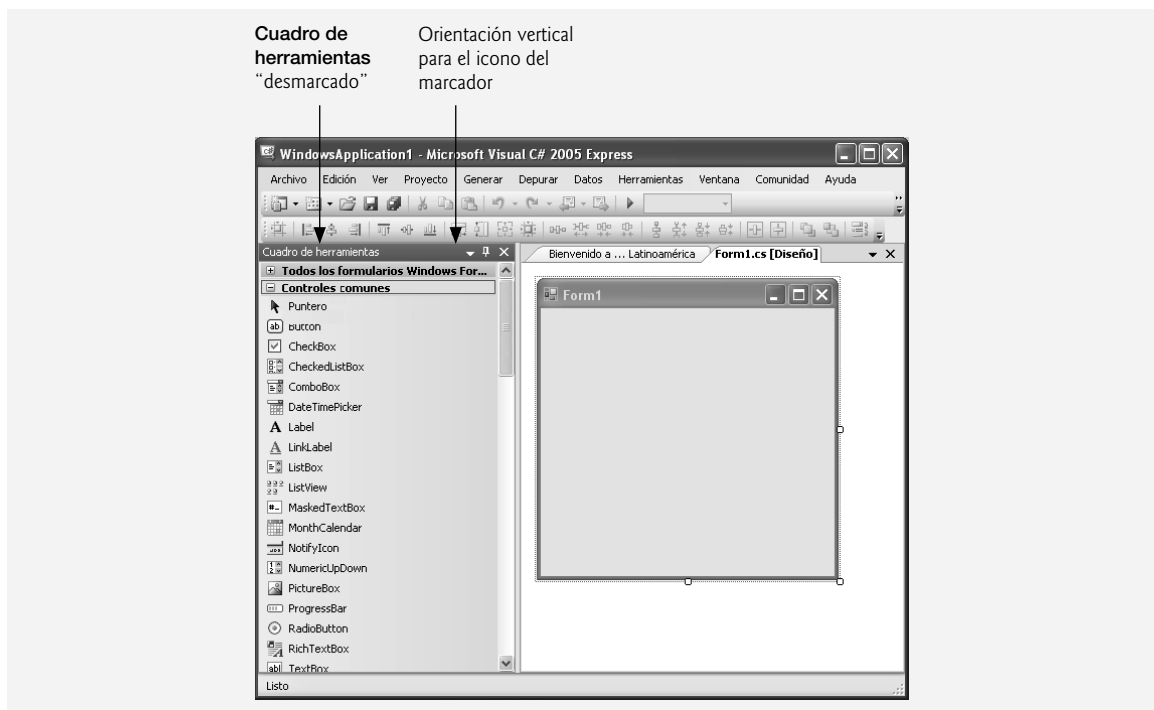


Figura 2.16 | Deshabilitar la característica de ocultarse automáticamente ("desmarcar" una ventana).

Para los programadores que utilizan Visual Studio por primera vez, la ventana del **Explorador de soluciones** sólo lista los archivos **Properties**, **References**, **Form1.cs** y **Program.cs** (figura 2.17). La ventana **Explorador de soluciones** incluye una barra de herramientas que contiene varios iconos.

El archivo de Visual C# que corresponde al formulario que se muestra en la figura 2.4 se llama **Form1.cs** (y es el que está seleccionado en la figura 2.17). (Los archivos de Visual C# utilizan la *extensión de nombre de archivo .cs*, abreviatura de "C Sharp").



Figura 2.17 | Explorador de soluciones con un proyecto abierto.

El IDE muestra de manera predeterminada sólo los archivos que probablemente necesiten editarse; los demás archivos generados por el IDE están ocultos. Al hacer clic en el *icono Mostrar todos los archivos* (figura 2.17) se muestran todos los archivos en la solución, incluyendo los generados por el IDE. Puede hacer clic en los cuadros de los signos más y menos que aparecen (figura 2.18) para expandir y colapsar el árbol de proyecto, en forma respectiva. Haga clic en el cuadro del signo más a la izquierda de **Properties** para mostrar los elementos agrupados bajo el titular a la derecha del cuadro de signo más (figura 2.19); haga clic en los cuadros de signo menos a la izquierda de **Properties** y **References** para colapsar el árbol de su estado expandido (figura 2.20). Otras ventanas de Visual Studio también utilizan esta convención de cuadros de signo más/signo menos.

2.4.2 Cuadro de herramientas

El **Cuadro de herramientas** contiene iconos que representan los controles utilizados para personalizar formularios (figura 2.21). Mediante el uso de la programación visual, podemos “arrastrar y soltar” controles en el formulario, lo cual es más rápido y simple que crearlos mediante la escritura de código de GUI (en el capítulo 5, Instrucciones de control: parte 1, presentaremos la escritura de este tipo de código). Así como no necesita saber cómo construir un motor para conducir un auto, tampoco necesita saber cómo crear controles para utilizarlos. La reutilización de controles preexistentes ahorra tiempo y dinero cuando se desarrollan programas. La amplia variedad de controles contenidos en el **Cuadro de herramientas** es una poderosa característica de la FCL de .NET. Más adelante en este capítulo utilizará el **Cuadro de herramientas** al crear su primer programa.

El **Cuadro de herramientas** contiene grupos de controles relacionados. Ejemplos de estos grupos son: **Todos los formularios Windows Forms**, **Controles comunes**, **Contenedores**, **Menús y barras de herramientas**, **Datos**, **Componentes**, **Impresión**, **Cuadros de diálogo** y **General**, los cuales se listan en la figura 2.21. Observe nuevamente el uso de los cuadros de signos más y menos para expandir o colapsar un grupo de controles. El **Cuadro de herramientas** contiene aproximadamente 65 controles preconstruidos para utilizarse en Visual Studio, por lo que tal vez tenga

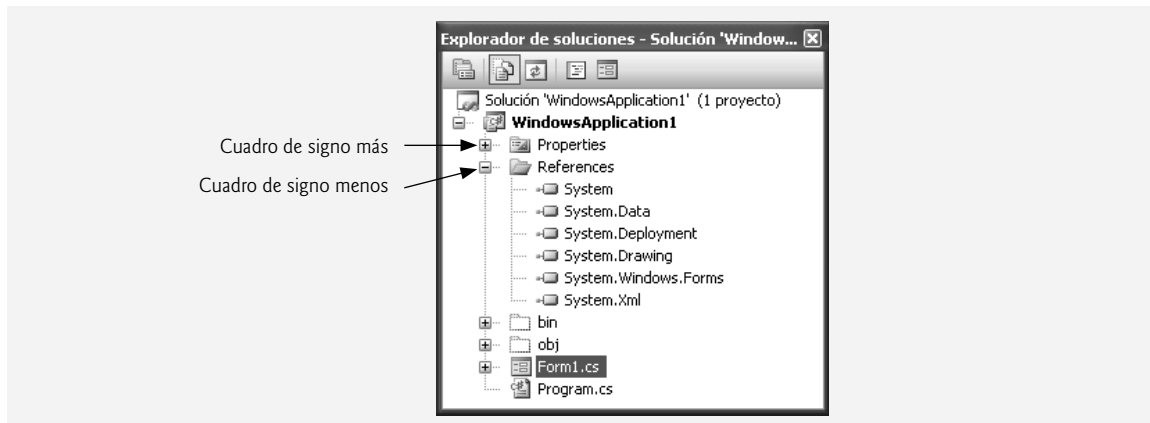


Figura 2.18 | El **Explorador de soluciones** muestra cuadros de signos más y menos para expandir y colapsar el árbol, de manera que se muestren o se oculten los archivos del proyecto.

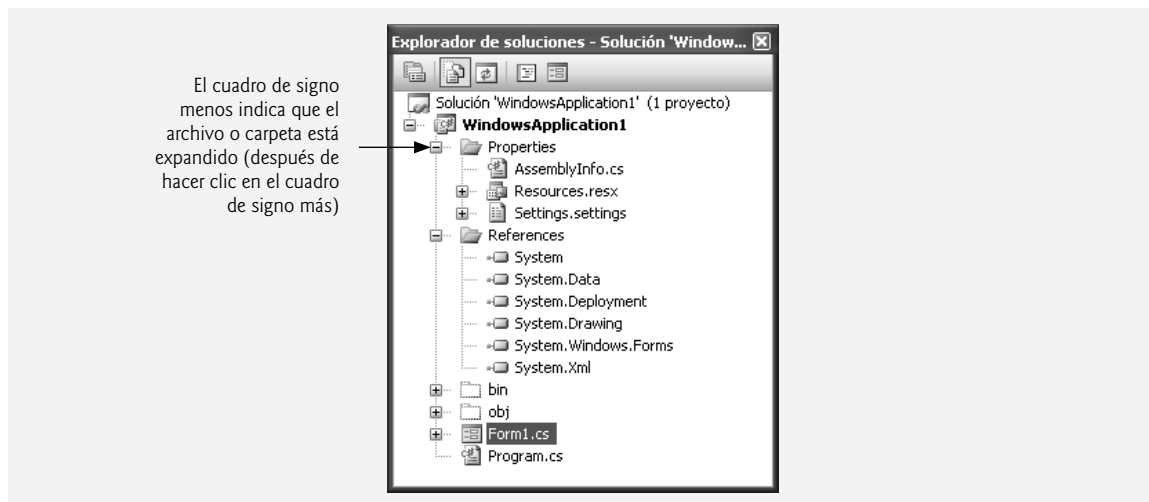


Figura 2.19 | El Explorador de soluciones expande el archivo **Properties** después de hacer clic en su cuadro de signo más.

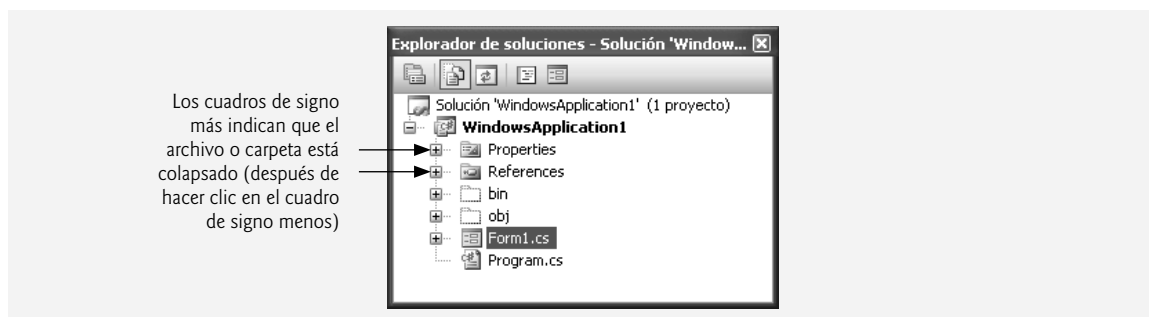


Figura 2.20 | El Explorador de soluciones colapsa todos los archivos después de hacer clic en cualquier cuadro de signo menos.

que desplazarse por el **Cuadro de herramientas** para ver controles adicionales a los que se muestran en la figura 2.21. A lo largo de este libro hablaremos sobre muchos de los controles del **Cuadro de herramientas** y su funcionalidad.

2.4.3 Ventana Propiedades

Para mostrar la ventana **Propiedades** en caso de que no esté visible, puede seleccionar **Ver > Ventana Propiedades**, hacer clic en el icono ventana **Propiedades** que se muestra en la figura 2.13 o puede oprimir la tecla **F4**. La **ventana Propiedades** muestra las propiedades para el formulario (Form) (figura 2.22), control o archivo que esté seleccionado en ese momento en vista de diseño. Las **propiedades** especifican información acerca del formulario o control, como su tamaño, color y posición. Cada formulario o control tiene su propio conjunto de propiedades; la descripción de una propiedad se muestra en la parte inferior de la ventana **Propiedades** cada vez que se selecciona esa propiedad.

La figura 2.22 muestra la ventana **Propiedades** del formulario. La columna izquierda lista las propiedades del Formulario; la columna derecha muestra el valor actual de cada propiedad. Los iconos en la barra de herramientas ordenan las propiedades, ya sea alfabéticamente al hacer clic en el icono **Alfabético** o por categorías, al hacer clic en el icono **Por categorías**. Puede ordenar de manera alfabética las propiedades en orden ascendente o descendente; al hacer clic en el icono **Alfabético** varias veces se alterna el orden de las propiedades entre A-Z y Z-A. Al utilizar el orden por categorías se agrupan las propiedades de acuerdo con su uso (por ejemplo, **Apariencia**, **Comportamiento**, **Diseño**). Dependiendo del tamaño de la ventana **Propiedades**, algunas de las propiedades pueden ocultarse de la vista en la pantalla, en cuyo caso los usuarios pueden desplazarse a través de la lista de propiedades. Más adelante en este capítulo le mostraremos cómo establecer propiedades individuales.

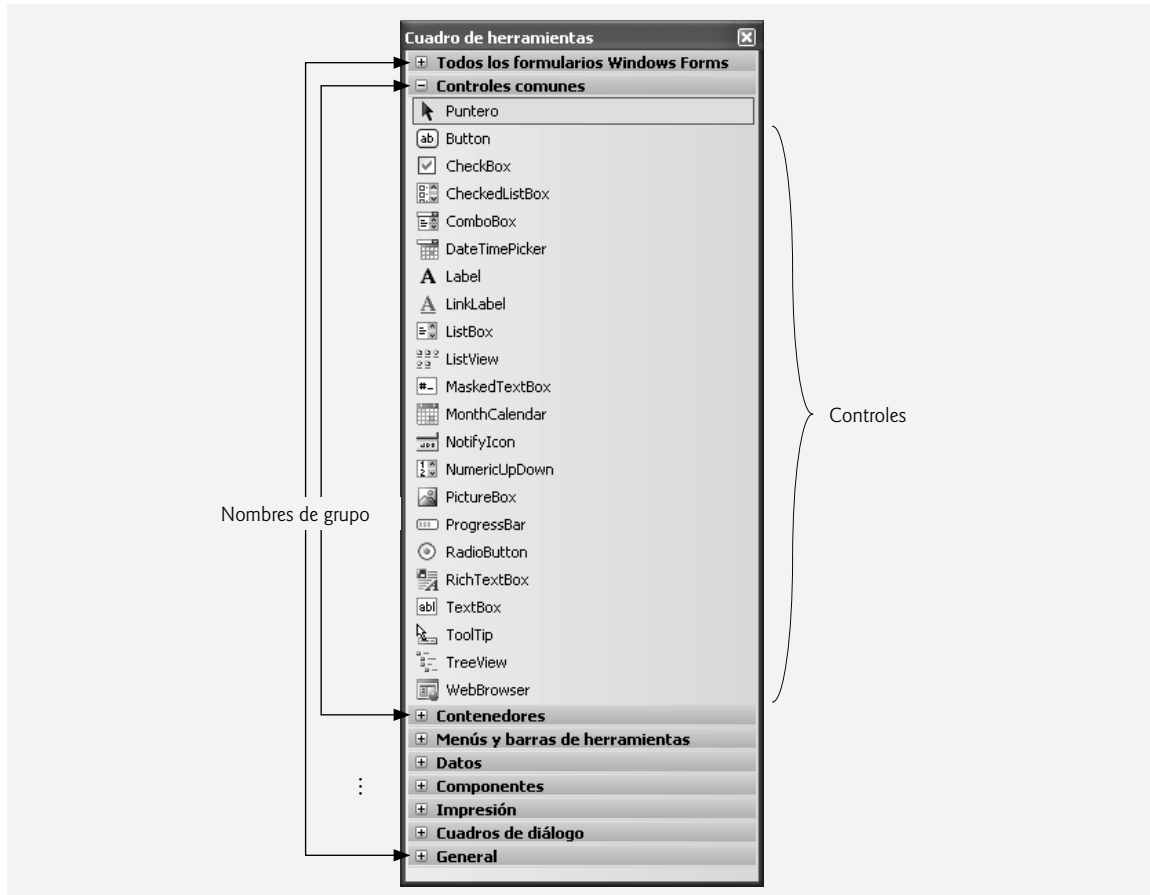


Figura 2.21 | Ventana del **Cuadro de herramientas**, en la cual se muestran los controles para el grupo **Controles comunes**.

La ventana **Propiedades** es imprescindible para la programación visual, ya que nos permite modificar las propiedades de un control en forma visual, sin necesidad de escribir código. Podemos ver qué propiedades están disponibles para modificarse y, cuando sea apropiado, podemos conocer el rango de valores aceptables para una propiedad dada. La ventana **Propiedades** muestra una breve descripción de la propiedad seleccionada, con lo cual nos ayuda a comprender su propósito. Podemos establecer una propiedad rápidamente a través de esta ventana; por lo general sólo se requiere un clic y no hay que escribir código.

En la parte superior de la ventana **Propiedades** está el *cuadro de lista desplegable de selección de componente*, la cual nos permite seleccionar el formulario o control cuyas propiedades deseamos mostrar en la ventana **Propiedades** (figura 2.22). Una manera alternativa de mostrar las propiedades de un control sin seleccionar el formulario o control en la GUI es mediante el cuadro de lista desplegable de selección de componente.

2.5 Uso de la Ayuda

Visual Studio cuenta con muchas características de ayuda. En la figura 2.23 se sintetizan los comandos del *menú Ayuda*.

La *ayuda dinámica* (figura 2.24) es una excelente y rápida forma para obtener información acerca del IDE y sus características. Proporciona una lista de artículos que corresponden al “contenido actual” (es decir, los elementos seleccionados). Para abrir la *ventana Ayuda dinámica*, seleccione **Ayuda > Ayuda dinámica**. Después, cuando haga clic en una palabra o componente (como en un formulario o control) aparecerán vínculos hacia artículos correspondientes en la ventana **Ayuda dinámica**. La ventana lista los temas de ayuda, ejemplos de código y demás información relevante. También hay una barra de herramientas que proporciona acceso a las características de ayuda **Cómo**, **Buscar**, **Índice** y **Contenido**.

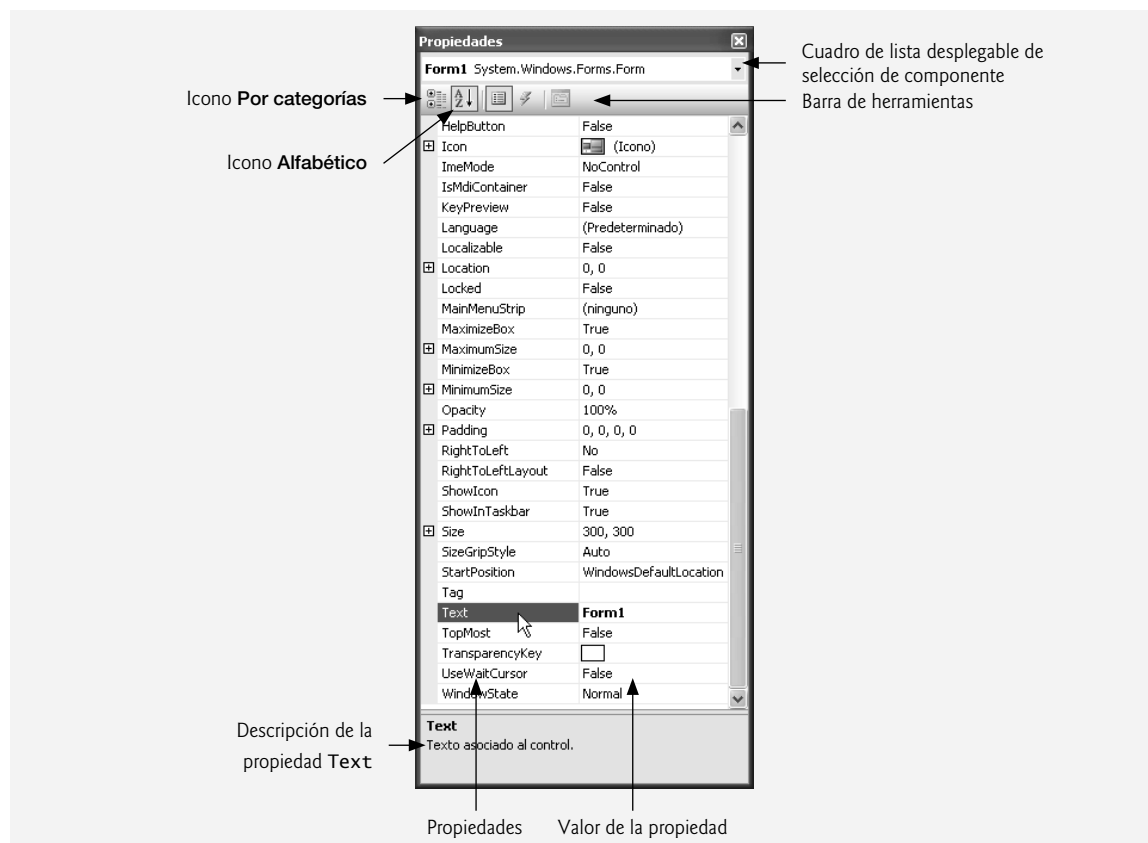


Figura 2.22 | La ventana **Propiedades** muestra la propiedad **Text** del formulario.

Visual Studio también cuenta con *ayuda sensible al contexto*, la cual es similar a la ayuda dinámica, sólo que muestra al instante un artículo de ayuda relevante, en vez de presentar una lista de artículos. Para utilizarla haga clic sobre un elemento, como el formulario, y oprima **F1**. La figura 2.25 muestra los artículos relacionados con un formulario.

Las opciones de **Ayuda** pueden establecerse en el cuadro de diálogo **Opciones** (para acceder a este cuadro de diálogo, seleccione **Herramientas > Opciones...**). Para mostrar todas las configuraciones que puede modificar (incluyendo la configuración de las opciones de **Ayuda**), asegúrese de que esté seleccionada la casilla de verificación

Comando	Descripción
Cómo	Contiene vínculos a temas relevantes, incluyendo cómo actualizar programas y aprender más acerca de los servicios Web, la arquitectura y el diseño, los archivos y la E/S, datos, depuración y muchos más.
Buscar	Busca artículos de ayuda con base en palabras clave de búsqueda.
Índice	Muestra una lista en orden alfabético de los temas que usted puede explorar.
Contenido	Muestra una tabla por categorías del contenido en el que los artículos de ayuda se organizan por tema.

Figura 2.23 | Comandos del menú **Ayuda**.

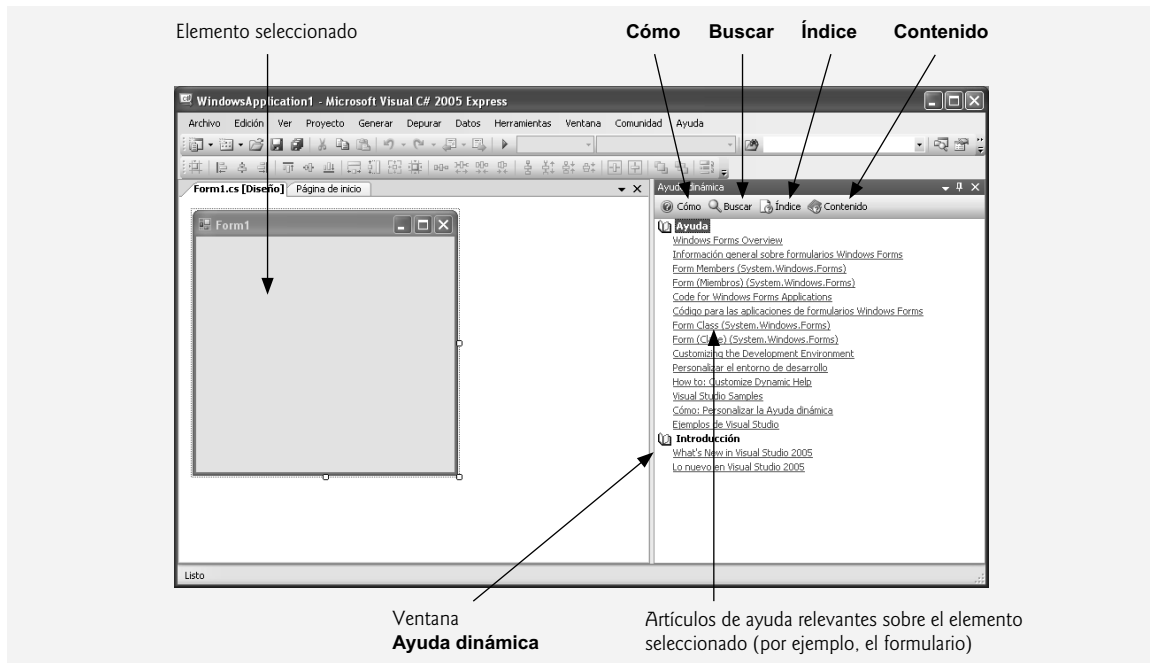


Figura 2.24 | La ventana **Ayuda dinámica**.

Mostrar todas las configuraciones en la esquina inferior izquierda del cuadro de diálogo (figura 2.26). Para cambiar la opción de mostrar la **Ayuda** en el interior o en el exterior, seleccione **Ayuda** a la izquierda y después ubique el cuadro de lista desplegable **Mostrar la Ayuda usando:** que se encuentra a la derecha. Dependiendo de su preferencia, si selecciona **Visor de ayuda externa** se mostrará un artículo de ayuda relevante en una ventana separada fuera del IDE (algunos programadores prefieren ver las páginas Web por separado del proyecto en el que se encuentran trabajando). Si selecciona **Visor de ayuda integrada** se mostrará un artículo de ayuda en forma de ventana tipo ficha dentro del IDE.

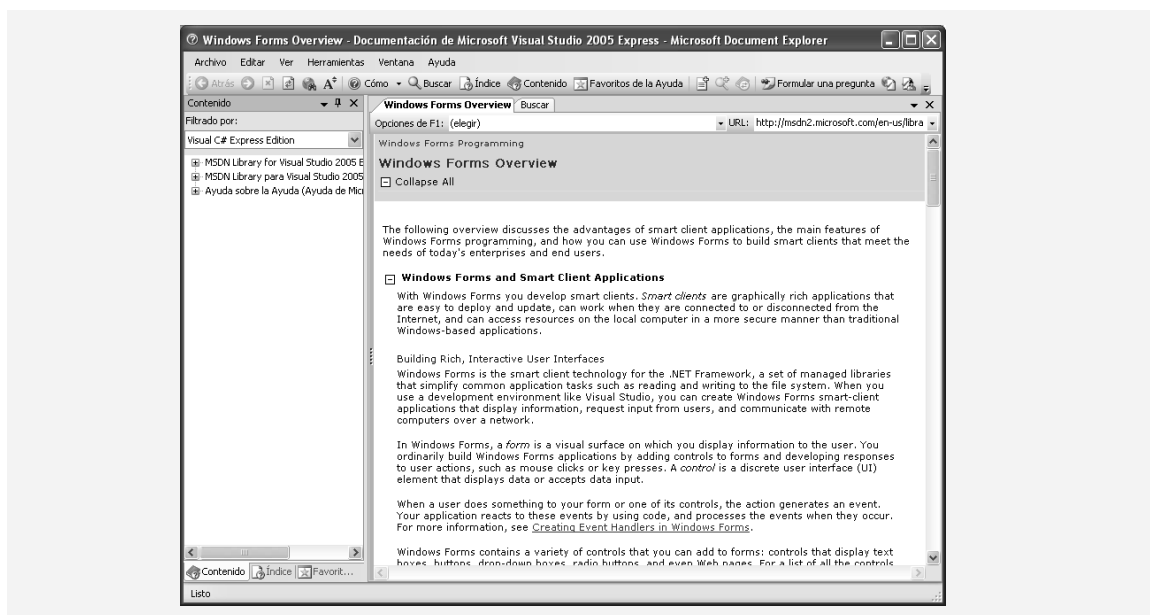


Figura 2.25 | Uso de la ayuda sensible al contexto.

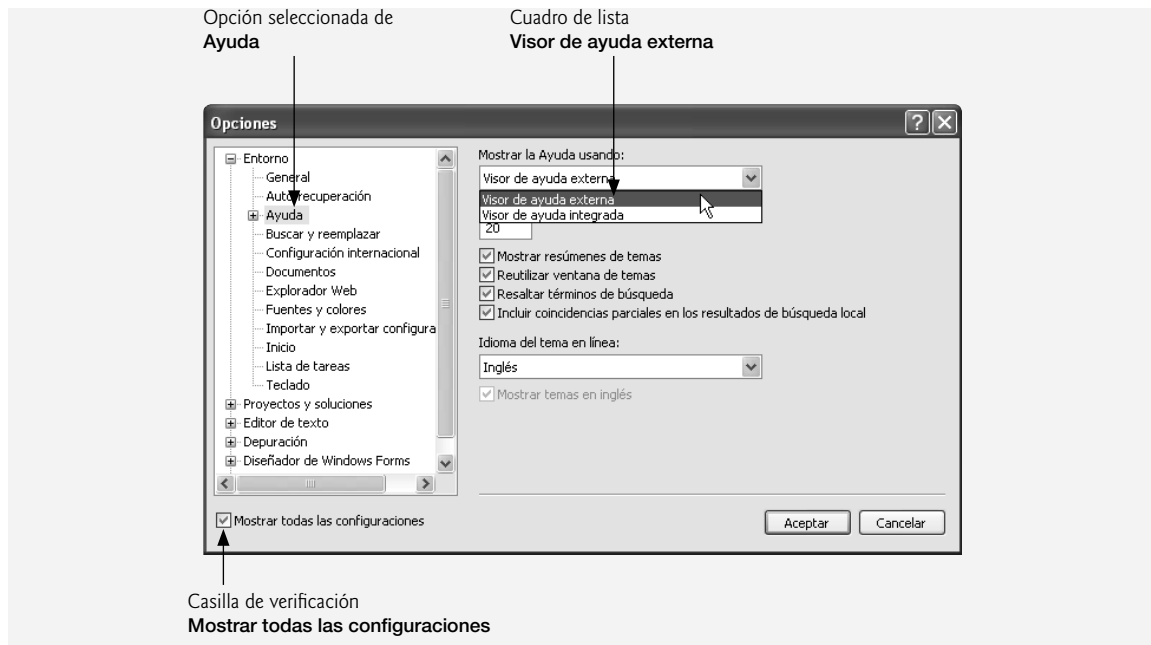


Figura 2.26 | El cuadro de diálogo **Opciones** muestra las configuraciones de **Ayuda**.

2.6 Uso de la programación visual para crear un programa simple que muestra texto e imagen

En esta sección crearemos un programa que muestra el texto “¡Bienvenido a Visual C#!” y una imagen del insecto mascota de Deitel & Associates. El programa consiste en un solo formulario que utiliza un control `Label` y un control `PictureBox`. La figura 2.27 muestra el resultado de la ejecución del programa. Este programa y la imagen del insecto están disponibles junto con los ejemplos de este capítulo. Puede descargar los ejemplos de la página Web www.deitel.com/books/csharpforprogrammers2/index.html.

Para crear el programa cuya salida se muestra en la figura 2.27, no escribiré ni una sola línea de código. En vez de ello utilizaré las técnicas de la programación visual. Visual Studio procesa sus acciones (como hacer clic con el ratón, arrastrar y soltar) para generar código de programa. En el capítulo 3 comenzaremos nuestra discusión acerca de cómo escribir código de programa. A lo largo del libro producirán programas cada vez más robustos que a menudo incluirán una combinación de código escrito por usted y código generado por Visual Studio. El código generado puede ser difícil de entender para los programadores principiantes; por fortuna, muy pocas veces tendrá que ver este código.

La programación visual es útil para crear programas con uso intensivo de la GUI, que por ende requieren mucha interacción con el usuario. La programación visual no puede utilizarse para crear programas que no tengan GUIs; debe escribir código de manera directa.

Para crear, ejecutar y terminar éste, su primer programa, realice los siguientes 13 pasos:

1. *Cree el nuevo proyecto.* Si ya hay un proyecto abierto, ciérrelo seleccionando **Archivo > Cerrar solución**. Tal vez aparezca un cuadro de diálogo que le pregunte si desea guardar el proyecto actual. Haga clic en **Guardar** para guardar cualquier cambio. Para crear una nueva aplicación Windows para nuestro programa, seleccione **Archivo > Nuevo proyecto...** para desplegar el cuadro de diálogo **Nuevo proyecto** (figura 2.28). De las opciones de plantillas, seleccione **Aplicación para Windows**. Utilice el nombre **UnProgramaSimple** para este proyecto y haga clic en **Aceptar**. [Nota: los nombres de archivo deben ajustarse a ciertas reglas. Por ejemplo, no pueden contener símbolos (como: ;, :, *, <, >, # y %) o caracteres de control Unicode® (Unicode es un conjunto especial de caracteres que se describe en el apéndice E). Además, los nombres de archivos no pueden ser nombres reservados para el sistema, como “CON”, “PRN”, “AUX” y “COM1” o “. ” y “. ”, y no pueden tener una longitud mayor a 256 caracteres.] Anteriormente en este capítulo mencionamos que debe establecer el directorio en el que se guardará este proyecto. En el Visual Studio

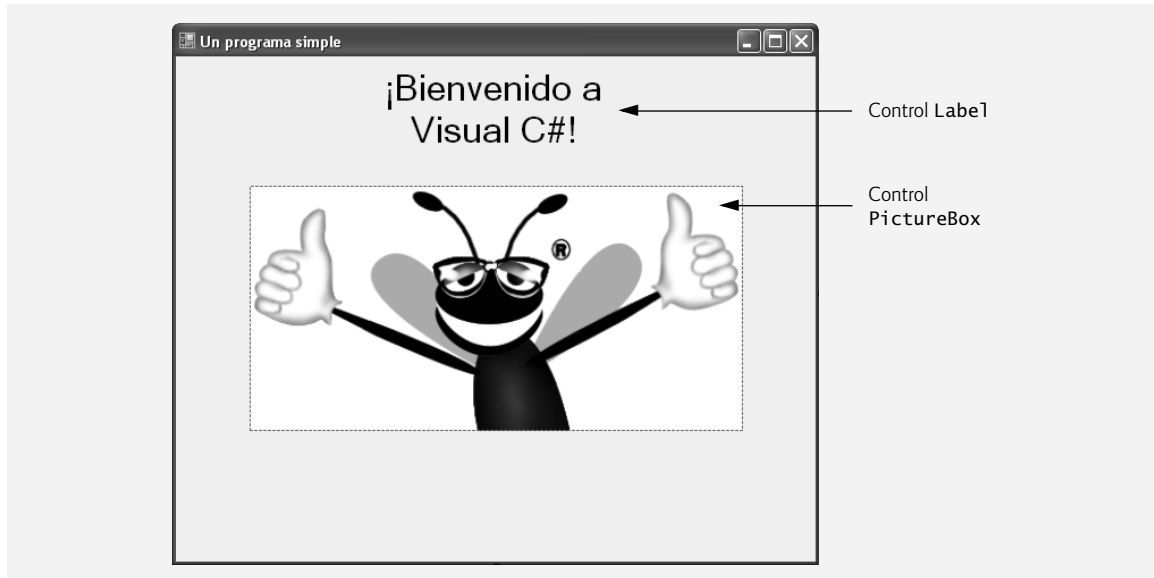


Figura 2.27 | Ejecución de un programa simple.

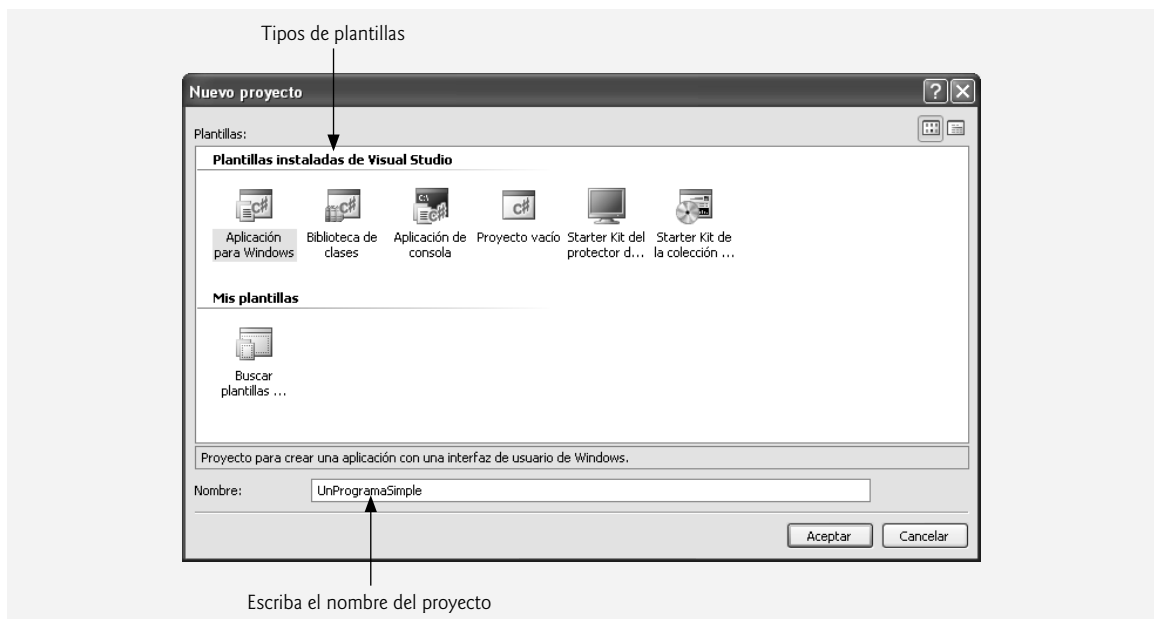


Figura 2.28 | Cuadro de diálogo Nuevo proyecto.

completo, puede hacer esto en el cuadro de diálogo **Nuevo proyecto**. Para especificar el directorio en Visual C# Express, seleccione **Archivo > Guardar todo** para que aparezca el **cuadro de diálogo Guardar proyecto** (figura 2.29). Para establecer la ubicación del proyecto haga clic en el botón **Examinar...**, el cual abrirá el **cuadro de diálogo Ubicación del proyecto** (figura 2.30). Navegue a través de los directorios, seleccione uno en el que se colocará el proyecto (en nuestro ejemplo, utilizamos un directorio llamado **MisProyectos**) y haga clic en **Aceptar** para cerrar el cuadro de diálogo. Asegúrese de que esté seleccionada la casilla de verificación **Crear directorio para la solución** (figura 2.29). Haga clic en **Guardar** para cerrar el cuadro de diálogo **Guardar proyecto**.

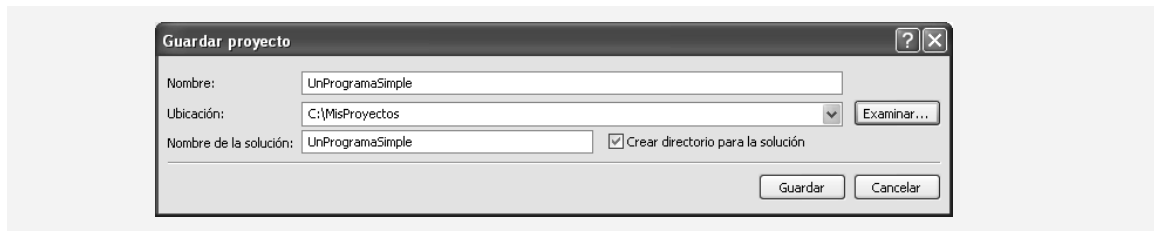


Figura 2.29 | Cuadro de diálogo **Guardar proyecto**.

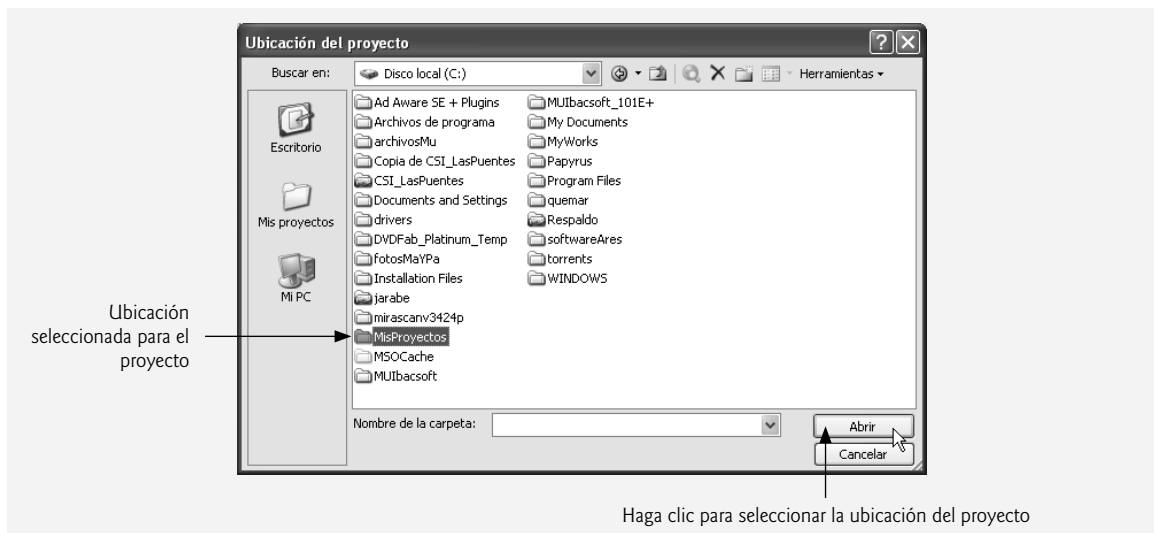


Figura 2.30 | Establezca la ubicación del proyecto en el cuadro de diálogo **Ubicación del proyecto**.

La primera vez que comience a trabajar en el IDE, estará en *modo de diseño* (es decir, el programa se está diseñando y no se encuentra en ejecución). Mientras que el IDE se encuentre en modo de diseño, usted tendrá acceso a todas las ventanas del entorno (por ejemplo, **Cuadro de herramientas**, **Propiedades**), menús y barras de herramientas, como verá en breve.

2. *Establezca el texto en la barra de título del formulario.* El texto en la **barra de título** del formulario es el valor de la **propiedad Text** (figura 2.31) del formulario. Si la ventana **Propiedades** no está abierta, haga clic en el icono de propiedades en la barra de herramientas o seleccione **Ver > Ventana Propiedades**. Haga clic en cualquier parte del formulario para mostrar las propiedades del mismo en la ventana **Propiedades**. Haga clic en el cuadro de texto a la derecha del cuadro de la propiedad **Text** y escriba "Un programa simple", como en la figura 2.31. Oprima **Intro** cuando termine; la barra de título del formulario se actualizará de inmediato (figura 2.32).
3. *Cambie el tamaño del formulario.* Haga clic y arrastre uno de los **manejadores de tamaño habilitados** (los pequeños cuadros blancos que aparecen alrededor del formulario, como se muestra en la figura 2.32). Utilice el ratón para seleccionar y arrastrar el manejador de tamaño para modificar el tamaño del formulario (figura 2.33).
4. *Cambie el color de fondo del formulario.* La **propiedad BackColor** especifica el color de fondo de un formulario o control. Al hacer clic en **BackColor** en la ventana **Propiedades** aparece un botón con una flecha hacia abajo enseguida del valor de la propiedad (figura 2.34). Al hacer clic en este botón, se despliega un conjunto de otras opciones, que varían dependiendo de la propiedad. En este caso, dicho botón muestra las

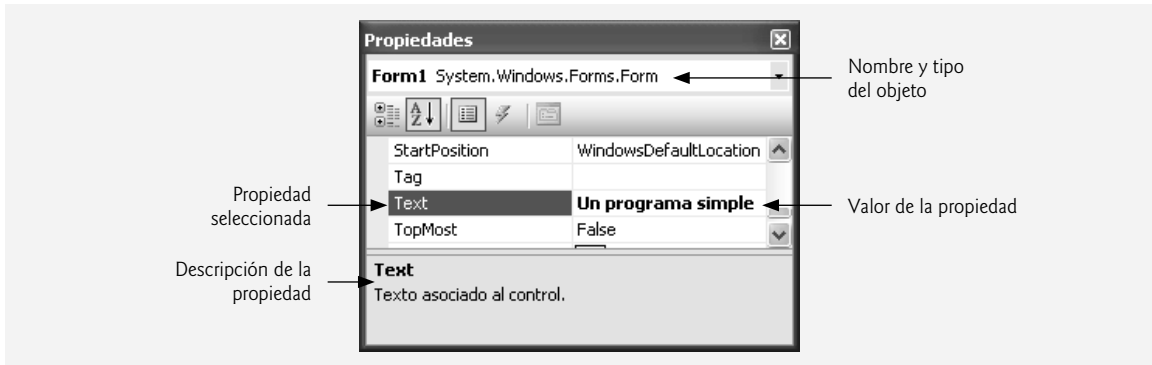


Figura 2.31 | Establezca la propiedad Text del formulario en la ventana **Propiedades**.

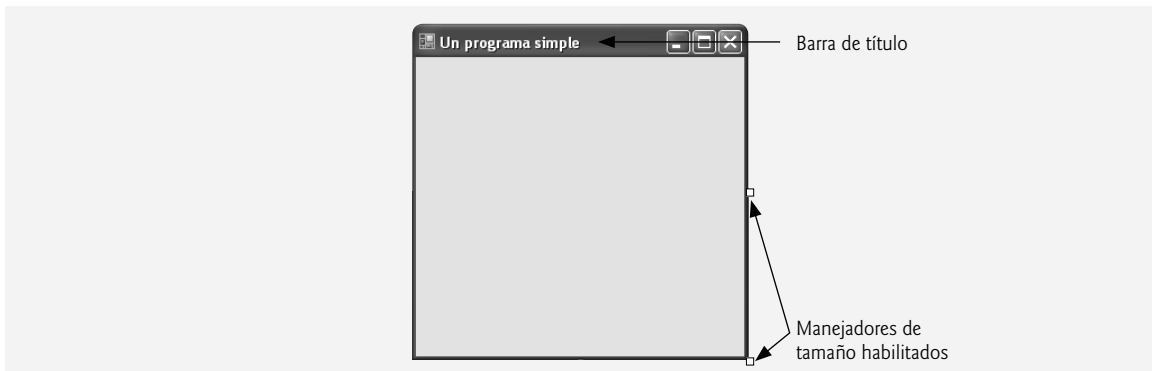


Figura 2.32 | Forma con manejadores de tamaño habilitados.

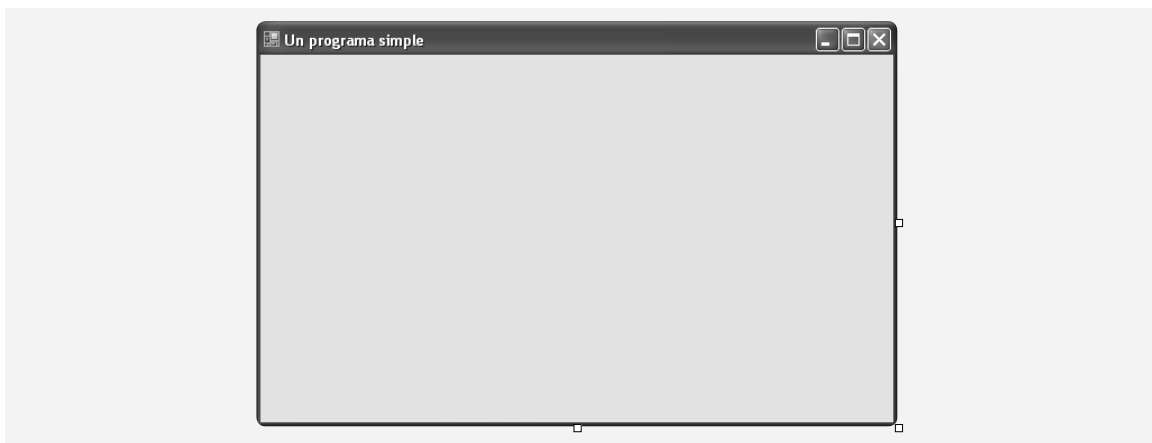


Figura 2.33 | Formulario con nuevo tamaño.

fichas **Personalizado**, **Web** y **Sistema** (la predeterminada). Haga clic en la *ficha Personalizado* para mostrar la *paleta* (una rejilla de colores). Seleccione el cuadro que representa el color azul claro. Una vez que seleccione el color, la paleta se cerrará y el color de fondo del formulario cambiará a azul claro (figura 2.35).

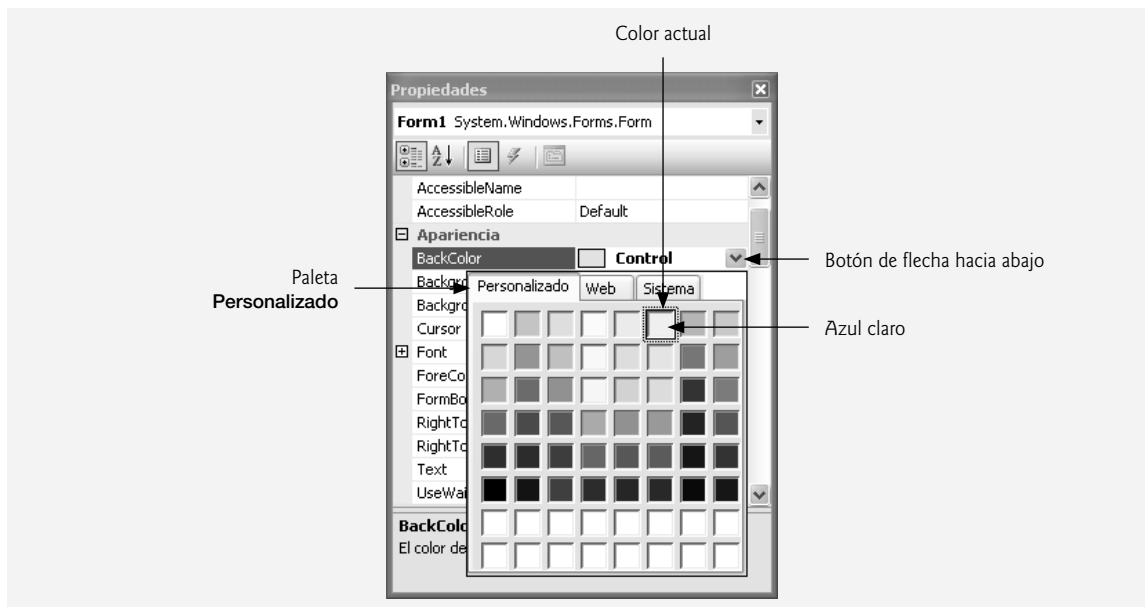


Figura 2.34 | Cambie la propiedad BackColor del formulario.

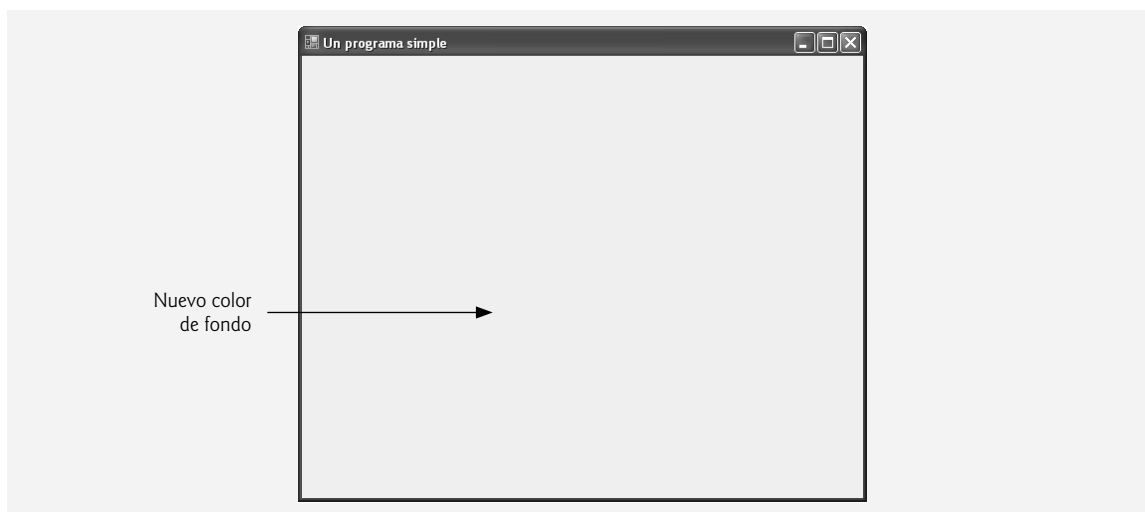


Figura 2.35 | Se aplicó la nueva propiedad BackColor en el formulario.

5. *Agregue un control Label al formulario.* Si el **Cuadro de herramientas** ya no se encuentra abierto, seleccione **Ver > Cuadro de herramientas** para mostrar el conjunto de controles que utilizará para crear sus programas. Para el tipo de programa que crearemos en este capítulo, los controles típicos que utilizaremos se encuentran en la categoría **Todos los formularios Windows Forms** del **Cuadro de herramientas** o del grupo **Controles comunes**. Si se colapsa cualquiera de los dos nombres de grupo, haga clic en el signo más para expandirlo (los grupos **Todos los formularios Windows Forms** y **Controles comunes** se muestran cerca de la parte superior de la figura 2.21). A continuación, haga doble clic en el control **Label** del **Cuadro de herramientas**. Esta acción hará que aparezca un control **Label** en la esquina superior izquierda del formulario (figura 2.36). [Nota: si el formulario está detrás del **Cuadro de herramientas**, tal vez necesite cerrar el **Cuadro de herramientas** para ver el control **Label**.] Aunque si hace doble clic sobre cualquier control del **Cuadro de herramientas** se coloca ese control en el formulario, también puede “arrastrar” los controles del **Cuadro de herramientas** hacia

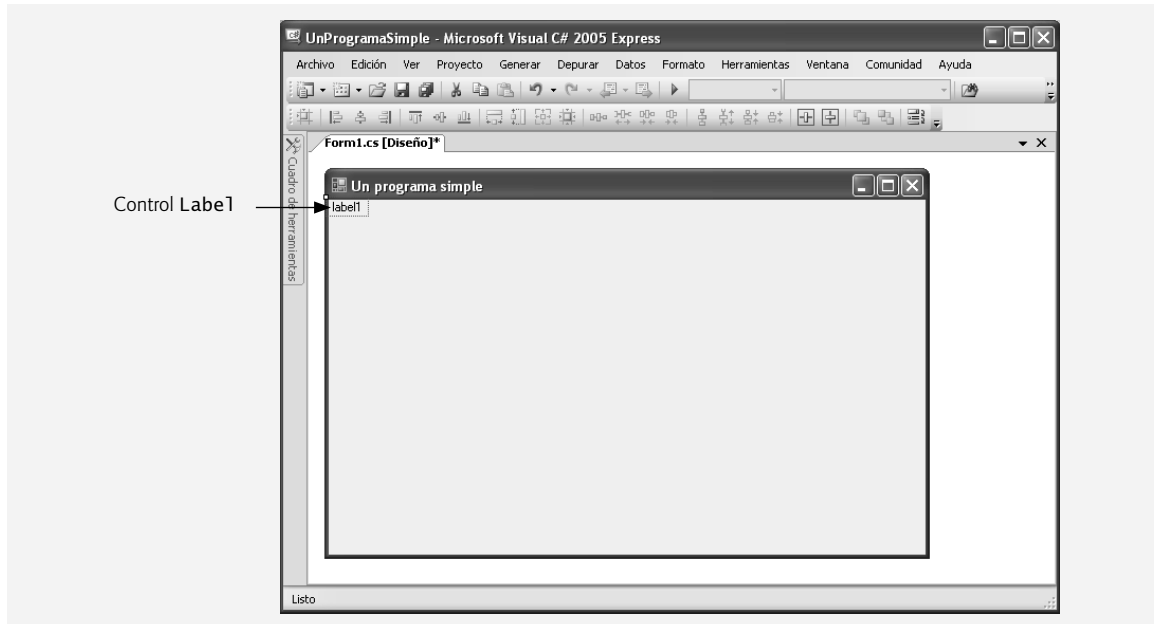


Figura 2.36 | Agregue un control `Label` al formulario.

el formulario (tal vez prefiera arrastrar el control, ya que puede colocarlo en la posición que desee). Nuestro control `Label` muestra el texto `label1` de manera predeterminada. Observe que el color de fondo de nuestro control `Label` es el mismo que el color de fondo del formulario. Cuando se agrega un control al formulario, su propiedad `BackColor` se hace igual a la propiedad `BackColor` del formulario. Para cambiar el color de fondo del control `Label` a un color distinto del formulario, cambie su propiedad `BackColor`.

6. *Personalice la apariencia del control `Label`.* Seleccione el control `Label` haciendo clic sobre él. Ahora deberán aparecer sus propiedades en la ventana **Propiedades** (figura 2.37). La propiedad `Text` del control `Label` determina el texto (si es que lo hay) que muestra la etiqueta. Tanto el formulario como el control `Label` tienen su propiedad `Text`; los formularios y los controles pueden tener los mismos tipos de propiedades (como `BackColor` y `Text`) sin conflictos. Escriba el texto **¡Bienvenido a Visual C#!** en la propiedad `Text` del control `Label`. Observe que el control `Label` cambia de tamaño para alojar todo el texto que se escribe en una línea. De manera predeterminada, la *propiedad `AutoSize`* del control `Label` se establece en `True`, lo cual permite al control `Label` ajustar su tamaño para acomodar a todo el texto, en caso de ser necesario. Establezca la propiedad `AutoSize` en `False` (figura 2.37) para que pueda cambiar el tamaño del control `Label` por su cuenta. Cambie el tamaño del control `Label` (mediante el uso de los manejadores de tamaño) de manera que pueda verse todo el texto. Desplace el control `Label` hacia la parte superior central del formulario; para ello arrástrelo o utilice las teclas de flecha izquierda y derecha para ajustar su posición (figura 2.38). De manera alternativa, cuando esté seleccionado el control `Label` podrá centrarlo horizontalmente si selecciona **Formato > Centrar en el formulario > Horizontalmente**.
7. *Establezca el tamaño de letra del control `Label`.* Para cambiar el tipo de letra y la apariencia del texto del control `Label`, seleccione el valor de la *propiedad `Font`*, con lo cual aparecerá un *botón de elipsis* (...) enseguida del valor (figura 2.39). Al hacer clic en el botón de elipsis aparece un cuadro de diálogo que contiene valores adicionales (en este caso, el *cuadro de diálogo `Fuente`* (figura 2.40)). En este cuadro de diálogo puede seleccionar el nombre de la fuente (por ejemplo, **Microsoft Sans Serif**, **MingLiU**, **Mistral**, **Modern No. 20**; las opciones pueden variar, dependiendo de su sistema), el estilo (**Normal**, **Cursiva**, **Negrita**, etc.) y el tamaño (**16**, **18**, **20**, etc.). El área **Ejemplo** muestra texto de ejemplo con las opciones de fuente seleccionadas. En **Tamaño**, seleccione **24** puntos y haga clic en **Aceptar**. Si el texto del control `Label` no cabe en una sola línea, se ajusta a la siguiente línea. Cambie el tamaño vertical del control `Label` si no es lo suficientemente grande como para que aparezca todo el texto.



Figura 2.37 | Establezca la propiedad `AutoSize` del control `Label` en `False`.

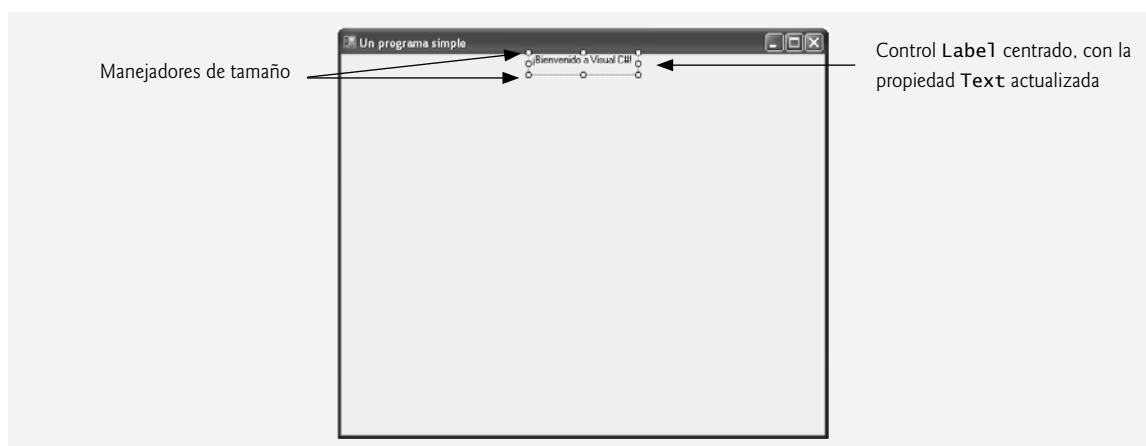


Figura 2.38 | La GUI, una vez que se personalizan el formulario y el control `Label`.



Figura 2.39 | La ventana **Propiedades** muestra las propiedades del control `Label`.

8. *Alinee el texto del control `Label`.* Seleccione la propiedad **`TextAlign`** del control `Label`, la cual determina la forma en que se alinea el texto dentro del control `Label`. A continuación aparecerá una rejilla de tres por tres botones, la cual representa las opciones de alineación (figura 2.41). La posición de cada botón corresponde al lugar en el que aparecerá el texto en el control `Label`. Para este programa, seleccione la opción `MiddleCenter` para la propiedad `TextAlign` en la rejilla de tres por tres; esta selección hará que

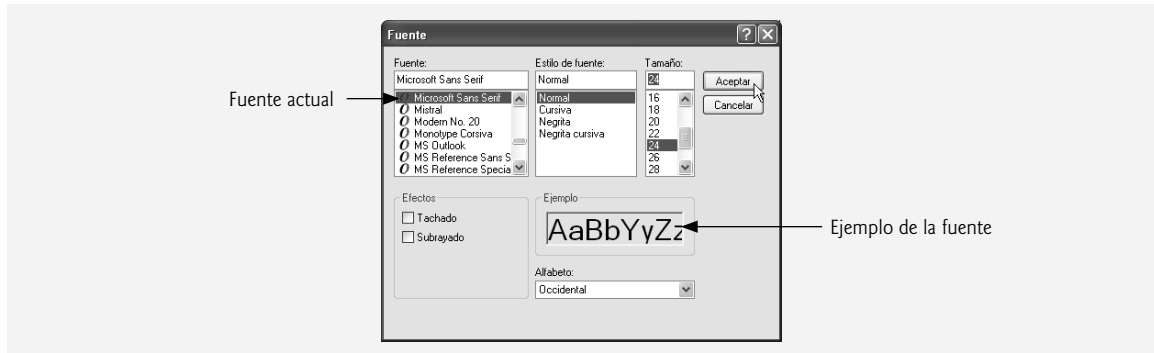


Figura 2.40 | Cuadro de diálogo **Fuente** para seleccionar fuentes, tamaños y estilos.

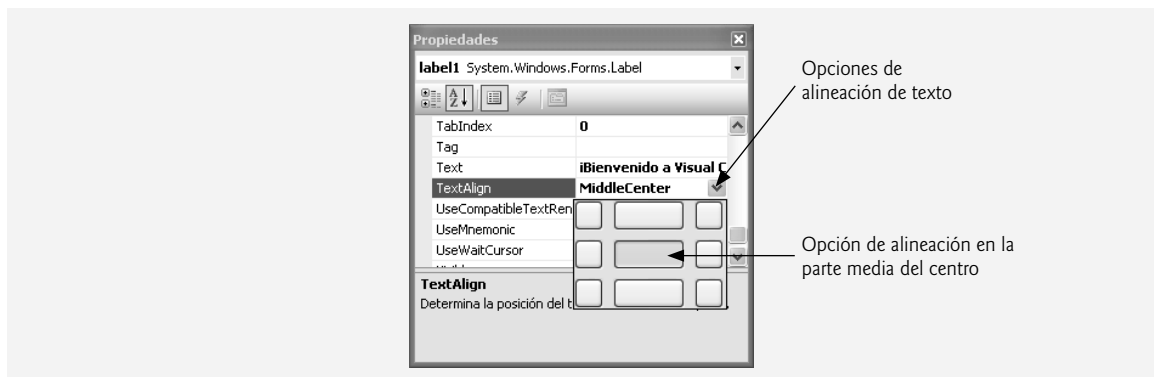


Figura 2.41 | Centre el texto del control `Label1`.

el texto aparezca centrado en la parte media del control `Label1`, con un espaciamiento igual desde el texto hacia todos los lados del control `Label1`. Los demás valores de `TextAlign`, como `TopLeft`, `TopRight` y `BottomCenter`, se pueden utilizar para posicionar el texto en cualquier parte dentro de un control `Label1`. Ciertos valores de alineación pueden requerir que usted cambie el tamaño del control `Label1` para hacerlo más grande o más pequeño, de manera que se ajuste mejor el texto.

9. *Agregue un control `PictureBox` al formulario.* El control `PictureBox` muestra imágenes. El proceso involucrado en este paso es similar al del *paso 5*, en donde agregamos un control `Label1` al formulario. Localice el control `PictureBox` en el menú del **Cuadro de herramientas** (figura 2.21) y haga doble clic para agregar el control `PictureBox` al formulario. Cuando aparezca este control, desplácelo por debajo del control `Label1`, ya sea arrastrándolo o mediante las teclas de flecha (figura 2.42).
10. *Inserte una imagen.* Haga clic en el control `PictureBox` para mostrar sus propiedades en la ventana **Propiedades** (figura 2.43). Localice la **propiedad `Image`**, la cual muestra una vista previa de la imagen, en caso de que exista. Como no se ha asignado ninguna imagen, el valor de la propiedad `Image` muestra (**ninguno**). Haga clic en el botón de elipsis para que aparezca el **cuadro de diálogo `Seleccionar recurso`** (figura 2.44). Este cuadro de diálogo se utiliza para importar archivos, como imágenes, en cualquier programa. Haga clic en el botón **Importar...** para buscar una imagen e insertarla. En nuestro caso, la imagen es `insecto.png`. En el cuadro de diálogo que aparezca (figura 2.45), haga clic en la imagen con el ratón y después haga clic en **Aceptar**. Aparecerá una vista previa de la imagen en el cuadro de diálogo **Seleccionar recurso** (figura 2.45). Haga clic en **Aceptar** para colocar la imagen en su programa. Los formatos de imagen soportados son PNG (Gráficos portables de red), GIF (Formato de intercambio de gráficos), JPEG (Grupo de expertos en fotografía unidos) y BMP (Mapa de bits de Windows). Para crear una nueva imagen se requiere software de edición de imágenes, como Jasc® Paint Shop Pro™

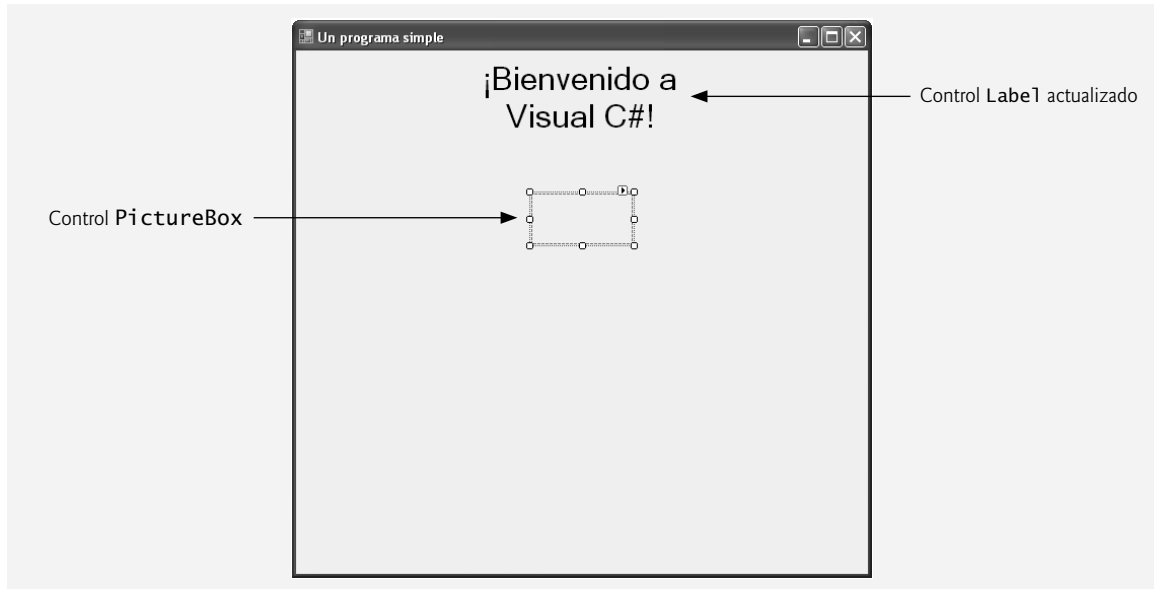


Figura 2.42 | Inserte y alinee un control PictureBox.

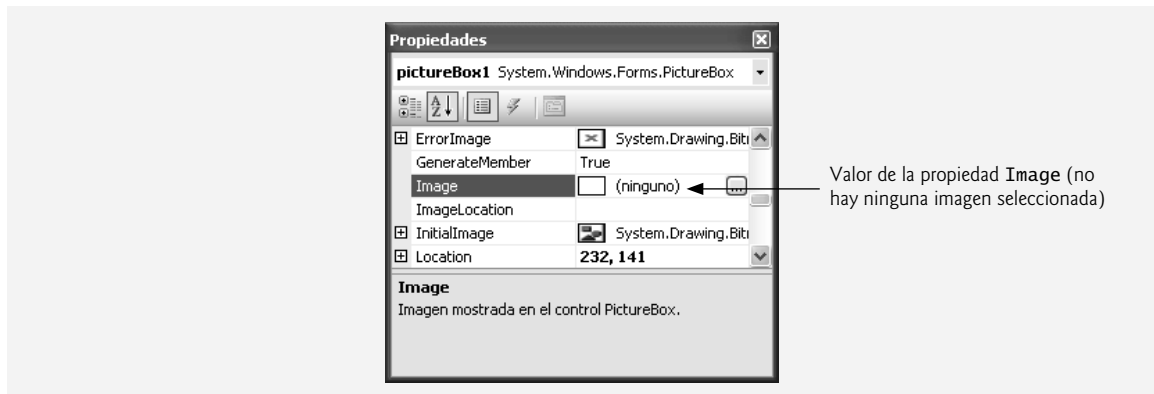


Figura 2.43 | Propiedad Image del control PictureBox.



Figura 2.44 | En el cuadro de diálogo **Seleccionar recurso** puede seleccionar una imagen para el control PictureBox.



Figura 2.45 | El cuadro de diálogo **Seleccionar recurso** muestra una vista previa de una imagen seleccionada.

(www.jasc.com), Adobe® Photoshop™ Elements (www.adobe.com) o Microsoft Paint (que se incluye con Windows). Para ajustar el tamaño de la imagen al control PictureBox, modifique la *propiedad SizeMode* a **StretchImage** (figura 2.46), con lo cual la imagen se escalará al tamaño del control PictureBox. Cambie el tamaño del control PictureBox para hacerlo más grande (figura 2.47).

11. *Guarde el proyecto.* Seleccione **Archivo > Guardar todo** para guardar la solución completa. El archivo de la solución contiene el nombre y la ubicación de su proyecto, y el archivo de proyecto contiene los nombres y las ubicaciones de todos los archivos en el proyecto.
12. *Ejecute el proyecto.* Recuerde que hasta este punto hemos estado trabajando en el modo de diseño del IDE (es decir, el programa que estamos creando no se está ejecutando). En el *modo de ejecución* el programa se está ejecutando y usted puede interactuar con algunas cuantas características del IDE; las características no disponibles están deshabilitadas (en color gris). El texto **Form1.cs [Diseño]** en la ficha (figura 2.48) significa que estamos diseñando el formulario de manera visual, en vez de hacerlo por medio de la programación. Si hubiéramos escrito código, la ficha sólo contendría el texto **Form1.cs**. Para ejecutar el programa, primero debe generar la solución. Seleccione **Generar > Generar solución** para compilar el proyecto (figura 2.48). Una vez que genere la solución (el IDE mostrará la leyenda “Generación satisfactoria” en la esquina inferior izquierda del IDE, a la cual se le conoce también como barra de estado), seleccione **Depurar > Iniciar depuración** para ejecutar el programa. La figura 2.49 muestra el IDE en modo de ejecución (lo cual se indica por el texto de la barra de título **UnProgramaSimple (Ejecutando) – Microsoft Visual C# 2005 Express Edition**). Observe que muchos iconos y menús de la barra de herramientas están deshabilitados. El programa en ejecución aparecerá en una ventana separada, fuera del IDE (figura 2.49).

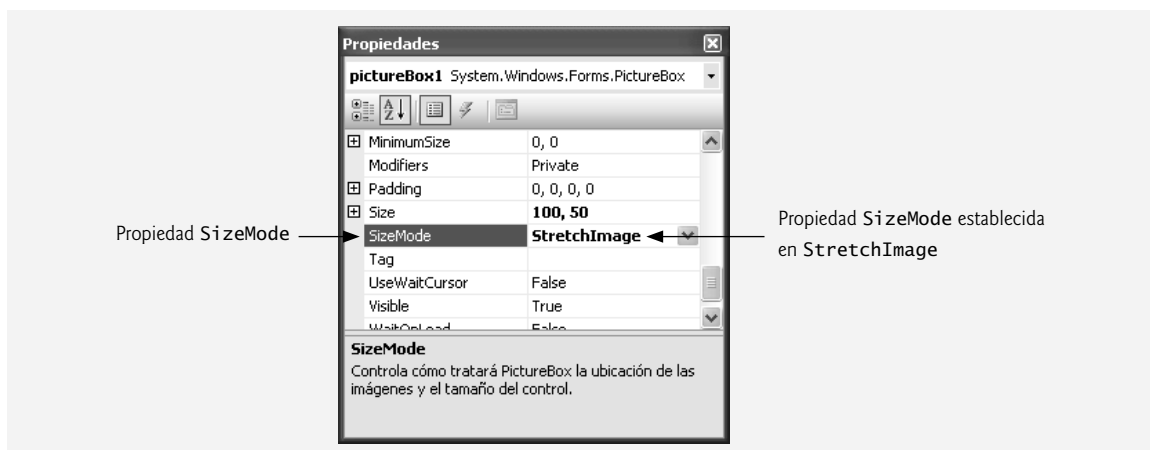


Figura 2.46 | Escale una imagen para ajustarla al tamaño del control PictureBox.

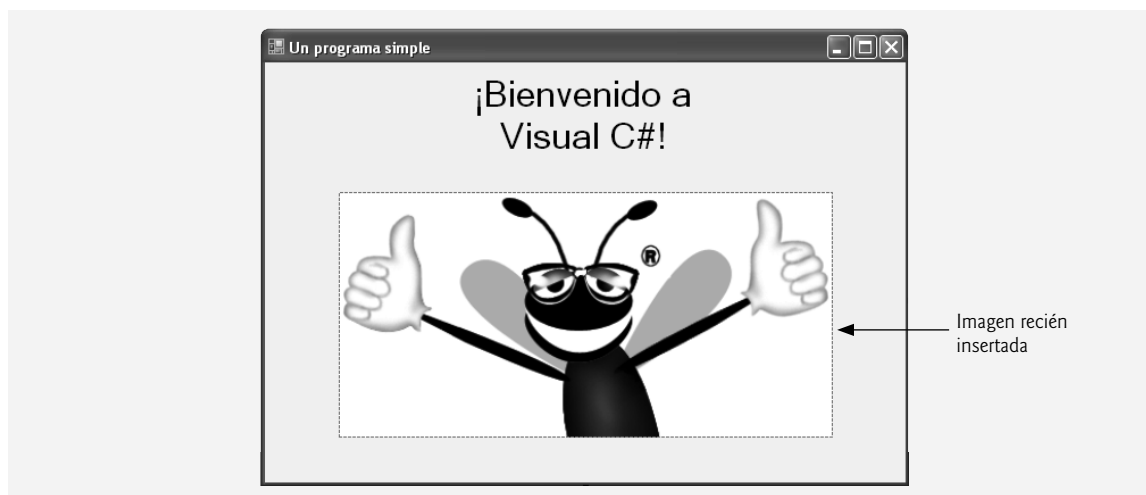


Figura 2.47 | El control PictureBox muestra una imagen.

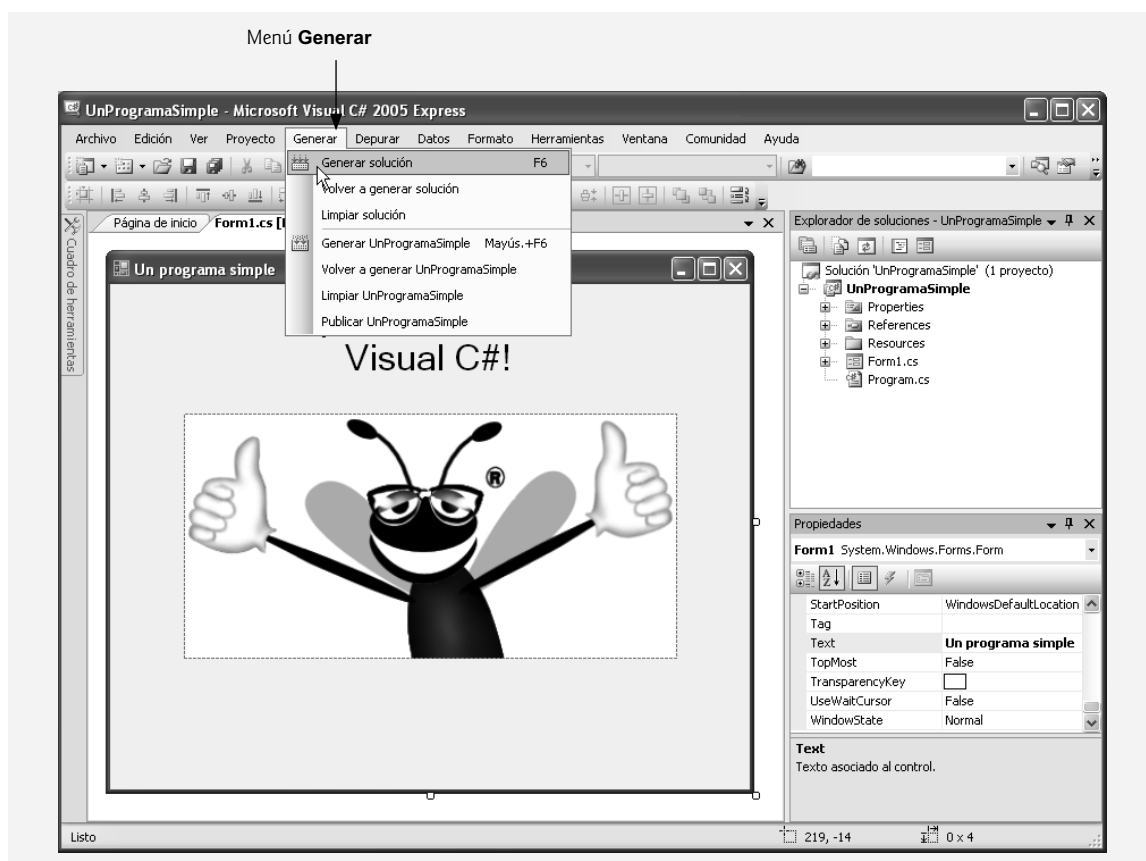


Figura 2.48 | Genere una solución.

13. *Termine la ejecución.* Para terminar el programa, haga clic en el cuadro Cerrar del programa en ejecución (la X en la esquina superior derecha de la ventana del programa en ejecución). Esta acción detendrá la ejecución del programa y regresará el IDE al modo de diseño.

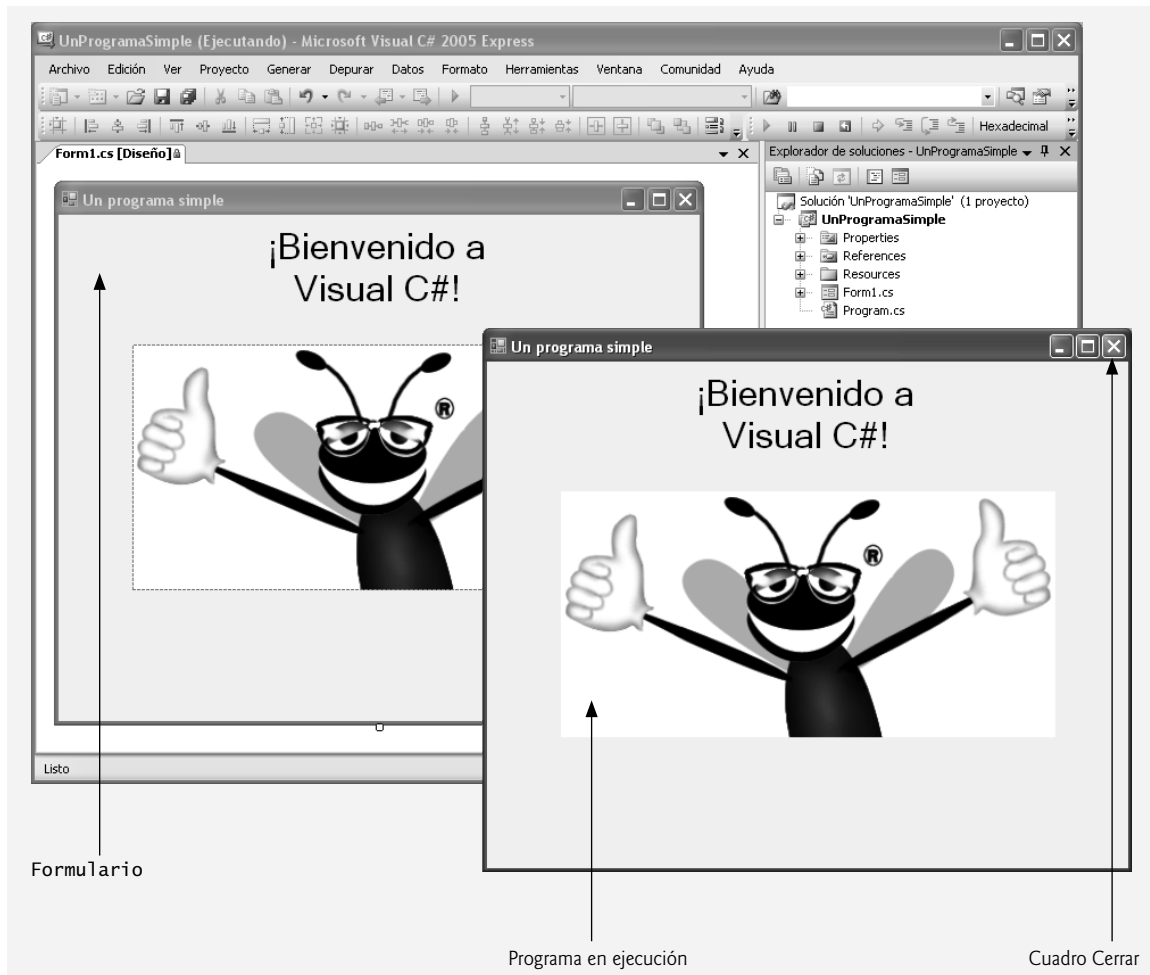


Figura 2.49 | El IDE en modo de ejecución, con el programa en ejecución en la ventana principal.

2.7 Conclusión

En este capítulo le presentamos las características clave del Entorno Integrado de Desarrollo (IDE) de Visual Studio. Utilizó la técnica de la programación visual para crear un programa funcional en Visual C#, sin necesidad de escribir una sola línea de código. La programación en Visual C# es una mezcla de los dos estilos: la programación visual le permite desarrollar GUIs con facilidad y evitar la tediosa programación de GUIs; la programación convencional (que presentaremos en el capítulo 3) le permite especificar el comportamiento de sus programas.

En este capítulo creó una aplicación Windows en Visual C# con un formulario. Trabajó con las ventanas **Explorador de soluciones**, **Cuadro de herramientas** y **Propiedades**, las cuales son esenciales para desarrollar programas en Visual C#. La ventana **Explorador de soluciones** le permite administrar los archivos de su solución en forma visual. La ventana **Cuadro de herramientas** contiene una robusta colección de controles para crear GUIs. La ventana **Propiedades** le permite establecer los atributos de un formulario y de los controles.

También exploró las características de ayuda de Visual Studio, incluyendo la ventana **Ayuda dinámica** y el menú **Ayuda**. La ventana **Ayuda dinámica** muestra vínculos relacionados con el elemento sobre el que usted haga clic con el ratón. Aprendió a establecer las opciones de **Ayuda** para mostrar y utilizar recursos de ayuda. También demostramos cómo utilizar la ayuda sensible al contexto.

Utilizó la programación visual para diseñar con rapidez y facilidad las porciones de un programa relacionadas con la GUI, mediante la acción de arrastrar y soltar controles (un `Label` y un `PictureBox`) en un formulario (Form) o haciendo doble clic en los controles en el **Cuadro de herramientas**.

Al crear el programa **UnProgramaSimple** utilizó la ventana **Propiedades** para establecer las propiedades **Text** y **BackColor** del formulario. Aprendió que los controles **Label** muestran texto y que los controles **PictureBox** muestran imágenes. Mostró texto en un control **Label** y agregó una imagen a un control **PictureBox**. También trabajó con las propiedades **AutoSize**, **TextAlign** y **SizeMode** de un control **Label**.

En el siguiente capítulo hablaremos sobre la programación “no visual” o “convencional”: usted creará sus primeros programas que contendrán código en Visual C# escrito por usted, en vez de hacer que Visual Studio escriba el código. Estudiará las aplicaciones de consola (programas que muestran texto en pantalla sin utilizar una GUI). También aprenderá acerca de los conceptos de memoria, aritmética, toma de decisiones y cómo utilizar un cuadro de diálogo para mostrar un mensaje.

2.8 Recursos Web

En inglés

msdn.microsoft.com/vs2005

El sitio de Microsoft Visual Studio ofrece noticias, documentación, descargas y otros recursos.

lab.msdn.microsoft.com/vs2005/

Este sitio ofrece información acerca de la versión más reciente de Visual Studio, incluyendo descargas, información de la comunidad y recursos.

www.worldofdotnet.net

Este sitio ofrece noticias sobre Visual Studio y vínculos a grupos de noticias y otros recursos.

www.c-sharpcorner.com

Este sitio contiene artículos, reseñas de libros y software, documentación, descargas, vínculos e información acerca de C#.

www.devx.com/dotnet

Este sitio tiene una zona dedicada a desarrolladores .NET, que contiene artículos, opiniones, grupos de noticias, código, tips y demás recursos acerca de Visual Studio 2005.

www.csharp-station.com

Este sitio proporciona artículos acerca de Visual C#, en especial las actualizaciones del 2005. También incluye tutoriales, descargas y otros recursos.

En español

www.microsoft.com/spanish/msdn/vs2005/default.msp

El sitio de Microsoft Visual Studio en español ofrece noticias, documentación, descargas y otros recursos.

www.microsoft.com/spanish/msdn/vstudio/express/VCS/default.msp

En este sitio encontrará la versión Visual C# 2005 Express Edition para descargar, junto con noticias y tutoriales.

msdn2.microsoft.com/es-es/library/kx37x362.aspx

Este sitio contiene la documentación de Microsoft Visual Studio 2005, además de ejemplos sobre el uso de Visual C#.

3

Introducción a las aplicaciones de C#

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Cómo escribir aplicaciones simples en C# usando código, en lugar de la programación visual.
- Escribir instrucciones que envíen y reciban datos hacia/desde la pantalla.
- Cómo declarar y utilizar datos de diversos tipos.
- Almacenar y recuperar datos de la memoria.
- Utilizar los operadores aritméticos.
- Determinar el orden en que se aplican los operadores.
- Escribir instrucciones de toma de decisiones.
- Utilizar los operadores relacionales y de igualdad.
- Utilizar cuadros de diálogo de mensaje para mostrar mensajes.

¿Qué hay en un nombre? Lo que llamamos rosa aun con otro nombre mantendría su perfume.

—William Shakespeare

Cuando me topo con una decisión, siempre pregunto, “¿Qué sería lo más divertido?”

—Peggy Walker

“Toma un poco más de té”, dijo el conejo blanco a Alicia, con gran seriedad. “No he tomado nada todavía.” Alicia contestó en tono ofendido, “Entonces no puedo tomar más”. “Querrás decir que no puedes tomar menos”, dijo el sombrerero loco, “es muy fácil tomar más que nada”.

—Lewis Carroll

- 3.1 Introducción
- 3.2 Una aplicación simple en C#: mostrar una línea de texto
- 3.3 Cómo crear una aplicación simple en Visual C# Express
- 3.4 Modificación de su aplicación simple en C#
- 3.5 Formato del texto con `Console.Write` y `Console.WriteLine`
- 3.6 Otra aplicación en C#: suma de enteros
- 3.7 Conceptos sobre memoria
- 3.8 Aritmética
- 3.9 Toma de decisiones: operadores de igualdad y relacionales
- 3.10 (Opcional) Caso de estudio de ingeniería de software: análisis del documento de requerimientos del ATM
- 3.11 Conclusión

3.1 Introducción

Ahora presentaremos la programación de aplicaciones en C#, que nos facilita un método disciplinado para su diseño. La mayoría de las aplicaciones de C# que estudiaremos en este libro procesa información y muestra resultados. En este capítulo introduciremos las **aplicaciones de consola**; éstas envían y reciben texto en una **ventana de consola**. En Microsoft Windows 95/98/ME, la ventana de consola es el **símbolo de MS-DOS**. En otras versiones de Microsoft Windows, la ventana de consola es el **Símbolo del sistema**.

Empezaremos con diversos ejemplos que sólo muestran mensajes en la pantalla. Después demostraremos una aplicación que obtiene dos números de un usuario, calcula la suma y muestra el resultado. Aprenderá a realizar diversos cálculos aritméticos y a guardar los resultados para utilizarlos posteriormente. Muchas aplicaciones contienen lógica que requiere que la aplicación tome decisiones; el último ejemplo de este capítulo demuestra los fundamentos de la toma de decisiones al enseñarle cómo comparar números y mostrar mensajes con base en los resultados de las comparaciones. Por ejemplo, la aplicación muestra un mensaje que indica que dos números son iguales sólo si tienen el mismo valor. Analizaremos cuidadosamente cada ejemplo, una línea a la vez.

3.2 Una aplicación simple en C#: mostrar una línea de texto

Consideremos ahora una aplicación simple que muestra una línea de texto. (Más adelante en esta sección veremos cómo compilar y ejecutar una aplicación.) La figura 3.1 muestra esta aplicación y su salida. La aplicación ilustra varias características importantes del lenguaje C#. Para su conveniencia, cada uno de los programas que presentamos en este libro incluye números de línea, que no forman parte del verdadero código de C#. En la sección 3.3 le indicaremos cómo mostrar los números de línea para su código de C# en el IDE. Pronto veremos que la línea 10 es la que hace el verdadero trabajo de la aplicación; a saber, muestra en la pantalla la frase ¡Bienvenido a la programación en C#! Ahora analizaremos cada una de las líneas de esta aplicación.

La línea 1

```
// Figura 3.1: Bienvenido1.cs
```

comienza con `//`, lo cual indica que el resto de la línea es un **comentario**. Empezamos cada aplicación con un comentario en el que se indica el número de figura y el nombre del archivo en el que se guarda la aplicación.

A un comentario que comienza con `//` se le llama **comentario de una sola línea**, ya que termina al final de la línea en la que aparece. Este tipo de comentario también puede comenzar en medio de una línea y continuar hasta el final de la misma (como en las líneas 7, 11 y 12).

Los **comentarios delimitados** como

```
/* Éste es un comentario
delimitado. Puede dividirse
en muchas líneas*/
```

```

1 // Fig. 3.1: Bienvenido1.cs
2 // Aplicación para imprimir texto.
3 using System;
4
5 public class Bienvenido1
6 {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10         Console.WriteLine( "¡Bienvenido a la programación en C#!" );
11     } // fin del método Main
12 } // fin de la clase Bienvenido1

```

```
¡Bienvenido a la programación en C#!
```

Figura 3.1 | Aplicación para imprimir texto en pantalla.

pueden esparcirse a través de varias líneas. Este tipo de comentario comienza con el delimitador `/*` y termina con el delimitador `*/`. El compilador ignora todo el texto entre los delimitadores. C# incorporó los comentarios delimitados y los de una sola línea de los lenguajes de programación C y C++, en forma respectiva. En este libro utilizaremos comentarios de una sola línea en nuestros programas.

La línea 2

```
// Aplicación para imprimir texto.
```

es un comentario de una sola línea, que describe el propósito de la aplicación.

La línea 3

```
using System;
```

es una **directiva using**, que ayuda al compilador a localizar una clase que se utiliza en esta aplicación. Uno de los puntos fuertes de C# es su robusto conjunto de clases predefinidas que usted puede utilizar, en lugar de “reinventar la rueda”. Estas clases se organizan bajo **espacios de nombres**: colecciones con nombre de clases relacionadas. A los espacios de nombres de .NET se les conoce como **Biblioteca de Clases del .NET Framework (FCL)**. Cada directiva `using` identifica las clases predefinidas que puede utilizar una aplicación de C#. La directiva `using` en la línea 3 indica que este ejemplo utiliza clases del espacio de nombres `System`, el cual contiene la clase predefinida `Console` (que veremos en breve) que se utiliza en la línea 10, y muchas otras clases útiles.



Error común de programación 3.1

Todas las directivas `using` deben aparecer antes de cualquier otro código (excepto los comentarios) en un archivo de código fuente de C#; en caso contrario se produce un error de compilación.



Tip de prevención de errores 3.1

Si olvida incluir una directiva `using` para una clase que se utilice en su aplicación, es muy probable que se produzca un error de compilación en el que se muestre un mensaje como “El nombre ‘Console’ no existe en el contexto actual.” Cuando esto ocurra, verifique que haya proporcionado las directivas `using` apropiadas y que los nombres en las directivas `using` estén deletreados de forma correcta, incluyendo el uso apropiado de letras mayúsculas y minúsculas.

Para cada nueva clase .NET que utilicemos, debemos indicar el espacio de nombres en el que se encuentra. Esta información es importante, ya que nos ayuda a localizar las descripciones de cada clase en la **documentación de .NET**. Puede encontrar una versión basada en Web de este documento en las siguientes páginas Web:

msdn2.microsoft.com/en-us/library/ms229335 (inglés)
msdn2.microsoft.com/es-es/library/ms306608 (español)

También puede encontrar esta información en la documentación de Visual C# Express, en el menú **Ayuda**. Incluso puede colocar el cursor sobre el nombre de cualquier clase o método .NET y después oprimir *F1* para obtener más información.

La línea 4 es tan sólo una línea en blanco. Las líneas en blanco, los caracteres de espacio y los caracteres de tabulación se conocen como **espacio en blanco**. (En específico, los caracteres de espacio y los tabuladores se conocen como **caracteres de espacio en blanco**.) El compilador ignora todo el espacio en blanco. En este capítulo y en algunos de los siguientes hablaremos sobre las convenciones para el uso de espacio en blanco, con el fin mejorar la legibilidad de las aplicaciones.

La línea 5

```
public class Bienvenido1
```

comienza una **declaración de clase** para la clase Bienvenido1. Toda aplicación consiste de cuando menos una declaración de clase, la cual usted (como programador) define. A estas clases se les conoce como **clases definidas por el usuario**. La **palabra clave class** introduce una declaración de clase y va seguida inmediatamente del **nombre de la clase** (Bienvenido1). Las palabras clave (conocidas como **palabras reservadas**) están reservadas para uso exclusivo del lenguaje C# y siempre se escriben en minúsculas. En la figura 3.2 se muestra la lista completa de palabras clave de C#.

Por convención, todos los nombres de las clases comienzan con mayúscula y la primera letra de cada palabra que incluyen también se escribe en mayúscula (por ejemplo, NombreClaseEjemplo). Por lo general, a esto se le conoce como **estilo de mayúsculas/minúsculas Pascal**. El nombre de una clase es un **identificador**, una serie de caracteres que consiste de letras, dígitos y guiones bajos (_), que no comienza con un dígito y que no contiene espacios. Algunos identificadores válidos son Bienvenido1, identificador, _valor y m_campoEntrada1. El nombre 7boton no es un identificador válido, ya que comienza con un dígito; el nombre campo_entrada tampoco lo es, ya que contiene un espacio. Por lo general, un identificador que no comienza con letra mayúscula no es el nombre de una clase. C# es **sensible a mayúsculas y minúsculas**, es decir, las letras mayúsculas y minúsculas son distintas, por lo que a1 y A1 son identificadores distintos (pero ambos son válidos). También se puede colocar el carácter @ al principio de un identificador. Esto indica que una palabra debe interpretarse como identificador, aun y cuando sea una palabra clave (por ejemplo, @int). De esta forma el código de C# puede utilizar código escrito en otros lenguajes .NET, en los que un identificador podría tener el mismo nombre que una palabra clave de C#.

Palabras clave de C#				
abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Figura 3.2 | Palabras clave de C#.



Buena práctica de programación 3.1

Por convención, siempre se debe comenzar el identificador del nombre de una clase con letra mayúscula y cada palabra subsiguiente en el identificador se debe comenzar con letra mayúscula.



Error común de programación 3.2

C# es sensible a mayúsculas y minúsculas. Por lo general, si no se utiliza la combinación apropiada de letras mayúsculas y minúsculas para un identificador, se produce un error de compilación.

En los capítulos del 3 al 8, cada una de las clases que definimos comienza con la palabra clave **public**. Por ahora sólo requeriremos esta palabra clave. En el capítulo 9 aprenderá más acerca de las clases **public** y no **public**. Cuando guarda su declaración de clase **public** en un archivo, por lo general el nombre del archivo es el de la clase, seguido de la extensión de nombre de archivo **.cs**. Para nuestra aplicación, el nombre de archivo es **Bienvenido1.cs**.



Buena práctica de programación 3.2

*Por convención, un archivo que contiene una sola clase **public** debe tener un nombre que sea idéntico al nombre de la clase (más la extensión **.cs**) en términos tanto de ortografía como de uso de mayúsculas y minúsculas. Si nombra sus archivos de esta forma facilitará a los demás programadores (y a usted) el proceso de determinar la ubicación de las clases de una aplicación.*

Una **llave izquierda** (en la línea 6 de la figura 3.1), **{**, comienza el cuerpo de cualquier declaración de clase. Su correspondiente **llave derecha** (en la línea 12), **}**, debe terminar cada declaración de clase. Observe que las líneas de la 7 a la 11 tienen sangría. Esta sangría es una de las convenciones de espaciado que mencionamos antes. Definiremos cada convención de espaciado como una Buena práctica de programación.



Tip de prevención de errores 3.2

*Siempre que escriba una llave izquierda de apertura, **{**, en su aplicación, escriba de inmediato la llave derecha de cierre, **}**, y después reposicione el cursor entre las llaves y utilice sangría para comenzar a escribir el cuerpo. Esta práctica le ayudará a evitar errores por la omisión de llaves.*



Buena práctica de programación 3.3

*Aplique sangría a todo el cuerpo de cada declaración de clase; utilice un “nivel” de sangría entre las llaves izquierda y derecha que delimiten el cuerpo de su clase. Este formato enfatiza la estructura de la declaración de la clase y mejora la legibilidad. Para dejar que el IDE aplique formato a su código, seleccione **Edición > Avanzado > Dar formato al documento**.*



Buena práctica de programación 3.4

*Establezca una convención para el tamaño de sangría que prefiera y después aplique esa convención de manera uniforme. Puede utilizar la tecla **Tab** para crear sangrías, pero los marcadores de tabulación varían entre los distintos editores de texto. Le recomendamos que utilice tres espacios para formar cada nivel de sangría. En la sección 3.3 le mostraremos cómo hacer esto.*



Error común de programación 3.3

Si las llaves no se escriben en pares se produce un error de sintaxis.

La línea 7

```
// El método Main comienza la ejecución de la aplicación de C#
```

es un comentario que indica el propósito de las líneas 8-11 de la aplicación. La línea 8

```
public static void Main( string[] args )
```

es el punto de inicio de toda aplicación. Los **paréntesis** después del identificador `Main` indican que es un bloque de construcción de aplicaciones llamado método. Por lo general, las declaraciones de clases contienen uno o más métodos. Sus nombres siguen casi siempre las convenciones de estilo de mayúsculas/minúsculas Pascal que se utilizan para los nombres de las clases. Para cada aplicación es obligatorio que uno de los métodos en una clase se llame `Main` (que por lo general se define como se muestra en la línea 8); de no ser así, la aplicación no se ejecutará. Los métodos pueden realizar tareas y devolver información cuando completan su trabajo. La palabra clave **`void`** (línea 8) indica que este método no devolverá información después de completar su tarea. Más adelante veremos que muchos métodos sí devuelven información; en los capítulos 4 y 7 aprenderá más acerca de ellos. Por ahora, sólo imite la primera línea de `Main` en sus aplicaciones.

La llave izquierda en la línea 9 comienza el **cuerpo de la declaración del método**. El cuerpo del método debe terminar con su correspondiente llave derecha (línea 11 de la figura 3.1). Observe que la línea 10 en el cuerpo del método tiene sangría entre las llaves.



Buena práctica de programación 3.5

Al igual que con las declaraciones de clases, debe aplicar sangría a todo el cuerpo de la declaración de cada método; para ello utilice un "nivel" de sangría entre las llaves izquierda y derecha que definen el cuerpo del método. Este formato hace que la estructura del método resalte y mejora su legibilidad.

La línea 10

```
Console.WriteLine( "¡Bienvenido a la programación en C#!" );
```

instruye a la computadora para que **realice una acción**; a saber, imprimir (es decir, mostrar en pantalla) la **cadena** de caracteres contenida entre los dos símbolos de comillas dobles. A una cadena también se le conoce como **cadena de caracteres**, un **mensaje** o una **literal de cadena**. A los caracteres entre símbolos de comillas dobles les llamaremos simplemente **cadenas**. El compilador no ignora los caracteres de espacio en blanco en las cadenas.

La clase **`Console`** proporciona características de **entrada/salida estándar**, las cuales permiten a las aplicaciones leer y mostrar texto en la ventana de consola desde la cual se ejecuta la aplicación. El **método `Console.WriteLine`** muestra (o imprime) una línea de texto en la ventana de consola. La cadena entre paréntesis en la línea 10 es el **argumento** para el método. La tarea del método `Console.WriteLine` es mostrar (o enviar de salida) su argumento en la ventana de consola. Cuando `Console.WriteLine` completa su tarea, posiciona el **cursor de la pantalla** (el símbolo destellante que indica en dónde se mostrará el siguiente carácter) al principio de la siguiente línea en la ventana de consola. (Este movimiento del cursor es similar a lo que ocurre cuando un usuario oprime *Intro* mientras escribe en un editor de texto: el cursor se desplaza al principio de la siguiente línea en el archivo.)

A toda la línea 10, incluyendo `Console.WriteLine`, los paréntesis, el argumento `"¡Bienvenido a la programación en C#!"` entre paréntesis y el **punto y coma** (`;`), se le conoce como **instrucción**. Cada instrucción termina con un punto y coma. Cuando se ejecuta la instrucción de la línea 10, muestra el mensaje `¡Bienvenido a la programación en C#!` en la ventana de consola. Por lo general, un método está compuesto de una o más instrucciones, que realizan la tarea de ese método.



Error común de programación 3.4

Si se omite el punto y coma al final de una instrucción se produce un error de sintaxis.

Algunos programadores tienen problemas para asociar las llaves izquierda y derecha (`{` y `}`) que delimitan el cuerpo de la declaración de una clase o de un método cuando leen o escriben una aplicación. Por esta razón, es común incluir un comentario después de cada llave derecha de cierre (`}`) que termina la declaración de un método, y después de cada llave derecha de cierre que termina la declaración de una clase. Por ejemplo, la línea 11

```
} // fin del método Main
```

especifica la llave derecha de cierre del método `Main`, y la línea 12

```
} // fin de la clase Bienvenido1
```

especifica la llave derecha de cierre de la clase `Bienvenido1`. Cada uno de estos comentarios indica el método o la clase que termina con esa llave derecha. Visual Studio puede ayudarle a localizar las llaves que concuerden en su código. Sólo tiene que colocar el cursor a un lado de una llave y Visual Studio resaltará la otra.



Buena práctica de programación 3.6

Si después de la llave derecha de cierre del cuerpo de un método o de la declaración de una clase se coloca un comentario que indique el método o la declaración de la clase a la cual pertenece esa llave, se mejora la legibilidad de la aplicación.

3.3 Cómo crear una aplicación simple en Visual C# Express

Ahora que le hemos presentado nuestra primera aplicación de consola (figura 3.1), veremos una explicación paso a paso de cómo compilarla y ejecutarla mediante el uso de Visual C# Express.

Creación de la aplicación de consola

Después de abrir Visual C# 2005 Express, seleccione **Archivo > Nuevo proyecto...** para que aparezca el cuadro de diálogo **Nuevo proyecto** (figura 3.3) y luego seleccione la plantilla **Aplicación de consola**. En el campo **Nombre** del cuadro de diálogo, escriba `Bienvenido1` y haga clic en **Aceptar** para crear el proyecto. Ahora el IDE debe contener la aplicación de consola, como se muestra en la figura 3.4. Observe que la ventana del editor ya contiene algo de código proporcionado por el IDE. Parte de este código es similar al de la figura 3.1. Otra parte no lo es, ya que utiliza características que no hemos visto todavía. El IDE inserta este código adicional para ayudar a organizar la aplicación y proporcionar acceso a ciertas clases comunes en la Biblioteca de clases del .NET Framework; en este punto del libro, este código no se requiere ni es relevante a la discusión de esta aplicación. Por lo tanto, puede eliminarlo.

El esquema de colores del código que utiliza el IDE se llama **resaltado de sintaxis por colores**; este esquema nos ayuda a diferenciar en forma visual los elementos de la aplicación. Las palabras clave aparecen en color azul y el resto del texto en color negro. Cuando hay comentarios, aparecen en color verde. En este libro sombrearemos nuestro código de manera similar; texto en negrita y cursiva para las palabras clave, cursiva para los comentarios, negrita color gris para las literales y constantes, y color negro para el resto del texto. Un ejemplo de literal es la cadena que se pasa a `Console.WriteLine` en la línea 10 de la figura 3.1. Para personalizar los colores que se muestran en el editor de código seleccione **Herramientas > Opciones...** A continuación aparecerá el cuadro de diálogo

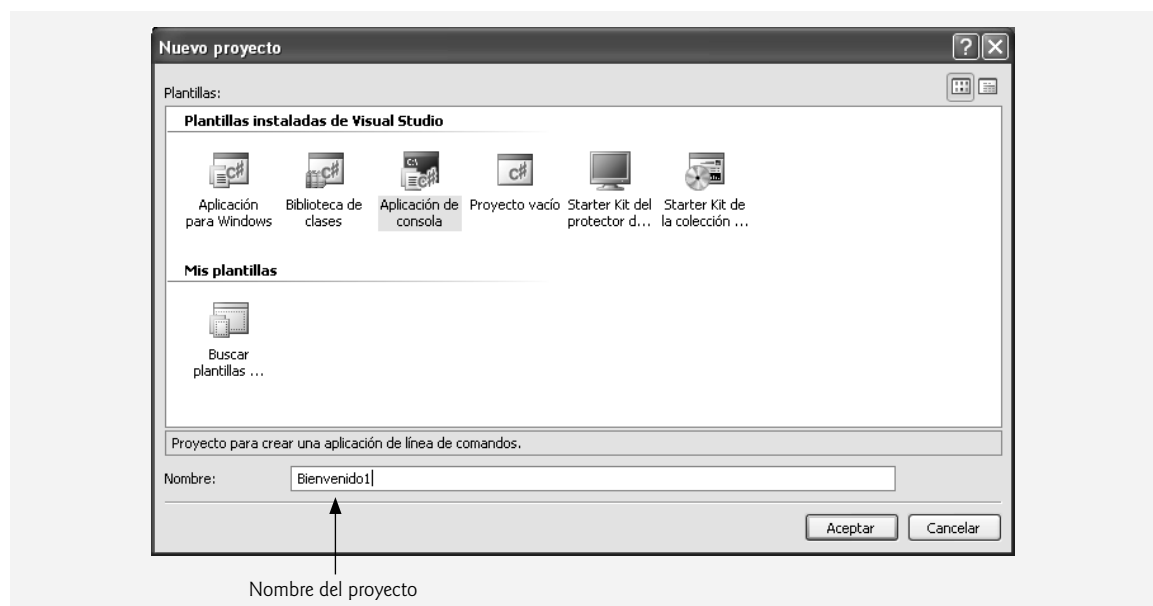


Figura 3.3 | Creación de una **Aplicación de consola** con el cuadro de diálogo **Nuevo proyecto**.

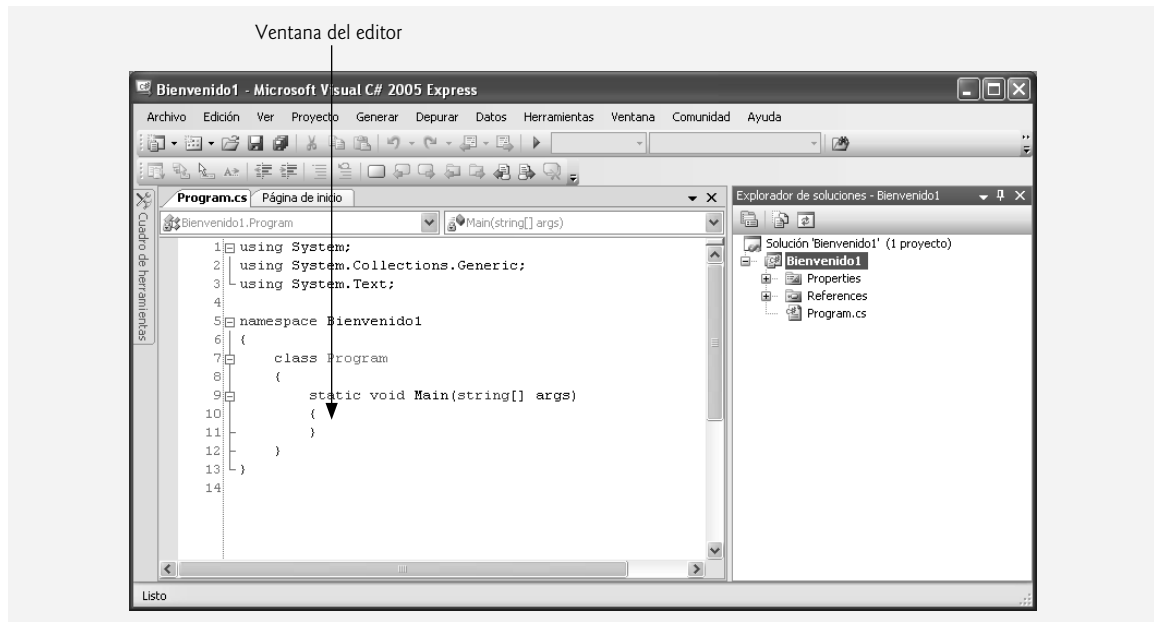


Figura 3.4 | El IDE con una aplicación de consola abierta.

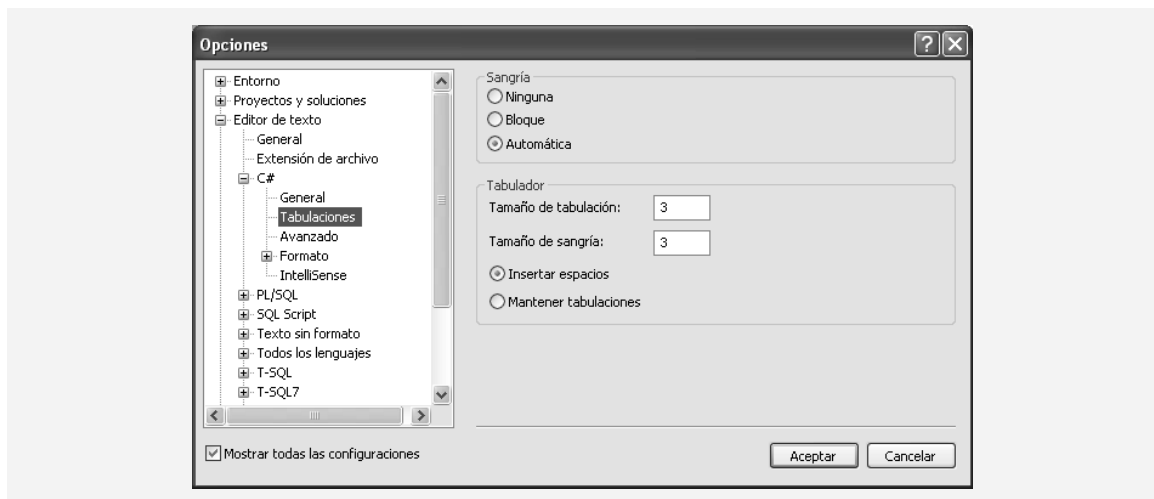


Figura 3.5 | Modificación de las opciones de configuración del IDE.

Opciones (figura 3.5). Después haga clic en el signo más (+) que está a un lado de **Entorno** y seleccione **Fuentes y colores**. Aquí puede modificar los colores para varios elementos de código.

Modificación de la configuración del editor para mostrar números de línea

Visual C# Express le ofrece muchas formas para personalizar su manera de trabajar con el código. En este paso modificará las configuraciones para que su código concuerde con el de este libro. Para que el IDE muestre los números de línea, seleccione **Herramientas > Opciones...** En el cuadro de diálogo que se despliega, haga clic en la casilla de verificación **Mostrar todas las configuraciones** que se encuentra en la parte inferior izquierda, después haga clic en el signo más que está a un lado de **Editor de texto** en el panel izquierdo y seleccione **Todos los lenguajes**. En el panel derecho, active la casilla de verificación **Números de línea**. Mantenga abierto el cuadro de diálogo **Opciones**.

Establecer la sangría del código a tres espacios

En el cuadro de diálogo **Opciones** que abrió en el paso anterior (figura 3.5), haga clic en el signo más que está a un lado de C# en el panel izquierdo y seleccione **Tabulaciones**. Escriba el número **3** en los campos **Tamaño de tabulación** y **Tamaño de sangría**. Todo el nuevo código que agregue utilizará ahora tres espacios para cada nivel de sangría. Haga clic en **Aceptar** para guardar sus nuevas configuraciones, cierre el cuadro de diálogo y regrese a la ventana del editor.

Modificación del nombre del archivo de aplicación

Para las aplicaciones que crearemos en este libro, modificaremos el nombre predeterminado del archivo de aplicación (es decir, Program.cs) para ponerle un nombre más descriptivo. Para cambiar el nombre del archivo, haga clic en el nombre Program.cs, en la ventana **Explorador de soluciones**. A continuación aparecerán las propiedades del archivo de aplicación en la ventana **Propiedades** (figura 3.6). Cambie la **propiedad Nombre de archivo** a Bienvenido1.cs.

Escritura de código

En la ventana del editor (figura 3.4), escriba el código de la figura 3.1. Después de escribir (en la línea 10) el nombre de la clase y un punto (por ejemplo, Console.) aparecerá una ventana con una barra de desplazamiento (figura 3.7). A esta característica del IDE se le llama **IntelliSense**; sirve para listar los **miembros** de una clase, incluyendo los nombres de los métodos. A medida que usted escribe caracteres, Visual C# Express resalta el primer miembro que concuerda con todos los caracteres escritos y después muestra un cuadro de información de herramienta (tool tip), que contiene una descripción de ese miembro. Puede escribir el nombre del miembro completo (por ejemplo, WriteLine), hacer doble clic en el nombre del miembro que se encuentra en la lista u oprimir **Tab** para completar el nombre. Una vez que se proporciona el nombre completo, la ventana **IntelliSense** se cierra.

Cuando escribe el carácter de paréntesis abierto (, después de Console.WriteLine, aparece la ventana **Información de parámetros** (figura 3.8). Esta ventana contiene información sobre los parámetros del método. Como aprenderá en el capítulo 7, puede haber varias versiones de un método; esto es, una clase puede definir varios métodos que tengan el mismo nombre, siempre y cuando tengan distintos números y/o tipos de parámetros.

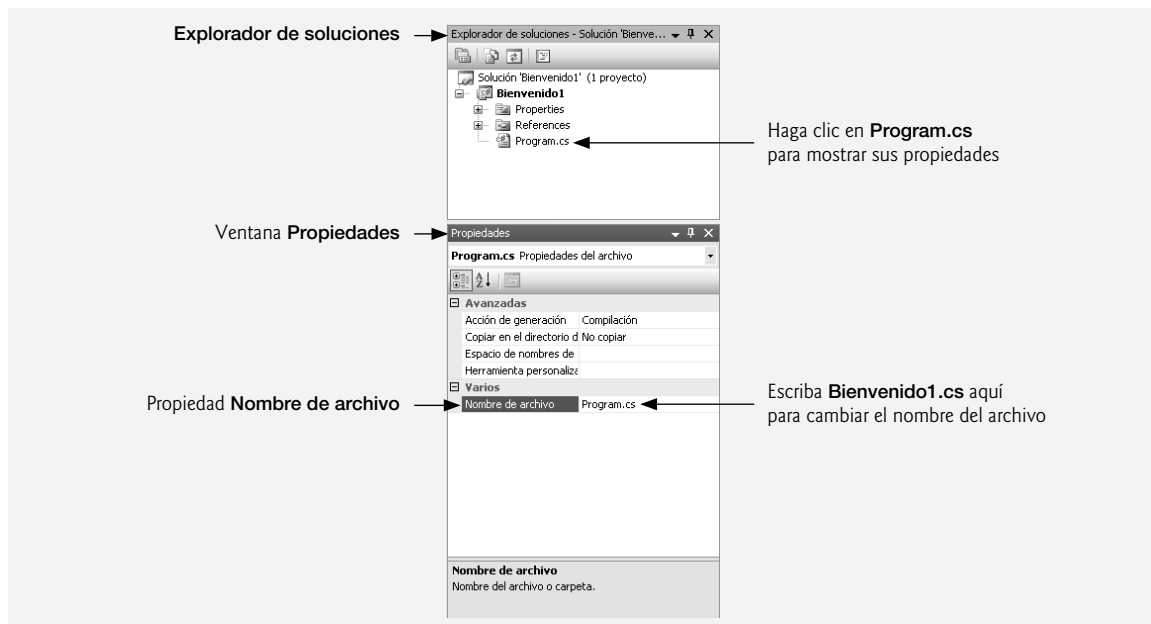


Figura 3.6 | Modificación del nombre del programa en la ventana **Propiedades**.

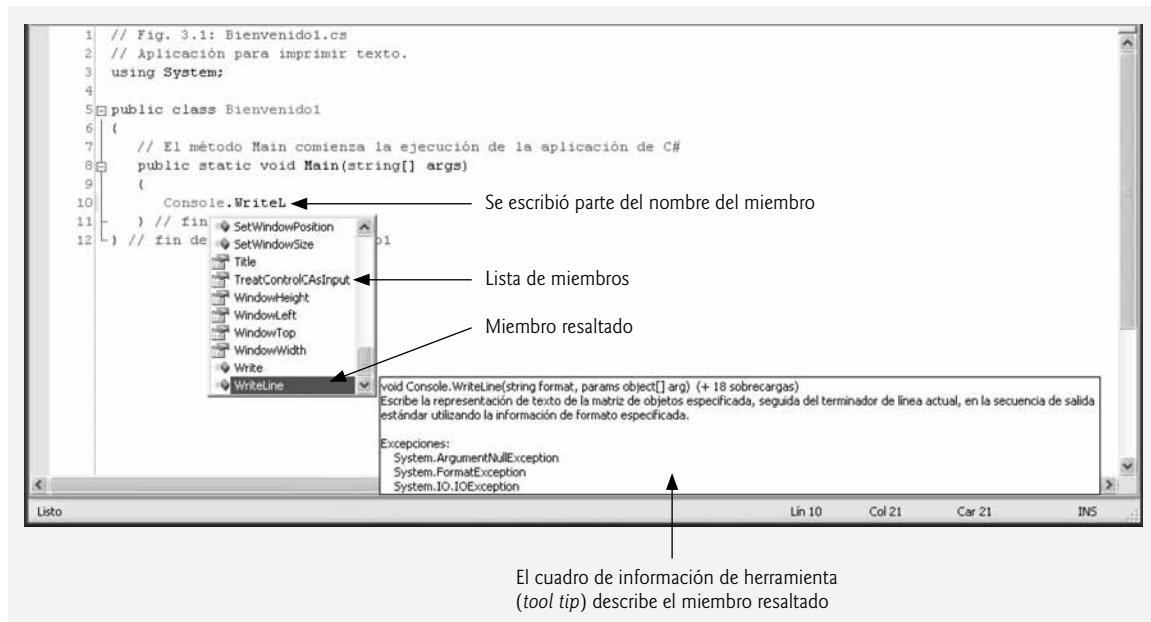


Figura 3.7 | Característica *IntelliSense* de Visual C# Express.

Por lo general, todos estos métodos realizan tareas similares. La ventana *Información de parámetros* indica cuántas versiones del método seleccionado hay disponibles y proporciona flechas hacia arriba y hacia abajo para desplazarse a través de las distintas versiones. Por ejemplo, hay 19 versiones del método `WriteLine`; en nuestra aplicación utilizamos una de esas 19 versiones. La ventana *Información de parámetros* es una de las muchas características que ofrece el IDE para facilitar el desarrollo de aplicaciones. En los siguientes capítulos aprenderá más acerca de la información que se muestra en estas ventanas. La ventana *Información de parámetros* es útil, en especial, cuando deseamos ver las distintas formas en las que se puede utilizar un método. Del código de la figura 3.1 ya sabemos que nuestra intención es mostrar una cadena con `WriteLine`, y como sabemos con exactitud cuál versión de `WriteLine` deseamos utilizar, por el momento sólo hay que cerrar la ventana *Información de parámetros* mediante la tecla `Esc`.

Guardar la aplicación

Seleccione **Archivo > Guardar todo** para que aparezca el cuadro de diálogo **Guardar proyecto** (figura 3.9). En el cuadro de texto **Ubicación**, especifique el directorio en el que desea guardar el proyecto. Nosotros optamos por guardarlo en el directorio `MisProyectos` de la unidad `C:`. Seleccione la casilla de verificación **Crear directorio para la solución** (para que Visual Studio cree el directorio, en caso de que no exista todavía) y haga clic en **Guardar**.

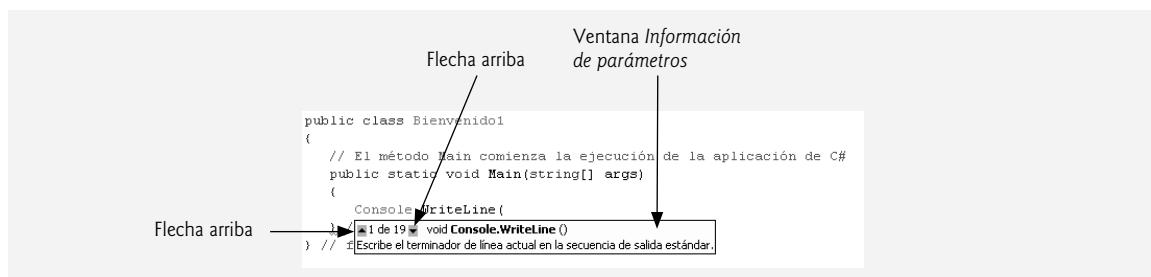


Figura 3.8 | Ventana *Información de parámetros*.

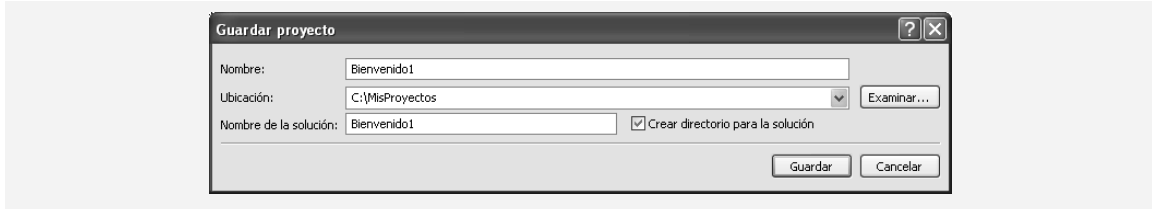


Figura 3.9 | Cuadro de diálogo Guardar proyecto.

Compilación y ejecución de la aplicación

Ahora está listo para compilar y ejecutar su aplicación. Dependiendo del tipo de aplicación, el compilador puede compilar el código en archivos con una **extensión .exe (ejecutable)**, con una **extensión .dll (biblioteca de vínculos dinámicos)** o con alguna otra disponible. Dichos archivos se llaman ensamblados y son las unidades de empaquetamiento para el código de C# compilado. Estos ensamblados contienen el código de Lenguaje intermedio de Microsoft (MSIL) para la aplicación.

Para compilar la aplicación, seleccione **Generar > Generar solución**. Si la aplicación no contiene errores de sintaxis, su aplicación de consola se compilará en un archivo ejecutable (llamado Bienvenido1.exe, en el directorio del proyecto). Para ejecutar esta aplicación de consola (es decir, Bienvenido1.exe), seleccione **Depurar > Iniciar sin depurar** (u oprima <Ctrl> F5), con lo cual se invocará el método Main (figura 3.1). La instrucción en la línea 10 de Main muestra ¡Bienvenido a la programación en C#!. La figura 3.10 muestra los resultados de la ejecución de esta aplicación. Observe que los resultados se despliegan en una ventana de consola. Deje la aplicación abierta en Visual C# Express; más adelante en esta sección volveremos a utilizarla.

Ejecución de la aplicación desde el símbolo del sistema

Como dijimos al principio del capítulo, puede ejecutar aplicaciones fuera del IDE a través del **Símbolo del sistema**. Esto es útil cuando sólo desea ejecutar una aplicación, en vez de abrirla para modificarla. Para abrir el **Símbolo del sistema**, seleccione **Inicio > Todos los programas > Accesorios > Símbolo del sistema**. [Nota: Los usuarios de Windows 2000 deben sustituir Todos los programas con Programas.] La ventana (figura 3.11) muestra la información de derechos de autor, seguida de un símbolo que indica el directorio actual. De manera predeterminada, el símbolo especifica el directorio actual del usuario en el equipo local (en nuestro caso, C:\Documents and Settings\deitel). En su equipo, el nombre de carpeta deitel se sustituirá con su nombre de usuario. Escriba el comando cd (que significa “cambiar directorio”), seguido del modificador /d (para cambiar de unidad, en caso de ser necesario) y después el directorio en el que se encuentra el archivo .exe de la aplicación (es decir, el directorio Release de su aplicación). Por ejemplo, el comando cd/d C:\MisProyectos\Bienvenido1\Bienvenido1\bin\Release (figura 3.12) cambia el directorio actual al directorio Release de la aplicación Bienvenido1 en la unidad C:. El siguiente símbolo muestra el nuevo directorio. Después de cambiar al directorio actual, usted puede ejecutar la aplicación compilada con sólo escribir el nombre del archivo .exe (es decir, Bienvenido1). La apli-



Figura 3.10 | La ejecución de la aplicación que se muestra en la figura 3.1.

cación se ejecutará hasta completarse y después aparecerá de nuevo el símbolo del sistema, esperando el siguiente comando. Para cerrar el **Símbolo del sistema**, escriba `exit` (figura 3.12) y oprima *Intro*.

Visual C# 2005 Express mantiene un directorio **Debug** y un directorio **Release** en el directorio **bin** de cada proyecto. El primero contiene una versión de la aplicación que puede utilizarse con el depurador (vea el apéndice C, Uso del depurador de Visual Studio® 2005). El directorio **Release** contiene una versión optimizada que usted puede proporcionar a sus clientes. En la versión completa de Visual Studio 2005 puede seleccionar la versión específica que desea generar en la lista desplegable **Configuraciones de la solución** de las barras de herramientas que se encuentran en la parte superior del IDE. La versión predeterminada es **Debug**.

[Nota: muchos entornos muestran ventanas **Símbolo del sistema** con fondos negros y texto blanco. Nosotros ajustamos estas configuraciones en nuestro entorno para mejorar la legibilidad de las capturas de pantalla.]

Errores de sintaxis, mensajes de error y la ventana Lista de errores

Regrese a la aplicación en Visual C# Express. Cuando escribe una línea de código y oprime *Intro*, el IDE responde ya sea mediante la aplicación de resaltado de sintaxis por colores o mediante la generación de un **error de sintaxis**, el cual indica una violación a las reglas de Visual C# para crear aplicaciones correctas (es decir, una o más instrucciones no están escritas correctamente). Los errores de sintaxis se producen por diversas razones, como omitir paréntesis y palabras clave mal escritas.

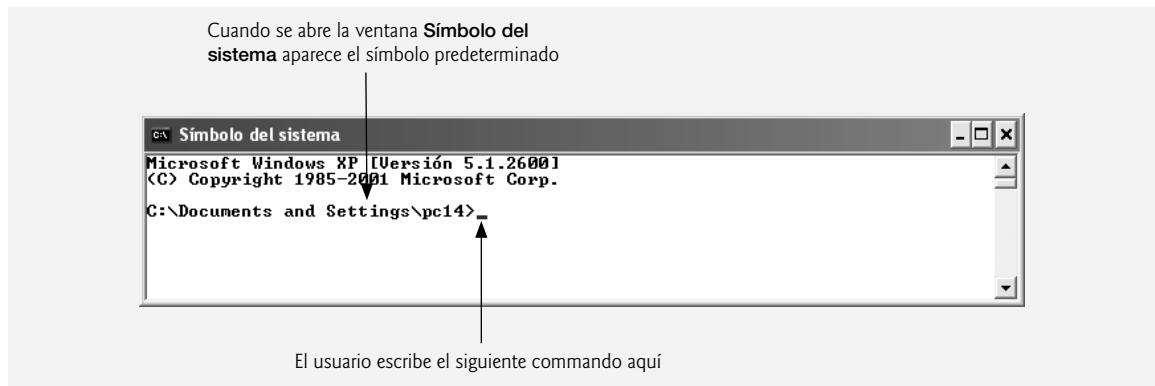


Figura 3.11 | Ejecución de la aplicación que se muestra en la figura 3.1, desde una ventana **Símbolo del sistema**.

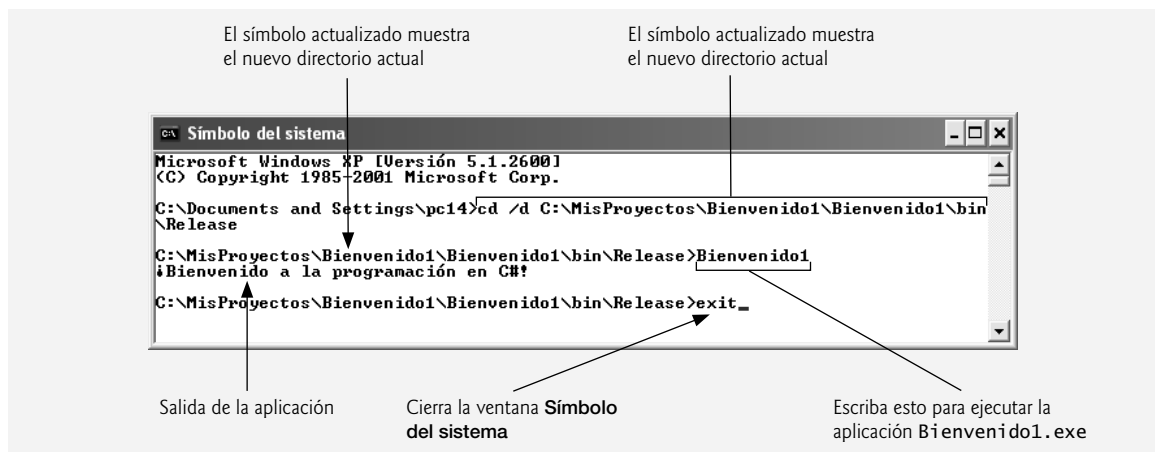


Figura 3.12 | Ejecución de la aplicación que se muestra en la figura 3.1, desde una ventana **Símbolo del sistema**.

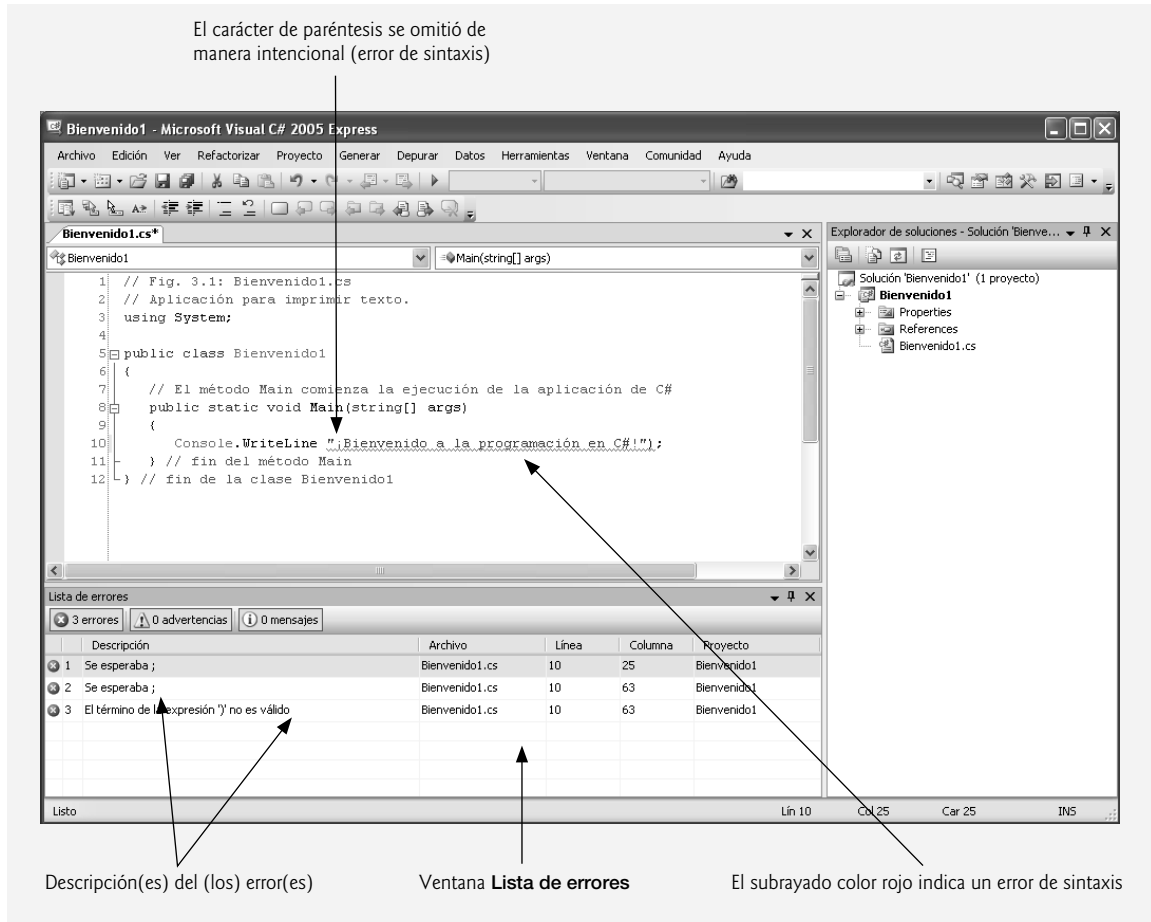


Figura 3.13 | Errores de sintaxis indicados por el IDE.

Cuando se produce un error de sintaxis, el IDE lo subraya en color rojo y proporciona una descripción del error en la **ventana Lista de errores** (figura 3.13). Si la ventana no está visible en el IDE, seleccione **Ver > Lista de errores** para mostrarla. En la figura 3.13 omitimos intencionalmente el primer paréntesis en la línea 10. El primer error contiene el texto **"Se esperaba ;"** y especifica que el error está en la columna 25 de la línea 10. Este mensaje de error aparece cuando el compilador piensa que la línea contiene una instrucción completa, seguida de un punto y coma, y el comienzo de otra instrucción. El segundo error contiene el mismo texto, pero especifica que está en la columna 54 de la línea 10, ya que el compilador piensa que éste es el final de la segunda instrucción. El tercer error tiene el texto **"El término de la expresión ')' no es válido"**, ya que el compilador está confundido por el paréntesis derecho sin su correspondiente paréntesis izquierdo. Aunque estamos tratando de incluir sólo una instrucción en la línea 10, el paréntesis izquierdo que falta ocasiona que el compilador asuma de manera incorrecta que hay más de una instrucción en esa línea, que malinterprete el paréntesis derecho y genere *tres* mensajes de error.



Tip de prevención de errores 3.3

*Un error de sintaxis puede producir varias entradas en la ventana **Lista de errores**. Con cada error que usted corrija, podrá eliminar varios mensajes de error subsiguientes al compilar nuevamente su aplicación. Por lo tanto, cuando vea un error que sabe cómo corregir, hágalo y vuelva a compilar el programa; esto puede hacer que desaparezcan varios errores.*

3.4 Modificación de su aplicación simple en C#

En esta sección continuamos con nuestra introducción a la programación en C# con dos ejemplos que modifican el de la figura 3.1, para imprimir texto en una línea mediante el uso de varias instrucciones e imprimir texto en varias líneas mediante el uso de una sola instrucción.

Mostrar una sola línea de texto con varias instrucciones

El texto "¡Bienvenido a la programación en C#!" puede mostrarse de varias formas. La clase `Bienvenido2`, que se muestra en la figura 3.14, utiliza dos instrucciones para producir el mismo resultado que el que se muestra en la figura 3.1. De aquí en adelante resaltaremos las nuevas características clave en cada listado de código, como se muestra en las líneas 10-11 de la figura 3.14.

La aplicación es casi idéntica a la figura 3.1. Sólo veremos los cambios. La línea 2

```
// Impresión de una línea de texto mediante varias instrucciones.
```

es un comentario que indica el propósito de esta aplicación. La línea 5 comienza la declaración de la clase `Bienvenido2`.

Las líneas 10-11 del método `Main`

```
Console.Write( "¡Bienvenido a " );
Console.WriteLine( "la programación en C#!" );
```

muestran una línea de texto en la ventana de consola. La primera instrucción utiliza el método **`Write`** de `Console` para mostrar una cadena. A diferencia de `WriteLine`, después de mostrar su argumento, el método `Write` no posiciona el cursor de la pantalla al comienzo de la siguiente línea en la ventana de consola; el siguiente carácter que muestre la aplicación aparecerá justo después del último carácter que muestre `Write`. Por ende, la línea 11 posiciona el primer carácter en su argumento (la letra "C") justo después del último carácter que muestra la línea 10 (el carácter de espacio que va antes del carácter de doble comilla de cierre de la cadena). Cada instrucción `Write` continúa mostrando caracteres desde donde la última instrucción `Write` mostró su último carácter.

Mostrar varias líneas de texto mediante una sola instrucción

Una sola instrucción puede mostrar varias líneas mediante el uso de caracteres de nueva línea, que indican a los métodos `Write` y `WriteLine` de `Console` cuándo deben posicionar el cursor de la pantalla en el comienzo de la siguiente línea en la ventana de consola. Al igual que los caracteres de espacio y los tabuladores, los caracteres de nueva línea son caracteres de espacio en blanco. La aplicación de la figura 3.15 muestra cuatro líneas de texto mediante el uso de caracteres de nueva línea, para indicar cuándo se debe comenzar cada nueva línea.

```
1 // Fig. 3.14: Bienvenido2.cs
2 // Impresión de una línea de texto mediante varias instrucciones.
3 using System;
4
5 public class Bienvenido2
6 {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10         Console.Write( "¡Bienvenido a " );
11         Console.WriteLine( "la programación en C#!" );
12     } // fin del método Main
13 } // fin de la clase Bienvenido2
```

```
¡Bienvenido a la programación en C#!
```

Figura 3.14 | Impresión de una línea de texto mediante varias instrucciones.

```

1 // Fig. 3.15: Bienvenido3.cs
2 // Impresión de varias líneas mediante una sola instrucción.
3 using System;
4
5 public class Bienvenido3
6 {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10         Console.WriteLine( "¡Bienvenido\na la\nprogramación en\nC#!" );
11     } // fin del método Main
12 } // fin de la clase Bienvenido3

```

```

¡Bienvenido
a la
programación en
C#!

```

Figura 3.15 | Impresión de varias líneas mediante una sola instrucción.

La mayor parte de la aplicación es idéntica a las aplicaciones de las figuras 3.1 y 3.14, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Impresión de varias líneas mediante una sola instrucción.
```

es un comentario que indica el propósito de esta aplicación. La línea 5 comienza la declaración de la clase Bienvenido3.

La línea 10

```
Console.WriteLine( "¡Bienvenido\na la\nprogramación en\nC#!" );
```

muestra cuatro líneas de texto separadas en la ventana de consola. Por lo general, los caracteres en una cadena se muestran en forma exacta a como aparecen en las comillas dobles. Sin embargo, los dos caracteres \ y n (que se repiten tres veces en la instrucción) no aparecen en la pantalla. A la **barra diagonal inversa** (\) se le conoce como **carácter de escape** y sirve para indicar a C# que hay un “carácter especial” en la cadena. Cuando aparece una barra diagonal inversa en una cadena de caracteres, C# combina el siguiente carácter con la barra diagonal inversa para formar una **secuencia de escape**. La secuencia de escape \n representa el **carácter de nueva línea**. Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir en pantalla mediante métodos de la clase Console, este carácter hace que el cursor de la pantalla se desplace hasta el comienzo de la siguiente línea en la ventana de consola. La figura 3.16 lista varias secuencias de escape comunes y describe cómo afectan a la visualización de caracteres en la ventana de consola.

3.5 Formato del texto con Console.Write y Console.WriteLine

Los métodos Write y WriteLine de Console también tienen la capacidad de mostrar datos con formato. La figura 3.17 imprime en pantalla las cadenas "¡Bienvenido a" y "la programación en C#!" mediante el uso de WriteLine.

La línea 10

```
Console.WriteLine( "{0}\n{1}", "¡Bienvenido a", "la programación en C#!" );
```

llama al método Console.WriteLine para mostrar la salida de la aplicación. La llamada al método especifica tres argumentos. Cuando un método requiere varios argumentos, éstos van separados por **comas** (,); a lo que se le conoce como **lista separada por comas**.

Secuencia de escape	Descripción
\n	Nueva línea. Posiciona el cursor de la pantalla en el comienzo de la siguiente línea.
\t	Tabulación horizontal. Desplaza el cursor de la pantalla a la siguiente marca de tabulación.
\r	Retorno de carro. Posiciona el cursor de la pantalla en el comienzo de la línea actual; no avanza el cursor a la siguiente línea. Cualquier carácter que se imprima en pantalla después del retorno de carro sobrescribirá a los caracteres que se hayan impreso antes en esa línea.
\\	Barra diagonal inversa. Se utiliza para colocar un carácter de barra diagonal inversa en una cadena.
\"	Doble comilla. Se utiliza para colocar un carácter de doble comilla (") en una cadena. Por ejemplo, <pre>Console.Write("\"entre comillas\"");</pre> se muestra en pantalla como <pre>"entre comillas"</pre>

Figura 3.16 | Algunas secuencias de escape comunes.



Buena práctica de programación 3.7

Coloque un espacio después de cada coma (,) en una lista de argumentos para que las aplicaciones tengan mejor legibilidad.

Recuerde que todas las instrucciones terminan con un punto y coma (;). Por lo tanto, la línea 10 sólo representa una instrucción. Las instrucciones largas pueden dividirse en muchas líneas, pero hay algunas restricciones.



Error común de programación 3.5

Si se divide una instrucción a la mitad de un identificador o una cadena, se produce un error de sintaxis.

El primer argumento del método `WriteLine` es una **cadena de formato**, la cual puede consistir de **texto fijo** y **elementos de formato**. El método `WriteLine` imprime en pantalla el texto fijo como se demostró en la figura 3.1. Cada elemento de formato es un receptáculo para un valor. Los elementos de formato también pueden incluir información de formato opcional.

Los elementos de formato van encerrados entre llaves y contienen una secuencia de caracteres, que indican al método qué argumento debe utilizar y cómo darle formato. Por ejemplo, el elemento de formato `{0}` es un

```

1  // Fig. 3.17: Bienvenido4.cs
2  // Impresión de varias líneas de texto con formato de cadenas.
3  using System;
4
5  public class Bienvenido4
6  {
7      // El método Main comienza la ejecución de la aplicación de C#
8      public static void Main( string[] args )
9      {
10         Console.WriteLine( "{0}\n{1}", "¡Bienvenido a", "la programación en C#!" );
11     } // fin del método Main
12 } // fin de la clase Bienvenido4

```

```

¡Bienvenido a
la programación en C#!

```

Figura 3.17 | Impresión de varias líneas de texto con formato de cadenas.

receptáculo para el primer argumento adicional (ya que C# empieza a contar desde 0), {1} es un receptáculo para el segundo argumento, etc. La cadena de formato en la línea 10 especifica que WriteLine debe imprimir en pantalla dos argumentos y que el primero debe ir seguido de un carácter de nueva línea. Por lo tanto, este ejemplo sustituye la cadena "¡Bienvenido a" por el {0} y la cadena "la programación en C#" por el {1}. En la salida en pantalla se muestran dos líneas de texto. Como por lo general las llaves en una cadena con formato indican un receptáculo para la sustitución de texto, debe escribir dos llaves izquierdas ({}) o dos llaves derechas (}) para poder insertar una llave izquierda o derecha respectivamente en una cadena con formato. Conforme sea necesario, iremos presentando características adicionales de formato en nuestros ejemplos.

3.6 Otra aplicación en C#: suma de enteros

Nuestra siguiente aplicación lee (o recibe como entrada) dos *enteros* (números como -22, 7, 0 y 1024) que escribe un usuario en el teclado, calcula la suma de los valores y muestra el resultado. Esta aplicación debe llevar el registro de los números que suministra el usuario para realizar el cálculo más adelante. Las aplicaciones recuerdan números y otros datos en la memoria de la computadora y acceden a ellos a través de ciertos elementos, conocidos como *variables*. La aplicación de la figura 3.18 muestra estos conceptos. En la salida de ejemplo, resaltamos, en negrita, los datos que escribe el usuario desde el teclado.

Las líneas 1-2

```
// Fig. 3.18: Suma.cs
// Muestra la suma de dos números que se introducen desde el teclado.
```

indican el número de la figura, el nombre del archivo y el propósito de la aplicación.

La línea 5

```
public class Suma
```

comienza la declaración de la clase Suma. Recuerde que el cuerpo de cada declaración de clase comienza con una llave izquierda de apertura (línea 6) y termina con una llave derecha de cierre (línea 26).

La aplicación comienza su ejecución con el método Main (líneas 8-25). La llave izquierda (línea 9) marca el comienzo del cuerpo de Main y la correspondiente llave derecha (línea 25) marca el final. Observe que el método Main tiene un nivel de sangría dentro del cuerpo de la clase Suma y que el código en el cuerpo de Main tiene otro nivel de sangría, para mejorar la legibilidad.

```
1 // Fig. 3.18: Suma.cs
2 // Muestra la suma de dos números que se introducen desde el teclado.
3 using System;
4
5 public class Suma
6 {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10         int numero1; // declara el primer número a sumar
11         int numero2; // declara el segundo número a sumar
12         int suma; // declara la suma de numero1 y numero2
13
14         Console.Write( "Escriba el primer entero: " ); // mensaje para el usuario
15         // lee el primer número del usuario
16         numero1 = Convert.ToInt32( Console.ReadLine() );
17
18         Console.Write( "Escriba el segundo entero: " ); // mensaje para el usuario
19         // lee el segundo número del usuario
```

Figura 3.18 | Impresión en pantalla de la suma de dos números introducidos desde el teclado. (Parte I de 2).

```

20     numero2 = Convert.ToInt32( Console.ReadLine() );
21
22     suma = numero1 + numero2; // suma los números
23
24     Console.WriteLine( "La suma es {0}", suma ); // muestra la suma
25 } // fin del método Main
26 } // fin de la clase Suma

```

```

Escriba el primer entero: 45
Escriba el segundo entero: 72
La suma es 117

```

Figura 3.18 | Impresión en pantalla de la suma de dos números introducidos desde el teclado. (Parte 2 de 2).

La línea 10

```
int numero1; // declara el primer número a sumar
```

es una **instrucción de declaración de variable** (también llamada **declaración**), la cual especifica el nombre y el tipo de una variable (`numero1`) que se utilizará en esta aplicación. El nombre de una variable permite que la aplicación acceda al valor de esa variable en memoria; el nombre puede ser cualquier identificador válido (en la sección 3.2 podrá consultar los requerimientos de nomenclatura para los identificadores). El tipo de una variable especifica qué tipo de información se almacena en esa ubicación en memoria. Al igual que las demás instrucciones, las de declaración terminan con un punto y coma (;).

La declaración en la línea 10 especifica que la variable llamada `numero1` es de tipo **int**: puede almacenar valores **enteros** (números como 7, -11, 0 y 31914). El rango de valores para un `int` es de -2,147,483,648 (`int.MinValue`) a +2,147,483,647 (`int.MaxValue`). Pronto hablaremos sobre los tipos **float**, **double** y **decimal** para especificar números reales, y del tipo **char** para especificar caracteres. Los números reales contienen puntos decimales, como 3.4, 0.0 y -11.19. Las variables de tipo `float` y `double` almacenan, en memoria, aproximaciones de números reales. Las variables de tipo `decimal` almacenan números reales con precisión (hasta 28-29 dígitos significativos), por lo que se utilizan con frecuencia en cálculos monetarios. Las variables de tipo `char` representan caracteres individuales, como una letra mayúscula (por ejemplo, A), un dígito (por ejemplo, 7), un carácter especial (por ejemplo, * o %) o una secuencia de escape (por ejemplo, el carácter de nueva línea, `\n`). Por lo general, a los tipos como `int`, `float`, `double`, `decimal` y `char` se les conoce como **tipos simples**. Los nombres de tipo simple son palabras clave y deben aparecer todos en minúscula. En el apéndice I se sintetizan las características de los trece tipos simples (`bool`, `byte`, `sbyte`, `char`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` y `decimal`).

Las instrucciones de declaración de variables en las líneas 11-12

```
int numero2; // declara el segundo número a sumar
int suma; // declara la suma de numero1 y numero2
```

declaran de manera similar las variables `numero2` y `suma` como de tipo `int`.

Las instrucciones de declaración de variables pueden dividirse en varias líneas, con los nombres de las variables separados por comas (es decir, una lista separada por comas de nombres de variables). Pueden declararse distintas variables del mismo tipo en una o en varias declaraciones. Por ejemplo, las líneas 10-12 también pueden escribirse de la siguiente manera:

```
int numero1, // declara el primer número a sumar
    numero2, // declara el segundo número a sumar
    suma; // declara la suma de numero1 y numero2
```



Buena práctica de programación 3.8

Declare cada variable en una línea separada. Este formato permite insertar fácilmente un comentario a continuación de cada declaración.

**Buena práctica de programación 3.9**

Seleccionar nombres de variables significativos ayuda a que un programa se **autodocumente** (es decir, que sea más fácil entender el código con sólo leerlo, en lugar de leer manuales o ver un número excesivo de comentarios).

**Buena práctica de programación 3.10**

Por convención, los identificadores de nombres de variables empiezan con una letra minúscula y cada una de las palabras en el nombre, que van después de la primera, deben empezar con una letra mayúscula. A esta convención se le conoce como **estilo de mayúsculas/minúsculas Camel**.

La línea 14

```
Console.Write( "Escriba el primer entero: " ); // mensaje para el usuario
```

utiliza `Console.Write` para mostrar el mensaje "Escriba el primer entero: ". Este mensaje se llama **indicador**, ya que indica al usuario que debe realizar una acción específica.

La línea 16

```
numero1 = Convert.ToInt32( Console.ReadLine() );
```

funciona en dos pasos. Primero llama al método **ReadLine** de `Console`. Este método espera a que el usuario escriba una cadena de caracteres en el teclado y que oprima *Intro* para enviar la cadena a la aplicación. Después, la cadena se utiliza como un argumento para el método **ToInt32** de la clase **Convert**, el cual convierte esta secuencia de caracteres en datos de tipo `int`. Como dijimos antes en este capítulo, algunos métodos realizan una tarea y después devuelven el resultado. En este caso, el método `ToInt32` devuelve la representación `int` de la entrada del usuario.

Técnicamente, el usuario puede escribir cualquier cosa como valor de entrada. `ReadLine` lo aceptará y lo pasará al método `ToInt32`. Este método asume que la cadena contiene un valor entero válido. En esta aplicación, si el usuario escribe un valor no entero, se producirá un error lógico en tiempo de ejecución y la aplicación terminará. En el capítulo 12, Manejo de excepciones, veremos cómo hacer que sus aplicaciones sean más robustas al permitir que manejen dichos errores y continúen ejecutándose. A esto también se le conoce como hacer que su aplicación sea **tolerante a fallas**.

En la línea 16, el resultado de la llamada al método `ToInt32` (un valor `int`) se coloca en la variable `numero1` mediante el uso del **operador de asignación**, `=`. La instrucción se lee como "numero1 obtiene el valor devuelto por `Convert.ToInt32`". Al operador `=` se le denomina **operador binario**, ya que tiene dos **operandos**: `numero1` y el resultado de la llamada al método `Convert.ToInt32`. Esta instrucción se llama **instrucción de asignación**, ya que asigna un valor a una variable. Todo lo que esté a la derecha del operador de asignación (`=`) siempre se evalúa antes de que se realice la asignación.

**Buena práctica de programación 3.11**

Coloque espacios en ambos lados de un operador binario para hacerlo que resalte y que el código sea más legible.

La línea 18

```
Console.Write( "Escriba el segundo entero: " ); // mensaje para el usuario
```

pide al usuario que escriba el segundo entero. La línea 20

```
numero2 = Convert.ToInt32( Console.ReadLine() );
```

lee un segundo entero y lo asigna a la variable `numero2`.

La línea 22

```
suma = numero1 + numero2; // suma los números
```

es una instrucción de asignación que calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma` mediante el uso del operador de asignación, `=`. La mayoría de los cálculos se realiza en instrucciones

de asignación. Cuando la aplicación encuentra el operador de suma, utiliza los valores almacenados en las variables `numero1` y `numero2` para realizar el cálculo. En la instrucción anterior, el operador de suma es un operador binario; sus dos **operandos** son `numero1` y `numero2`. A las porciones de las instrucciones que contienen cálculos se les llama **expresiones**. De hecho, una expresión es cualquier porción de una instrucción que tiene un valor asociado. Por ejemplo, el valor de la expresión `numero1 + numero2` es la suma de los números. De manera similar, el valor de la expresión `Console.ReadLine()` es la cadena de caracteres que escribe el usuario.

Después de realizar el cálculo, la línea 24

```
Console.WriteLine( "La suma es {0}", suma ); // muestra la suma
```

utiliza el método `Console.WriteLine` para mostrar la suma. El elemento de formato `{0}` es un receptáculo para el primer argumento después de la cadena de formato. Aparte del elemento de formato `{0}`, el resto de los caracteres en la cadena de formato son texto fijo. Por lo tanto, el método `WriteLine` muestra "La suma es", seguida del valor de suma (en la posición del elemento de formato `{0}`) y una nueva línea.

Los cálculos también pueden realizarse dentro de instrucciones de salida. Podríamos haber combinado las instrucciones en las líneas 22 y 24 en la instrucción

```
Console.WriteLine( "La suma es {0}", ( numero1 + numero2 ) );
```

Los paréntesis alrededor de la expresión `numero1 + numero2` no son requeridos; se incluyen para enfatizar que el valor de la expresión `numero1 + numero2` se imprime en pantalla en la posición del elemento de formato `{0}`.

3.7 Conceptos sobre memoria

Los nombres de las variables como `numero1`, `numero2` y `suma` en realidad corresponden a **ubicaciones** en la memoria de la computadora. Cada variable tiene un **nombre**, un **tipo**, un **tamaño** y un **valor**.

En la aplicación de suma de la figura 3.18, cuando la instrucción (línea 16)

```
numero1 = Convert.ToInt32( Console.ReadLine() );
```

se ejecuta, el número escrito por el usuario se coloca en una **ubicación de memoria** a la que el compilador asigna el nombre `numero1`. Suponga que el usuario escribe **45**. La computadora coloca ese valor entero en la ubicación `numero1`, como se muestra en la figura 3.19. Siempre que se coloca un valor en una ubicación de memoria, ese valor sustituye al anterior en esa ubicación, y el valor anterior se pierde.

Cuando la instrucción (línea 20)

```
numero2 = Convert.ToInt32( Console.ReadLine() );
```

se ejecuta, suponga que el usuario escribe 72. La computadora coloca ese valor entero en la ubicación `numero2`. La memoria ahora aparece como se muestra en la figura 3.20.

Después de que la aplicación de la figura 3.18 obtiene valores para `numero1` y `numero2`, los suma y coloca el resultado en la variable `suma`. La instrucción (línea 22)

```
suma = numero1 + numero2; // suma los números
```

realiza la suma y después sustituye el valor anterior de suma. Después de calcular el valor de suma, la memoria aparece como se muestra en la figura 3.21. Observe que los valores de `numero1` y `numero2` aparecen en la misma forma como se utilizaron en el cálculo de suma. Estos valores se utilizaron, pero no se destruyeron, cuando la computadora realizó el cálculo; cuando se lee un valor de una ubicación de memoria, el proceso es no destructivo.

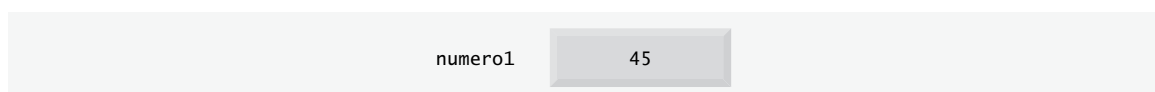


Figura 3.19 | Ubicación de memoria que muestra el nombre y el valor de la variable `numero1`.

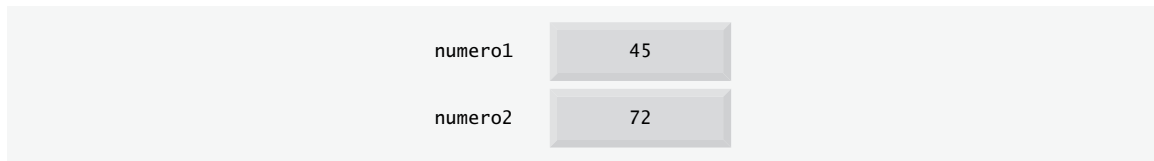


Figura 3.20 | Las ubicaciones de memoria después de almacenar valores para numero1 y numero2.

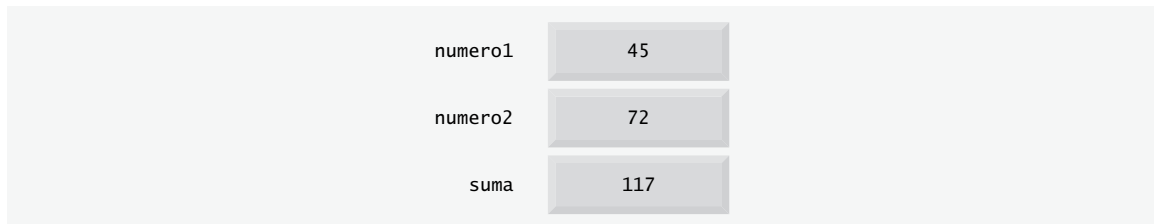


Figura 3.21 | Las ubicaciones de memoria, después de calcular y almacenar la suma de numero1 y numero2.

3.8 Aritmética

La mayoría de las aplicaciones realiza cálculos aritméticos. Los **operadores aritméticos** se sintetizan en la figura 3.22. Observe el uso de varios símbolos especiales que no se utilizan en álgebra. El **asterisco** (*) indica la multiplicación y el **signo porcentual** (%) es el **operador residuo** (en algunos lenguajes se le llama módulo), los cuales veremos en breve. Los operadores aritméticos de la figura 3.22 son operadores binarios; por ejemplo, la expresión $f + 7$ contiene el operador binario $+$ y los dos operandos f y 7 .

La **división de enteros** produce un cociente entero; por ejemplo, la expresión $7 / 4$ se evalúa como 1 y la expresión $17 / 5$ se evalúa como 3. Cualquier parte fraccional en una división de enteros se descarta (es decir, se trunca); no se realiza ningún redondeo. C# cuenta con el operador residuo (%), el cual produce el residuo después de la división. La expresión $x \% y$ produce el residuo después de que x se divide entre y . Por lo tanto, $7 \% 4$ produce 3 y $17 \% 5$ produce 2. Por lo general este operador se utiliza con operandos enteros, pero también puede utilizarse con operandos tipo `float`, `double` y `decimal`. En capítulos posteriores consideraremos varias aplicaciones interesantes del operador residuo, como determinar si un número es múltiplo de otro.

Las expresiones aritméticas deben escribirse en **formato de línea recta** para facilitar la introducción de aplicaciones en la computadora. Por ende, las expresiones como “a dividida entre b” deben escribirse como a / b , de manera que todas las constantes, variables y operadores aparezcan en una línea recta. Por lo general, los compiladores no aceptan la siguiente notación algebraica:

$$\frac{a}{b}$$

Operación de C#	Operador aritmético	Expresión algebraica	Expresión en C#
Suma	+	$f + 7$	<code>f + 7</code>
Resta	-	$p - c$	<code>p - c</code>
Multiplicación	*	$b \cdot m$	<code>b * m</code>
División	/	x/y o $\frac{x}{y}$ o $x \div y$	<code>x / y</code>
Residuo	%	$R \bmod s$	<code>r % s</code>

Figura 3.22 | Operadores aritméticos.

Los paréntesis se utilizan para agrupar términos en expresiones de C#, de la misma forma que en las expresiones algebraicas. Por ejemplo, para multiplicar a por la cantidad $b + c$, escribimos

$$a * (b + c)$$

Si una expresión contiene **paréntesis anidados**, como

$$((a + b) * c)$$

la expresión en el conjunto más interno de paréntesis ($a + b$ en este caso) se evalúa primero.

C# aplica los operadores en las expresiones aritméticas en una secuencia precisa, la cual se determina en base a las siguientes **reglas de precedencia de operadores**, que por lo general son las mismas que las que se siguen en álgebra (figura 3.23):

1. Primero se aplican las operaciones de multiplicación, división y residuo. Si una expresión contiene varias de estas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.
2. Después se aplican las operaciones de suma y resta. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de suma y restas tienen el mismo nivel de precedencia.

Estas reglas permiten a C# aplicar operadores en el orden correcto. Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su **asociatividad**. Más adelante verá que algunos operadores se asocian de derecha a izquierda. La figura 3.23 sintetiza estas reglas de precedencia de operadores. Expandiremos la tabla a medida que introduzcamos más operadores. En el apéndice A se incluye una tabla completa de precedencia.

Ahora consideremos varias expresiones en vista de las reglas de precedencia de los operadores. Cada ejemplo lista una expresión algebraica y su equivalente en C#. El siguiente es un ejemplo de una media aritmética (promedio) de cinco términos:

$$\text{Algebra:} \quad m = \frac{a + b + c + d + e}{5}$$

$$\text{C\#:} \quad m = (a + b + c + d + e) / 5;$$

Los paréntesis son requeridos, ya que la división tiene mayor precedencia que la suma. Toda la cantidad $(a + b + c + d + e)$ debe dividirse entre 5. Si se omiten los paréntesis por error, obtenemos $a + b + c + d + e / 5$, lo cual se evalúa como

$$a + b + c + d + \frac{e}{5}$$

Operador(es)	Operación(es)	Orden de evaluación (asociatividad)
<i>Se evalúa primero</i>		
*	Multiplicación	Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
/	División	
%	Residuo	
<i>Se evalúa después</i>		
+	Suma	Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
-	Resta	

Figura 3.23 | Precedencia de los operadores aritméticos.

A continuación se muestra un ejemplo de la ecuación de una línea recta:

Álgebra: $y = mx + b$

C#: $y = m * x + b;$

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene mayor precedencia que la suma. La asignación se realiza al último, ya que tiene una menor precedencia que la multiplicación o la suma.

El siguiente ejemplo contiene operaciones de residuo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr \% q + w/x - y$

C#: $z = p * r \% q + w / x - y;$



Los números dentro de los círculos bajo la instrucción indican el orden en el que C# aplica los operadores. Las operaciones de multiplicación, residuo y división se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma y la resta. Las operaciones de suma y resta se evalúan a continuación. Estas operaciones también se aplican de izquierda a derecha.

Para desarrollar una mejor comprensión de las reglas de precedencia de los operadores, considere la evaluación de un polinomio de segundo grado ($y = ax^2 + bx + c$):

$y = a * x * x + b * x + c;$



Los números dentro de los círculos indican el orden en el que C# aplica los operadores. Las operaciones de multiplicación se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma. Las operaciones de suma se evalúan a continuación y se aplican de izquierda a derecha. Como no hay operador aritmético para la exponenciación en C#, x^2 se representa como $x * x$. La sección 6.4 muestra una alternativa para realizar la exponenciación en C#.

Suponga que a , b , c y x en el polinomio anterior de segundo grado se inicializan (reciben valores) de la siguiente manera: $a = 2$, $b = 3$, $c = 7$ y $x = 5$. La figura 3.24 ilustra el orden en el que se aplican los operadores.

Al igual que en álgebra, es aceptable colocar paréntesis innecesarios en una expresión para mejorar su legibilidad. A éstos se les llama **paréntesis redundantes**. Por ejemplo, podrían utilizarse paréntesis en la instrucción de asignación anterior para resaltar sus términos de la siguiente manera:

$y = (a * x * x) + (b * x) + c;$

3.9 Toma de decisiones: operadores de igualdad y relacionales

Una **condición** es una expresión que puede ser **verdadera** o **falsa**. En esta sección introducimos una versión simple de la **instrucción if** de C#, que permite que una aplicación tome una **decisión** con base en el valor de una condición. Por ejemplo, la condición “calificación es mayor o igual a 60” determina si un estudiante pasó una prueba. Si la condición en una instrucción **if** es verdadera, se ejecuta el cuerpo de la instrucción **if**. Si la condición es falsa, el cuerpo no se ejecuta. En breve veremos un ejemplo.

Las condiciones en las instrucciones **if** pueden formarse mediante el uso de los **operadores de igualdad** (**=** y **!=**) y los **operadores relacionales** (**>**, **<**, **>=** y **<=**), los cuales se sintetizan en la figura 3.25. Los dos operadores de igualdad (**=** y **!=**) tienen cada uno el mismo nivel de precedencia, los operadores relacionales (**>**, **<**, **>=** y **<=**) tienen cada uno el mismo nivel de precedencia y los operadores de igualdad tienen menor precedencia que los operadores relacionales. Todos se asocian de izquierda a derecha.

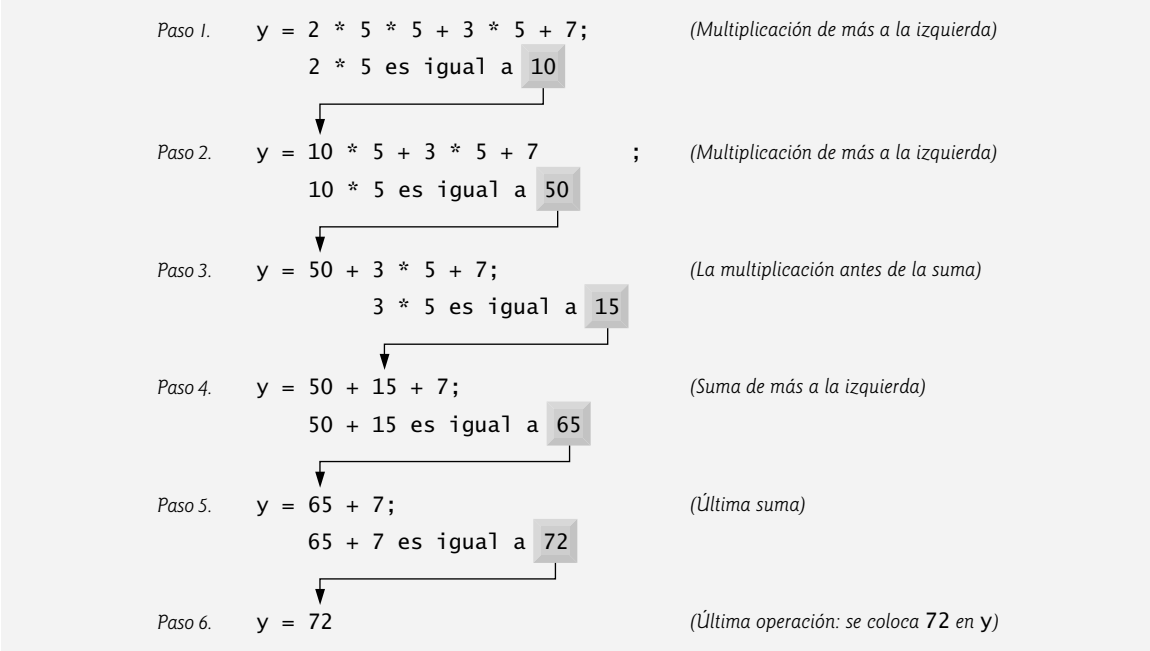


Figura 3.24 | Orden en el que se evalúa un polinomio de segundo grado.

La aplicación de la figura 3.26 utiliza seis instrucciones `if` para comparar dos enteros introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es verdadera, se ejecuta la instrucción de asignación asociada con esa instrucción `if`. La aplicación utiliza la clase `Console` para solicitar y leer dos líneas de texto del usuario, extrae los enteros de ese texto mediante el método `ToInt32` de la clase `Convert` y los almacena en las variables `numero1` y `numero2`. Después la aplicación compara los números y muestra los resultados de las comparaciones que son verdaderas.

La declaración de la clase `Comparacion` comienza en la línea 6

```
public class Comparacion
```

El método `Main` de la clase (líneas 9-39) comienza la ejecución de la aplicación.

Operadores estándar algebraicos de igualdad y relacionales	Operador de igualdad o relacional de C#	Operador de igualdad o relacional de C#	Significado de la condición C#
<i>Operadores de igualdad</i>			
=	==	x == y	x es igual a y
≠	!=	x != y	x no es igual a y
<i>Operadores relacionales</i>			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y

Figura 3.25 | Operadores de igualdad y relacionales.

Las líneas 11-12

```
int numero1; // declara el primer número a comparar
int numero2; // declara el segundo número a comparar
```

declaran las variables `int` que se utilizan para almacenar los valores introducidos por el usuario.

Las líneas 14-16

```
// pide al usuario y lee el primer número
Console.Write( "Escriba el primer entero: " );
numero1 = Convert.ToInt32( Console.ReadLine() );
```

piden al usuario que escriba el primer entero y reciben ese valor como entrada. El valor de entrada se almacena en la variable `numero1`.

```
1  // Fig. 3.26: Comparacion.cs
2  // Comparación de enteros mediante el uso de instrucciones if, operadores de
3  // igualdad y operadores relacionales.
4  using System;
5
6  public class Comparacion
7  {
8      // El método Main comienza la ejecución de la aplicación de C#
9      public static void Main( string[] args )
10     {
11         int numero1; // declara el primer número a comparar
12         int numero2; // declara el segundo número a comparar
13
14         //pide al usuario y lee el primer número
15         Console.Write( "Escriba el primer entero: " );
16         numero1 = Convert.ToInt32( Console.ReadLine() );
17
18         //pide al usuario y lee el segundo número
19         Console.Write( "Escriba el segundo entero: " );
20         numero2 = Convert.ToInt32( Console.ReadLine() );
21
22         if ( numero1 == numero2 )
23             Console.WriteLine( "{0} == {1}", numero1, numero2 );
24
25         if ( numero1 != numero2 )
26             Console.WriteLine( "{0} != {1}", numero1, numero2 );
27
28         if ( numero1 < numero2 )
29             Console.WriteLine( "{0} < {1}", numero1, numero2 );
30
31         if ( numero1 > numero2 )
32             Console.WriteLine( "{0} > {1}", numero1, numero2 );
33
34         if ( numero1 <= numero2 )
35             Console.WriteLine( "{0} <= {1}", numero1, numero2 );
36
37         if ( numero1 >= numero2 )
38             Console.WriteLine( "{0} >= {1}", numero1, numero2 );
39     } // fin del método Main
40 } // fin de la clase Comparacion
```

Figura 3.26 | Comparación de enteros mediante el uso de instrucciones `if`, operadores de igualdad y operadores relacionales. (Parte I de 2).

```

Escriba el primer entero: 42
Escriba el segundo entero: 42
42 == 42
42 <= 42
42 >= 42

```

```

Escriba el primer entero: 1000
Escriba el segundo entero: 2000
1000 != 2000
1000 < 2000
1000 <= 2000

```

```

Escriba el primer entero: 2000
Escriba el segundo entero: 1000
2000 != 1000
2000 > 1000
2000 >= 1000

```

Figura 3.26 | Comparación de enteros mediante el uso de instrucciones `if`, operadores de igualdad y operadores relacionales. (Parte 2 de 2).

Las líneas 18-20

```

// pide al usuario y lee el segundo número
Console.Write( "Escriba el segundo entero: " );
numero2 = Convert.ToInt32( Console.ReadLine() );

```

realizan la misma tarea, sólo que el valor de entrada se almacena en la variable `numero2`.

Las líneas 22-23

```

if ( numero1 == numero2 )
    Console.WriteLine( "{0} == {1}", numero1, numero2 );

```

comparan los valores de las variables `numero1` y `numero2` para determinar si son iguales. Una instrucción `if` siempre comienza con la palabra clave `if`, seguida de una condición entre paréntesis. Una instrucción `if` espera una instrucción en su cuerpo. La sangría de la instrucción del cuerpo que se muestra aquí no es requerida, pero aumenta la legibilidad del código al enfatizar que la instrucción de la línea 23 es parte de la instrucción `if` que comienza en la línea 22. La línea 23 se ejecuta sólo si los números almacenados en las variables `numero1` y `numero2` son iguales (es decir, que la condición sea verdadera). Las instrucciones `if` en las líneas 25-26, 28-29, 31-32, 34-35 y 37-38 comparan a `numero1` y `numero2` mediante los operadores `!=`, `<`, `>`, `<=` y `>=`, en forma respectiva. Si la condición en cualquiera de las instrucciones `if` es verdadera, se ejecuta la correspondiente instrucción del cuerpo.



Error común de programación 3.6

Olvidar los paréntesis izquierdo y/o derecho para la condición en una instrucción `if` es un error de sintaxis; los paréntesis son requeridos.



Error común de programación 3.7

Confundir el operador de igualdad, `==`, con el de asignación, `=`, puede provocar un error lógico o un error de sintaxis. El operador de igualdad debe leerse como “es igual a”, y el de asignación debe leerse como “obtiene” u “obtiene el valor de”. Para evitar confusión, algunas personas leen el operador de igualdad como “doble igual a” o “igual a igual a”.

**Error común de programación 3.8**

Si los operadores ==, !=, >= y <= contienen espacios entre sus símbolos, como en = =, ! =, > = y < =, en forma respectiva, se produce un error de sintaxis.

**Error común de programación 3.9**

Si se invierten los operadores !=, >= y <=, como en !=, => y =<, se produce un error de sintaxis.

**Buena práctica de programación 3.12**

Aplique sangría al cuerpo de una instrucción if para hacer que resalte y mejorar la legibilidad de la aplicación.

Observe que no hay punto y coma (;) al final de la primera línea de cada instrucción if. Dicho punto y coma produciría un error lógico en tiempo de ejecución. Por ejemplo,

```
if ( numero1 == numero2 ); // error lógico
    Console.WriteLine( "{0} == {1}", numero1, numero2 );
```

C# lo interpretaría en realidad como

```
if ( numero1 == numero2 )
    ; // instrucción vacía
    Console.WriteLine( "{0} == {1}", numero1, numero2 );
```

en donde el punto y coma en la línea por sí solo (a lo cual se le llama **instrucción vacía**) es la instrucción que se ejecutará si la condición en la instrucción if es verdadera. Cuando se ejecuta la instrucción vacía, no se realiza ninguna tarea en la aplicación. Así, la aplicación continúa con la instrucción de salida, que siempre se ejecuta sin importar que la condición sea verdadera o falsa, ya que no forma parte de la instrucción if.

**Error común de programación 3.10**

Colocar un punto y coma inmediatamente después del paréntesis derecho de la condición en una instrucción if es por lo general un error lógico.

Observe el uso del espacio en blanco en la figura 3.26. Recuerde que por lo general el compilador ignora los caracteres en blanco, como los tabuladores, nuevas líneas y espacios. Así, las instrucciones pueden dividirse en varias líneas y es posible utilizar espacios de acuerdo a sus preferencias, sin afectar el significado de una aplicación. Es incorrecto dividir identificadores, cadenas y operadores con varios caracteres (como >=). En teoría, las instrucciones deben mantenerse reducidas, pero esto no siempre es posible.

**Buena práctica de programación 3.13**

No coloque más de una instrucción por línea en una aplicación. Este formato mejora la legibilidad.

**Buena práctica de programación 3.14**

Una instrucción larga puede esparcirse en varias líneas. Si una sola instrucción debe dividirse en dos o más líneas, elija puntos de división que tengan sentido, como después de una coma en una lista separada por comas, o después de un operador en una expresión larga. Si se divide una instrucción en dos o más líneas, aplique sangría a todas las líneas subsiguientes hasta el final de la instrucción.

La figura 3.27 muestra la precedencia de los operadores que presentamos en este capítulo. Los operadores se muestran de arriba hacia abajo, en orden decreciente de precedencia. Todos estos operadores, excepto el operador de asignación (=), se asocian de izquierda a derecha. La suma es asociativa a la izquierda, por lo que una expresión como $x + y + z$ se evalúa como si se hubiera escrito así: $(x + y) + z$. El operador de asignación (=) se asocia de derecha a izquierda, por lo que una expresión como $x = y = 0$ se evalúa como si se hubiera escrito así: $x = (y = 0)$, lo cual, como pronto veremos, asigna primero el valor 0 a la variable y y después asigna el resultado de esa asignación (0) a x.

Operadores	Asociatividad	Tipo
* / %	izquierda a derecha	multiplicativa
+ -	izquierda a derecha	aditiva
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
=	derecha a izquierda	asignación

Figura 3.27 | Precedencia y asociatividad de las operaciones descritas.



Buena práctica de programación 3.15

Consulte la tabla de precedencia de operadores (en el apéndice A podrá ver la tabla completa) cuando escriba expresiones que contengan muchos operadores. Confirme que las operaciones en la expresión se realicen en el orden que espera. Si no está seguro acerca del orden de evaluación en una expresión compleja, utilice paréntesis para forzar el orden, como lo haría con las expresiones algebraicas. Tenga en cuenta que algunos operadores, como el de asignación (=), asocian de derecha a izquierda, en vez de asociar de izquierda a derecha.

3.10 (Opcional) Caso de estudio de ingeniería de software: análisis del documento de requerimientos del ATM

Ahora empezaremos nuestro caso de estudio opcional de diseño e implementación orientados a objetos. Las secciones tituladas Caso de estudio de ingeniería de software al final de éste y los siguientes capítulos lo llevarán a través de la orientación a objetos. Desarrollaremos software para un sistema simple de cajero automático (ATM), con lo cual le proporcionaremos una experiencia de diseño e implementación orientados a objetos concisa, minuciosa y completa. En los capítulos 4-9 y 11 llevaremos a cabo los diversos pasos de un proceso de diseño orientado a objetos (DOO) mediante el uso del UML, al mismo tiempo que relacionaremos estos pasos con los conceptos orientados a objetos que veremos en los capítulos. En el apéndice J implementaremos el ATM mediante el uso de las técnicas de la programación orientada a objetos (POO) en C# y presentaremos la solución completa al caso de estudio. Éste no es un ejercicio, sino una experiencia de aprendizaje de principio a fin, la cual concluiremos con un paseo detallado por el código completo de C# que implementa nuestro diseño. Con esto empezará a familiarizarse con los tipos de problemas sustanciales que se encuentran en la industria, junto con sus soluciones.

Empezaremos nuestro proceso de diseño mediante la presentación de un **documento de requerimientos**, el cual especifica el propósito general del sistema ATM y *qué* es lo que debe hacer. A lo largo del caso de estudio nos referiremos al documento de requerimientos para determinar con precisión qué funcionalidad debe incluir el sistema.

Documento de requerimientos

Un banco local pequeño pretende instalar un nuevo cajero automático (ATM) para permitir que los usuarios (es decir, clientes del banco) realicen transacciones financieras básicas (figura 3.28). Por cuestión de simpleza, cada usuario sólo puede tener una cuenta en el banco. Los usuarios del ATM deben poder ver su saldo, retirar efectivo (es decir, sacar dinero de una cuenta) y depositar fondos (es decir, colocar dinero en una cuenta).

La interfaz de usuario del cajero automático contiene los siguientes componentes de hardware:

- una pantalla que muestra mensajes al usuario.
- un teclado numérico que recibe entrada numérica del usuario.
- un dispensador de efectivo que entrega efectivo al usuario.
- una ranura para depósitos que recibe sobres para depósitos del usuario.

El dispensador de efectivo comienza cada día cargado con 500 billetes de \$20. [Nota: debido al alcance limitado de este caso de estudio, ciertos elementos del ATM que se describen aquí simplifican varios aspectos de uno real. Por ejemplo, con frecuencia un ATM contiene un dispositivo que lee el número de cuenta del

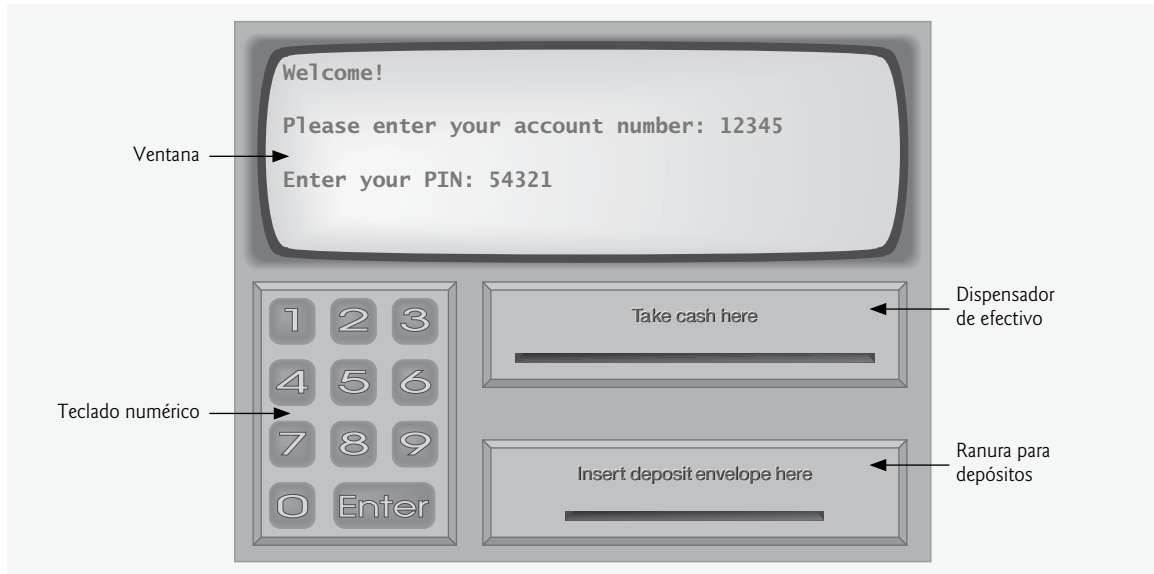


Figura 3.28 | Interfaz de usuario del cajero automático.

usuario de una tarjeta para ATM, mientras que este pide al usuario que escriba su número de cuenta en el teclado (el cual usted simulará con el teclado de su computadora personal). Además, un ATM real, por lo general, imprime un recibo de papel al final de una sesión, y toda la salida de este ATM aparece en la pantalla.]

El banco desea que usted desarrolle software para realizar las transacciones financieras que inicien los clientes del banco a través del ATM. El banco integrará posteriormente el software con el hardware del ATM. El software debe simular la funcionalidad de los dispositivos de hardware (por ejemplo: dispensador de efectivo, ranura para depósito) mediante componentes de software, pero no necesita preocuparse por cómo realizan estos dispositivos su trabajo. El hardware del ATM no se ha desarrollado aún, por lo que en vez de que usted escriba su software para que se ejecute en el ATM, deberá desarrollar una primera versión del software para que se ejecute en una computadora personal. Esta versión debe utilizar el monitor de la computadora para simular la pantalla del ATM y el teclado de la computadora para simular el teclado numérico del ATM.

Una sesión con el ATM consiste en la autenticación de un usuario (es decir, proporcionar la identidad del usuario) con base en un número de cuenta y un número de identificación personal (NIP), seguida de la creación y la ejecución de transacciones financieras. Para autenticar un usuario y realizar transacciones, el ATM debe interactuar con la base de datos de información sobre las cuentas del banco. [Nota: una base de datos es una colección organizada de datos almacenados en una computadora.] Para cada cuenta de banco, la base de datos almacena un número de cuenta, un NIP y un balance que indica la cantidad de dinero en la cuenta. [Nota: el banco planea construir sólo un ATM, por lo que no necesitamos preocuparnos porque varios ATMs accedan a la base de datos al mismo tiempo. Lo que es más, supongamos que el banco no realizará modificaciones en la información que hay en la base de datos mientras un usuario accede al ATM. Además, cualquier sistema comercial como un ATM se topa con cuestiones de seguridad con una complejidad razonable, las cuales van más allá del alcance de un curso de programación de primer o segundo semestre. No obstante, para simplificar nuestro ejemplo supongamos que el banco confía en el ATM para que acceda a la información en la base de datos y la manipule sin necesidad de medidas de seguridad considerables.]

Al acercarse al ATM, el usuario deberá experimentar la siguiente secuencia de eventos (vea la figura 3.28):

1. La pantalla muestra un mensaje de bienvenida y pide al usuario que introduzca un número de cuenta.
2. El usuario introduce un número de cuenta de cinco dígitos, mediante el uso del teclado.
3. Para fines de autenticación, la pantalla pide al usuario que introduzca su NIP (número de identificación personal) asociado con el número de cuenta especificado.

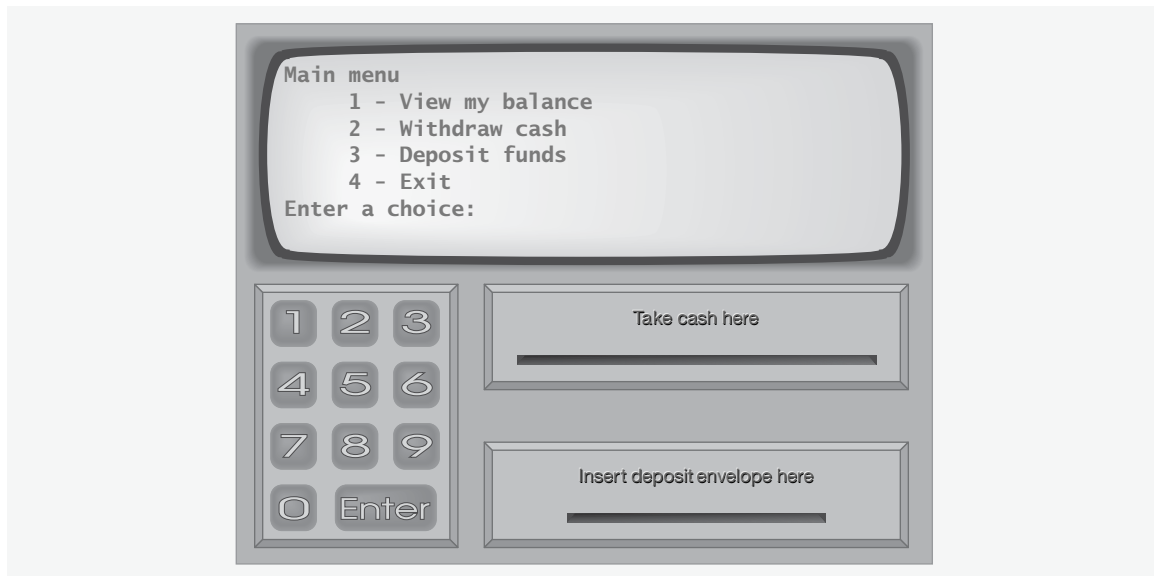


Figura 3.29 | Menú principal del ATM.

4. El usuario introduce un NIP de cinco dígitos mediante el teclado numérico.
5. Si el usuario introduce un número de cuenta válido y el NIP correcto para esa cuenta, la pantalla muestra el menú principal (figura 3.29). Si el usuario introduce un número de cuenta inválido o un NIP incorrecto, la pantalla muestra un mensaje apropiado y después el ATM regresa al *paso 1* para reiniciar el proceso de autenticación.

Una vez que el ATM autentique al usuario, el menú principal (figura 3.29) mostrará una opción numerada para cada uno de los tres tipos de transacciones: solicitud de saldo (opción 1), retiro (opción 2) y depósito (opción 3). El menú principal también mostrará una opción que permite al usuario salir del sistema (opción 4). Después, el usuario elegirá si desea realizar una transacción (oprimiendo 1, 2 o 3) o salir del sistema (4). Si el usuario introduce una opción inválida, la pantalla mostrará un mensaje de error y volverá a mostrar el menú principal.

Si el usuario oprime 1 para solicitar su saldo, la pantalla mostrará el saldo de esa cuenta bancaria. Para ello, el ATM deberá obtener el saldo de la base de datos del banco.

Las siguientes acciones se realizan cuando el usuario elige la opción 2 para hacer un retiro:

1. La pantalla muestra un menú (vea la figura 3.30) que contenga montos de retiro estándar: \$20 (opción 1), \$40 (opción 2), \$60 (opción 3), \$100 (opción 4) y \$200 (opción 5). El menú también contiene la opción 6, que permite al usuario cancelar la transacción.
2. El usuario introduce la selección del menú (1-6) mediante el teclado numérico.
3. Si el monto elegido a retirar es mayor que el saldo de la cuenta del usuario, la pantalla muestra un mensaje indicando esta situación y pide al usuario que seleccione un monto más pequeño. Entonces el ATM regresa al *paso 1*. Si el monto elegido a retirar es menor o igual que el saldo de la cuenta del usuario (es decir, un monto de retiro aceptable), el ATM procede al *paso 4*. Si el usuario opta por cancelar la transacción (opción 6), el ATM muestra el menú principal (figura 3.29) y espera la entrada del usuario.
4. Si el dispensador contiene suficiente efectivo para satisfacer la solicitud, el ATM procede al *paso 5*. En caso contrario, la pantalla muestra un mensaje indicando el problema y pide al usuario que seleccione un monto de retiro más pequeño. Después el ATM regresa al *paso 1*.
5. El ATM carga (es decir, resta) el monto de retiro al saldo de la cuenta del usuario en la base de datos del banco.

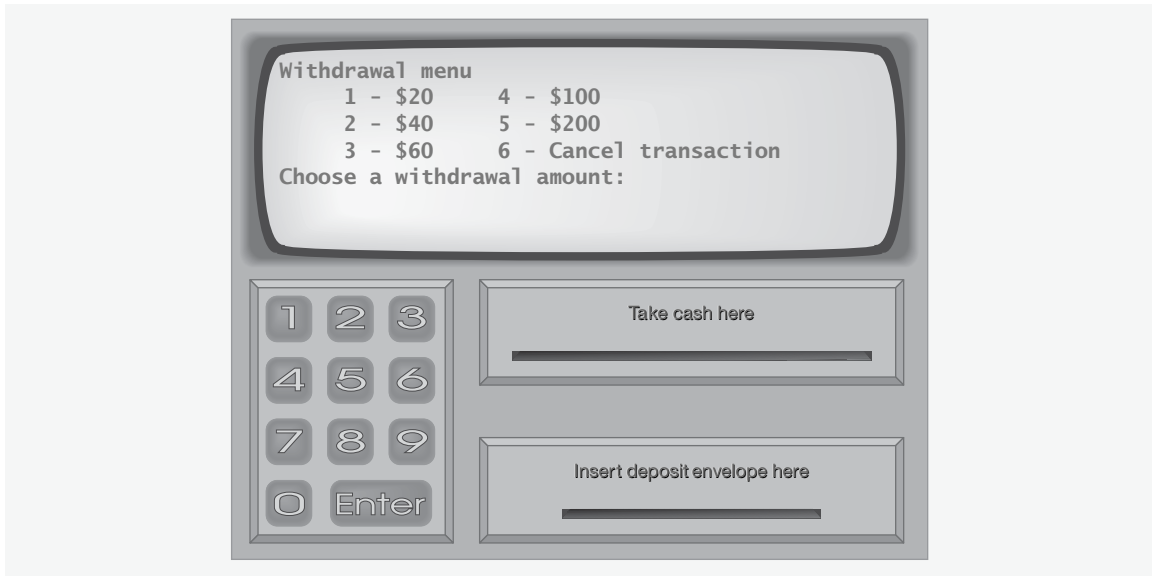


Figura 3.30 | Menú de retiro del ATM.

6. El dispensador de efectivo entrega el monto deseado de dinero al usuario.
7. La pantalla muestra un mensaje para recordar al usuario que tome el dinero.

Las siguientes acciones se realizan cuando el usuario elige la opción 3 (del menú principal) para hacer un depósito:

1. La pantalla pide al usuario que introduzca un monto de depósito o que escriba 0 (cero) para cancelar la transacción.
2. El usuario introduce un monto de depósito o 0 mediante el teclado numérico. [*Nota:* estos teclados no contienen un punto decimal o signo de moneda, por lo que el usuario no puede escribir una cantidad real (por ejemplo: \$147.25), sino que debe escribir un monto de depósito en forma de número de centavos (por ejemplo: 14725). Después, el ATM divide este número entre 100 para obtener un número que represente un monto ya sea en pesos o en dólares (por ejemplo, $14725 \div 100 = 147.25$).]
3. Si el usuario especifica un monto a depositar, el ATM procede al *paso 4*. Si elige cancelar la transacción (escribiendo 0), el ATM muestra el menú principal (figura 3.29) y espera la entrada del usuario.
4. La pantalla muestra un mensaje indicando al usuario que introduzca un sobre de depósito en la ranura para depósitos.
5. Si la ranura de depósitos recibe un sobre dentro de un plazo no mayor a 2 minutos, el ATM abona (es decir, suma) el monto de depósito al saldo de la cuenta del usuario en la base de datos del banco. [*Nota:* este dinero no está disponible de inmediato para retirarse. El banco primero debe verificar el monto de efectivo en el sobre y cualquier cheque que éste contenga debe validarse (es decir, el dinero debe transferirse de la cuenta del emisor del cheque a la cuenta del beneficiario). Cuando ocurra uno de estos eventos, el banco actualizará de manera apropiada el saldo del usuario que está almacenado en su base de datos. Esto ocurre de manera independiente al sistema ATM]. Si la ranura de depósitos no recibe un sobre dentro de un plazo no mayor a dos minutos, la pantalla muestra un mensaje indicando que el sistema canceló la transacción debido a la inactividad. Después, el ATM muestra el menú principal y espera la entrada del usuario.

Una vez que el sistema ejecuta una transacción en forma exitosa, debe volver a mostrar el menú principal (figura 3.29) para que el usuario pueda realizar transacciones adicionales. Si el usuario elige salir del sistema

(opción 4), la pantalla debe mostrar un mensaje de agradecimiento y después el mensaje de bienvenida para el siguiente usuario.

Análisis del sistema de ATM

En la declaración anterior se presentó un documento de requerimientos simplificado. Por lo general, dicho documento es el resultado de un proceso detallado de **recopilación de requerimientos**, el cual podría incluir entrevistas con usuarios potenciales del sistema y especialistas en campos relacionados con el mismo. Por ejemplo, un analista de sistemas que se contrate para preparar un documento de requerimientos para software bancario (es decir, el sistema ATM que describimos aquí) podría entrevistar a personas que hayan utilizado ATMs y a expertos financieros para obtener una mejor comprensión de *qué* es lo que debe hacer el software. El analista utilizaría la información recopilada para compilar una lista de **requerimientos del sistema** para guiar a los diseñadores de sistemas.

El proceso de recopilación de requerimientos es una tarea clave de la primera etapa del ciclo de vida del software. El **ciclo de vida del software** especifica las etapas a través de las cuales el software evoluciona desde el tiempo en que fue concebido hasta el tiempo en que se retira de su uso. Por lo general, estas etapas incluyen el análisis, diseño, implementación, prueba y depuración, despliegue, mantenimiento y retiro. Existen varios modelos de ciclo de vida del software, cada uno con sus propias preferencias y especificaciones con respecto a cuándo y qué tan a menudo deben llevar a cabo los ingenieros de software las diversas etapas. Los **modelos de cascada** realizan cada etapa una vez en sucesión, mientras que los **modelos iterativos** pueden repetir una o más etapas varias veces a lo largo del ciclo de vida de un producto.

La etapa de análisis del ciclo de vida del software se enfoca en definir con precisión el problema a resolver. Al diseñar cualquier sistema, es indudable que uno debe *resolver el problema de la manera correcta*, pero de igual manera uno debe *resolver el problema correcto*. Los analistas de sistemas recolectan los requerimientos que indican el problema específico a resolver. Nuestro documento de requerimientos describe nuestro sistema ATM simple con el suficiente detalle como para que usted no necesite pasar por una etapa de análisis exhaustiva; ya lo hicimos por usted.

Para capturar lo que debe hacer un sistema propuesto, los desarrolladores emplean a menudo una técnica conocida como **modelado de caso-uso**. Este proceso identifica los **casos de uso** del sistema, cada uno de los cuales representa una capacidad distinta que el sistema provee a sus clientes. Por ejemplo, es común que los ATMs tengan varios casos de uso, como “Ver saldo de cuenta”, “Retirar efectivo”, “Depositar fondos”, “Transferir fondos entre cuentas” y “Comprar estampas postales”. El sistema ATM simplificado que construiremos en este caso de estudio requiere sólo los tres primeros casos de uso (figura 3.31).

Cada uno de los casos de uso describe un escenario común en el cual el usuario utiliza el sistema. Usted ya leyó las descripciones de los casos de uso del sistema ATM en el documento de requerimientos; las listas de pasos requeridos para realizar cada tipo de transacción (es decir, solicitud de saldo, retiro y depósito) describen los tres casos de uso de nuestro ATM: “Ver saldo de cuenta”, “Retirar efectivo” y “Depositar fondos”.

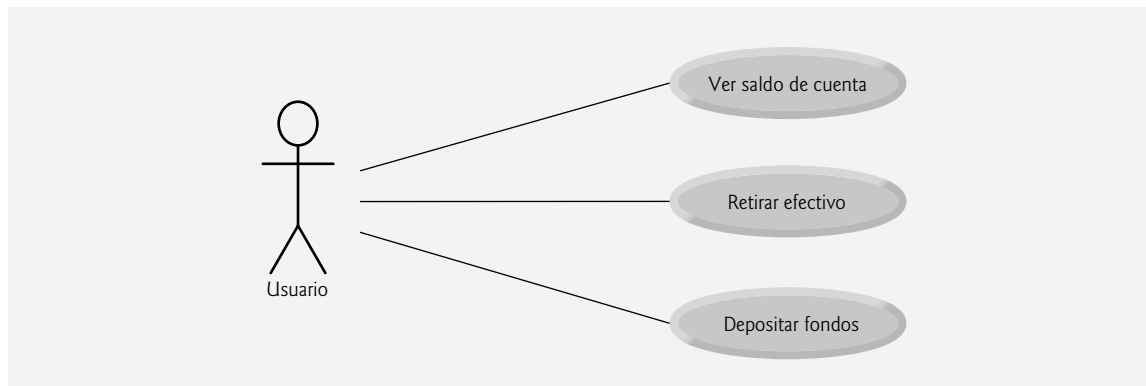


Figura 3.31 | Diagrama de caso-uso para el sistema ATM, desde la perspectiva del usuario.

Diagramas de caso-uso

Ahora presentaremos el primero de varios diagramas de UML en nuestro caso de estudio del ATM. Crearemos un **diagrama de caso-uso** para modelar las interacciones entre los clientes de un sistema (en este caso de estudio, los clientes del banco) y el sistema. El objetivo es mostrar los tipos de interacciones que tienen los usuarios con un sistema sin proveer los detalles; éstos se mostrarán en otros diagramas de UML (que presentaremos a lo largo del caso de estudio). A menudo, los diagramas de caso-uso se acompañan de texto informal que describe los casos de uso con más detalle; como el texto que aparece en el documento de requerimientos. Los diagramas de caso-uso se producen durante la etapa de análisis del ciclo de vida del software. En sistemas más grandes, los diagramas de caso-uso son herramientas simples pero indispensables, que ayudan a los diseñadores de sistemas a enfocarse en satisfacer las necesidades de los usuarios.

La figura 3.31 muestra el diagrama de caso-uso para nuestro sistema ATM. La figura humana representa a un **actor**, quien define los roles que desempeña una entidad externa (como una persona u otro sistema) cuando interactúa con el sistema. Para nuestro cajero automático, el actor es un Usuario que puede ver el saldo de una cuenta, retirar efectivo y depositar fondos mediante el uso del ATM. El Usuario no es una persona real, sino que constituye los roles que puede desempeñar una persona real (al desempeñar el papel de un Usuario) mientras interactúa con el ATM. Hay que tener en cuenta que un diagrama de caso-uso puede incluir varios actores. Por ejemplo, el diagrama de caso-uso para un sistema ATM de un banco real podría incluir también un actor llamado Administrador, que rellene el dispensador de efectivo a diario.

Para identificar al actor en nuestro sistema debemos examinar el documento de requerimientos, el cual dice que “los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo y depositar fondos”. El actor en cada uno de estos tres casos de uso es simplemente el Usuario que interactúa con el ATM. Una entidad externa (una persona real) desempeña el papel del Usuario para realizar transacciones financieras. La figura 3.31 muestra un actor, cuyo nombre (Usuario) aparece debajo del actor en el diagrama. UML modela cada caso de uso como un óvalo conectado a un actor con una línea sólida.

Los ingenieros de software (específicamente, los diseñadores de sistemas) deben analizar el documento de requerimientos o un conjunto de casos de uso, y diseñar el sistema antes de que los programadores lo implementen en un lenguaje de programación específico. Durante la etapa de análisis, los diseñadores de sistemas se enfocan en comprender el documento de requerimientos para producir una especificación de alto nivel que describa *qué* es lo que el sistema debe hacer. El resultado de la etapa de diseño (una **especificación de diseño**) debe especificar *cómo* debe construirse el sistema para satisfacer estos requerimientos. En las siguientes secciones del Caso de estudio de ingeniería de software, llevaremos a cabo los pasos de un proceso simple de DOO con el sistema ATM para producir una especificación de diseño que contenga una colección de diagramas de UML y texto de apoyo. Recuerde que UML está diseñado para utilizarse con cualquier proceso de DOO. Existen muchos de esos procesos, de los cuales el más conocido es Rational Unified Process™ (RUP), desarrollado por Rational Software Corporation (ahora una división de IBM). RUP es un proceso robusto para diseñar aplicaciones a nivel industrial. Para este caso de estudio, presentaremos un proceso de diseño simplificado.

Diseño del sistema ATM

Ahora comenzaremos la etapa de diseño de nuestro sistema ATM. Un **sistema** es un conjunto de componentes que interactúan para resolver un problema. Por ejemplo, para realizar sus tareas designadas, nuestro sistema ATM tiene una interfaz de usuario (figura 3.28), contiene software para ejecutar transacciones financieras e interactúa con una base de datos de información de cuentas bancarias. La **estructura del sistema** describe los objetos del sistema y sus interrelaciones. El **comportamiento del sistema** describe la manera en que cambia el sistema a medida que sus objetos interactúan entre sí. Todo sistema tiene tanto estructura como comportamiento; los diseñadores deben especificar ambos. Existen diversos tipos de estructuras y comportamientos de un sistema. Por ejemplo, las interacciones entre los objetos en el sistema son distintas a las interacciones entre el usuario y el sistema, pero aun así ambas constituyen una porción del comportamiento del sistema.

UML 2 especifica 13 tipos de diagramas para documentar los modelos de un sistema. Cada tipo de diagrama modela una característica distinta de la estructura o del comportamiento de un sistema; seis tipos de diagramas se relacionan con la estructura del sistema; los siete restantes se relacionan con su comportamiento. Aquí listaremos sólo los seis tipos de diagramas que utilizaremos en nuestro caso de estudio, uno de los cuales (el diagrama de clases) modela la estructura del sistema; los otros cinco modelan el comportamiento. En el apéndice K, UML 2: Tipos de diagramas adicionales, veremos las generalidades sobre los siete tipos restantes de diagramas de UML.

1. Los **diagramas de caso-uso**, como el de la figura 3.31, modelan las interacciones entre un sistema y sus entidades externas (actores) en términos de casos de uso (capacidades del sistema, como “Ver saldo de cuenta”, “Retirar efectivo” y “Depositar fondos”).
2. Los **diagramas de clases**, que estudiará en la sección 4.11, modelan las clases o “bloques de construcción” que se utilizan en un sistema. Cada sustantivo u “objeto” que se describe en el documento de requerimientos es candidato para ser una clase en el sistema (por ejemplo, “cuenta”, “teclado”). Los diagramas de clases nos ayudan a especificar las relaciones estructurales entre las partes del sistema. Por ejemplo, el diagrama de clases del sistema ATM especificará, entre otras cosas, que el ATM está compuesto físicamente de una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos.
3. Los **diagramas de máquina de estado**, que estudiará en la sección 6.9, modelan las formas en que un objeto cambia de estado. El **estado** de un objeto se indica mediante los valores de todos los atributos del objeto, en un momento dado. Cuando un objeto cambia de estado, en consecuencia puede comportarse de manera distinta en el sistema. Por ejemplo, después de validar el NIP de un usuario, el ATM cambia del estado “usuario no autenticado” al estado “usuario autenticado”, punto en el cual el ATM permite al usuario realizar transacciones financieras (por ejemplo, ver el saldo de su cuenta, retirar efectivo, depositar fondos).
4. Los **diagramas de actividad**, que también estudiará en la sección 6.9, modelan la **actividad** de un objeto: el flujo de trabajo (secuencia de eventos) del objeto durante la ejecución del programa. Un diagrama de actividad modela las acciones que realiza el objeto y especifica el orden en el cual desempeña estas acciones. Por ejemplo, un diagrama de actividad muestra que el ATM debe obtener el saldo de la cuenta del usuario (de la base de datos de información de las cuentas del banco) antes de que la pantalla pueda mostrar el saldo al usuario.
5. Los **diagramas de comunicación** (llamados diagramas de colaboración en versiones anteriores de UML) modelan las interacciones entre los objetos en un sistema, con un énfasis acerca de *qué* interacciones ocurren. En la sección 8.14 aprenderá que estos diagramas muestran cuáles objetos deben interactuar para realizar una transacción en el ATM. Por ejemplo, el ATM debe comunicarse con la base de datos de información de las cuentas del banco para obtener el saldo de una cuenta.
6. Los **diagramas de secuencia** modelan también las interacciones entre los objetos en un sistema, pero a diferencia de los diagramas de comunicación, enfatizan *cuándo* ocurren las interacciones. En la sección 8.14 aprenderá que estos diagramas ayudan a mostrar el orden en el que ocurren las interacciones al ejecutar una transacción financiera. Por ejemplo, la pantalla pide al usuario que escriba un monto de retiro antes de dispensar el efectivo.

En la sección 4.11 seguiremos diseñando nuestro sistema ATM; ahí identificaremos las clases del documento de requerimientos. Para lograr esto, vamos a extraer sustantivos clave y frases nominales del documento de requerimientos. Mediante el uso de estas clases, desarrollaremos nuestro primer borrador del diagrama de clases que modelará la estructura de nuestro sistema ATM.

Recursos en Internet y Web

Las siguientes URLs proporcionan información sobre el diseño orientado a objetos con el UML.

www-306.ibm.com/software/rational/uml/

Lista preguntas frecuentes acerca del UML, proporcionado por IBM Rational.

www.douglass.co.uk/documents/softdocwiz.com.UML.htm

Vínculos al Diccionario del Lenguaje unificado de modelado, el cual define todos los términos utilizados en el UML.

www.agilemodeling.com/essays/umlDiagrams.htm

Proporciona descripciones detalladas y tutoriales acerca de cada uno de los 13 tipos de diagramas de UML 2.

www-306.ibm.com/software/rational/offerings/design.html

IBM proporciona información acerca del software de Rational disponible para el diseño de sistemas, y descargas de versiones de prueba de 30 días de varios productos, como IBM Rational Rose® XDE (Entorno de desarrollo extendido) Developer.

www.embarcadero.com/products/describe/index.html

Proporciona una licencia de prueba de 15 días para la herramienta de modelado de UML Describe™ de Embarcadero Technologies®.

www.borland.com/together/index.html

Proporciona una licencia gratuita de 30 días para descargar una versión de Borland® Together® Control-Center™; una herramienta de desarrollo de software que soporta el UML.

www.ilogix.com/rhapsody/rhapsody.cfm

Proporciona una licencia gratuita de 30 días para descargar una versión de prueba de I-Logix Rhapsody®: un entorno de desarrollo controlado por modelos y basado en UML 2.

argouml.tigris.org

Contiene información y descargas para ArgoUML, una herramienta gratuita de software libre de UML.

www.objectsbydesign.com/books/booklist.html

Provee una lista de libros acerca de UML y el diseño orientado a objetos.

www.objectsbydesign.com/tools/umltools_byCompany.html

Provee una lista de herramientas de software que utilizan UML, como IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody y Gentleware Poseidon para UML.

www.ootips.org/ood-principles.html

Proporciona respuestas a la pregunta “¿Qué se requiere para tener un buen diseño orientado a objetos?”

www.cetus-links.org/oo_uml.html

Introduce el UML y proporciona vínculos a numerosos recursos sobre UML.

Lecturas recomendadas

Los siguientes libros proporcionan información acerca del diseño orientado a objetos con el UML.

Ambler, S. *The Elements of the UML 2.0 Style*. Nueva York: Cambridge University Press, 2005.

Booch, G. *Object-Oriented Analysis and Design with Applications, Tercera edición*. Boston: Addison-Wesley, 2004.

Eriksson, H. *et al. UML 2 Toolkit*. Nueva York: John Wiley, 2003.

Kruchten, P. *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley, 2004.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Segunda edición. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. *UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions*. Nueva York: John Wiley, 2004.

Rosenberg, D. y K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Rumbaugh, J., I. Jacobson y G. Booch. *The Complete UML Training Course*. Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

3.1 Suponga que habilitamos a un usuario de nuestro sistema ATM para transferir dinero entre dos cuentas bancarias. Modifique el diagrama de caso-uso de la figura 3.31 para reflejar este cambio.

3.2 Los _____ modelan las interacciones entre los objetos en un sistema, con énfasis acerca de *cuándo* ocurren estas interacciones.

- a) Diagramas de clases
- b) Diagramas de secuencia
- c) Diagramas de comunicación
- d) Diagramas de actividad

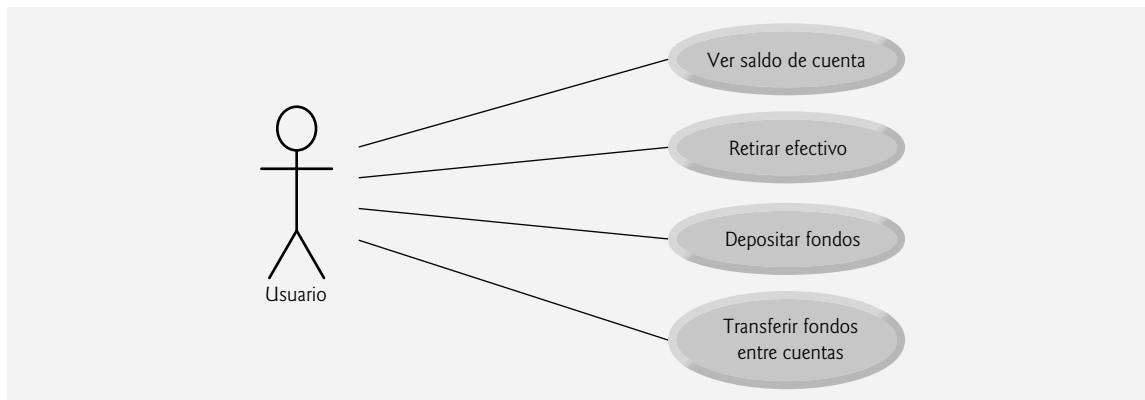


Figura 3.32 | Diagrama de caso-usuario para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre varias cuentas.

- 3.3** ¿Cuál de las siguientes opciones lista las etapas de un ciclo de vida de software común en orden secuencial?
- a) diseño, análisis, implementación, prueba
 - b) diseño, análisis, prueba, implementación
 - c) análisis, diseño, prueba, implementación
 - d) análisis, diseño, implementación, prueba

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

3.1 La figura 3.32 contiene un diagrama de caso-usuario para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre cuentas.

3.2 b.

3.3 d.

3.11 Conclusión

En este capítulo aprendió muchas características importantes de C#, incluyendo: mostrar datos en la pantalla en un símbolo del sistema, introducir datos desde el teclado, realizar cálculos y tomar decisiones. Las aplicaciones que presentamos aquí lo introdujeron a los conceptos básicos de programación. Como verá en el capítulo 4, las aplicaciones de C# por lo general contienen sólo unas cuantas líneas de código en el método `Main`; estas instrucciones por lo común crean los objetos que realizan el trabajo de la aplicación. En el capítulo 4 aprenderá a implementar sus propias clases y a utilizar objetos de esas clases en aplicaciones.

4

Introducción a las clases y los objetos

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Qué son las clases, los objetos, los métodos y las variables de instancia.
- Declarar una clase y utilizarla para crear un objeto.
- Implementar los comportamientos de una clase como métodos.
- Cómo implementar los atributos de una clase como variables de instancia y propiedades.
- Cómo llamar a los métodos de un objeto para que realicen sus tareas.
- Las diferencias entre las variables de instancia de una clase y las variables locales de un método.
- Utilizar un constructor para asegurar que los datos de un objeto se inicialicen cuando al crear el objeto.
- Las diferencias entre tipos por valor y tipos por referencia.

*Nada puede tener valor sin
ser un objeto de utilidad.*

—Karl Marx

*Sus sirvientes públicos
le sirven bien.*

—Adlai E. Stevenson

*Saber cómo responder
a alguien que habla,
Contestar a alguien que
envía un mensaje.*

—Amenemope

*Usted verá algo nuevo.
Dos cosas. Y las llamo:
Cosa Uno y Cosa Dos.*

—Dr. Theodor Seuss Geisel

- 4.1 Introducción
- 4.2 Clases, objetos, métodos, propiedades y variables de instancia
- 4.3 Declaración de una clase con un método e instanciamiento del objeto de una clase
- 4.4 Declaración de un método con un parámetro
- 4.5 Variables de instancia y propiedades
- 4.6 Diagrama de clases de UML con una propiedad
- 4.7 Ingeniería de software con propiedades y los descriptores de acceso `set` y `get`
- 4.8 Comparación entre tipos por valor y tipos por referencia
- 4.9 Inicialización de objetos con constructores
- 4.10 Los números de punto flotante y el tipo `decimal`
- 4.11 (Opcional) Caso de estudio de ingeniería de software: identificación de las clases en el documento de requerimientos del ATM
- 4.12 Conclusión

4.1 Introducción

En la sección 1.9 presentamos la terminología básica y los conceptos acerca de la programación orientada a objetos. En el capítulo 3 comenzó a utilizar esos conceptos para crear aplicaciones simples que mostraran mensajes al usuario, que obtuvieran información de él, realizaran cálculos y tomaran decisiones. Una característica común de todas las aplicaciones en el capítulo 3 fue que todas las instrucciones que realizaban tareas se encontraban en el método `Main`. Por lo general, las aplicaciones que usted desarrollará en este libro constarán de dos o más clases, y cada una contendrá uno o más métodos. Si se convierte en parte de un equipo de desarrollo en la industria, podría trabajar en aplicaciones que contengan cientos, o incluso hasta miles de clases. En este capítulo presentaremos un marco de trabajo simple para organizar las aplicaciones orientadas a objetos en C#.

Primero explicaremos el concepto de las clases mediante el uso de un ejemplo real. Después presentaremos cinco aplicaciones completas para demostrarle cómo crear y utilizar sus propias clases. Nuestro primer paso es comenzar el caso de estudio acerca de cómo desarrollar una clase tipo libro de calificaciones, que los instructores pueden utilizar para mantener las calificaciones de las pruebas de sus estudiantes. Durante los siguientes capítulos ampliaremos este caso de estudio y culminaremos con la versión que se presenta en el capítulo 8, Arreglos. El último ejemplo en este capítulo introduce el tipo `decimal` y lo utiliza para declarar cantidades monetarias dentro del contexto de una clase tipo cuenta bancaria, la cual mantiene el saldo de un cliente.

4.2 Clases, objetos, métodos, propiedades y variables de instancia

Comenzaremos con una analogía simple para ayudarle a comprender el concepto de las clases y su contenido. Suponga que desea conducir un auto y para hacer que aumente su velocidad debe presionar el pedal del acelerador. ¿Qué debe ocurrir antes de que pueda hacer esto? Bueno, antes de poder conducir un auto, alguien tiene que diseñarlo. Por lo general un auto empieza en forma de dibujos de ingeniería, similares a los planos de construcción que se utilizan para diseñar una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador, para que el auto aumente su velocidad. El pedal “oculta” los complejos mecanismos que se encargan de que el auto aumente su velocidad, de igual forma que el pedal del freno “oculta” los mecanismos que disminuyen la velocidad del auto y el volante “oculta” los mecanismos que hacen que el auto dé vuelta. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores puedan conducir un auto con facilidad.

Desafortunadamente, no puede conducir los dibujos de ingeniería de un auto. Antes de poder conducir un auto, éste debe construirse a partir de los dibujos de ingeniería que lo describen. Un auto completo tendrá un pedal acelerador verdadero para hacer que aumente su velocidad, pero aun así no es suficiente; el auto no acelerará por su propia cuenta, así que el conductor debe oprimir el pedal del acelerador.

Ahora utilizaremos nuestro ejemplo del auto para introducir los conceptos clave de programación de esta sección. Para realizar una tarea en una aplicación se requiere un método. El *método* describe los mecanismos que

se encargan de realizar sus tareas; y oculta al usuario las tareas complejas que realiza, de la misma forma que el pedal del acelerador de un auto oculta al conductor los complejos mecanismos para hacer que el auto vaya más rápido. En C# empezamos por crear una unidad de aplicación llamada **clase** para alojar (entre otras cosas) a un método, así como los dibujos de ingeniería de un auto alojan (entre otras cosas) el diseño del pedal del acelerador. En una clase se proporcionan uno o más métodos, que están diseñados para realizar las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener un método para depositar dinero, otro para retirar dinero y un tercer método para solicitar el saldo actual.

Así como no podemos conducir un dibujo de ingeniería de un auto, tampoco podemos “conducir” una clase. De la misma forma que alguien tiene que construir un auto a partir de sus dibujos de ingeniería para poder conducirlo, también debemos construir un **objeto** de una clase para poder hacer que una aplicación realice las tareas descritas por la clase. Ésta es una de las razones por las cuales C# se conoce como un lenguaje de programación orientado a objetos.

Cuando usted conduce un auto, si oprime el pedal del acelerador se envía un mensaje al auto para que realice una tarea: hacer que el auto vaya más rápido. De manera similar, se envían **mensajes** a un objeto; a cada mensaje se le conoce como la **llamada a un método** e indica a un método del objeto que realice su tarea.

Hasta ahora hemos utilizado la analogía del auto para introducir las clases, los objetos y los métodos. Además de las capacidades con las que cuenta un auto, también tiene muchos **atributos** como su color, el número de puertas, la cantidad de gasolina en su tanque, su velocidad actual y el total de kilómetros recorridos (es decir, la lectura de su odómetro). Al igual que las capacidades del auto, estos atributos se representan como parte del diseño del auto en sus diagramas de ingeniería. Cuando usted conduce un auto, estos atributos siempre están asociados con él. Cada auto mantiene sus propios atributos. Por ejemplo, cada auto sabe cuánta gasolina tiene en su propio tanque, pero no cuánta hay en los tanques de otros autos. De manera similar, un objeto tiene atributos que lleva consigo cuando se utiliza en una aplicación. Estos atributos se especifican como parte de la clase del objeto. Por ejemplo, un objeto cuenta bancaria tiene un atributo llamado saldo, que representa la cantidad de dinero en la cuenta. Cada objeto cuenta bancaria conoce el saldo en la cuenta que representa, pero no los saldos de las otras cuentas en el banco. Los atributos se especifican mediante las **variables de instancia** de la clase.

Hay que tener en cuenta que no es necesario tener acceso directo a estos atributos. El fabricante de autos no desea que los conductores desarmen el motor del auto para observar la cantidad de gasolina en su tanque. En vez de ello, el conductor puede revisar el medidor en el tablero. El banco no desea que sus clientes entren a la bóveda para contar la cantidad de dinero en una cuenta. En vez de ello, los clientes van con un cajero en el banco. De manera similar, usted no necesita tener acceso a las variables de instancia de un objeto para poder utilizarlas. Puede utilizar las **propiedades** de un objeto. Las propiedades contienen **descriptores de acceso get** para leer los valores de las variables y **descriptores de acceso set** para almacenar valores en ellas.

El resto de este capítulo contiene ejemplos que demuestran los conceptos que presentamos aquí, dentro del contexto de la analogía del auto. Los primeros cuatro ejemplos que se sintetizan a continuación, se encargan de construir en forma incremental una clase llamada LibroCalificaciones:

1. El primer ejemplo presenta una clase llamada LibroCalificaciones con un método que sólo muestra un mensaje de bienvenida cuando se le llama. Le mostraremos cómo **crear un objeto** de esa clase y cómo llamarlo para que muestre el mensaje de bienvenida.
2. El segundo ejemplo modifica el primero, al permitir que el método reciba el nombre de un curso como “argumento” y al mostrar ese nombre como parte del mensaje de bienvenida.
3. El tercer ejemplo muestra cómo almacenar el nombre del curso en un objeto tipo LibroCalificaciones. Para esta versión de la clase, también le mostraremos cómo utilizar las propiedades para establecer el nombre del curso y obtenerlo.
4. El cuarto ejemplo demuestra cómo pueden inicializarse los datos en un objeto tipo LibroCalificaciones, a la hora de crear el objeto; el constructor de la clase se encarga de realizar el proceso de inicialización.

El último ejemplo en el capítulo presenta una clase llamada Cuenta, la cual refuerza los conceptos presentados en los primeros cuatro ejemplos e introduce el tipo decimal; un número decimal puede contener un punto decimal, como en 0.0345, -7.23 y 100.7, y se utiliza para realizar cálculos precisos, en especial los que implican valores monetarios. Para este fin presentamos una clase llamada Cuenta, la cual representa una cuenta bancaria

y mantiene su saldo `decimal`. La clase contiene un método para acreditar un depósito a la cuenta, con lo cual se incrementa el saldo; y una propiedad para obtener el saldo y asegurarse de que todos los valores asignados al mismo no sean negativos. El constructor de la clase inicializa el saldo de cada objeto tipo `Cuenta`, a la hora de crear el objeto. Crearemos dos objetos tipo `Cuenta` y haremos depósitos en cada uno de ellos para mostrar que cada objeto mantiene su propio saldo. El ejemplo también demuestra cómo introducir e imprimir en pantalla números tipo `decimal`.

4.3 Declaración de una clase con un método e instanciamiento del objeto de una clase

Empezaremos con un ejemplo que consiste en las clases `LibroCalificaciones` (figura 4.1) y `PruebaLibroCalificaciones` (figura 4.2). La clase `LibroCalificaciones` (declarada en el archivo `LibroCalificaciones.cs`) se utilizará para mostrar un mensaje en la pantalla (figura 4.2) para dar la bienvenida, al instructor, a la aplicación del libro de calificaciones. La clase `PruebaLibroCalificaciones` (declarada en el archivo `PruebaLibroCalificaciones.cs`) es una clase de prueba en la que el método `Main` creará y utilizará un objeto de la clase `LibroCalificaciones`. Por convención declaramos a las clases `LibroCalificaciones` y `PruebaLibroCalificaciones` en archivos separados, de tal forma que el nombre de cada archivo concuerde con el nombre de la clase que contiene.

Para empezar, seleccione **Archivo > Nuevo proyecto...** para abrir el cuadro de diálogo **Nuevo proyecto**, y después cree una **Aplicación de consola** llamada `LibroCalificaciones`. Elimine todo el código que el IDE proporciona de manera automática y sustitúyalo con el código de la figura 4.1.

Clase LibroCalificaciones

La **declaración de la clase** `LibroCalificaciones` (figura 4.1) contiene un método llamado `MostrarMensaje` (líneas 8-11), que muestra un mensaje en la pantalla. La línea 10 de la clase muestra el mensaje. Recuerde que una clase es como un plano de construcción; necesitamos crear un objeto de esta clase y llamar a su método para hacer que se ejecute la línea 10 y muestre su mensaje (haremos esto en la figura 4.2).

La declaración de la clase comienza en la línea 5. La palabra clave `public` es un **modificador de acceso**. Por ahora, a todas las clases las declaramos como `public`. La declaración de cada clase contiene la palabra clave `class`, seguida del nombre de la clase. El cuerpo de cada clase va encerrado entre un par de llaves izquierda y derecha (`{ }`), como en las líneas 6 y 12 de `LibroCalificaciones`.

En el capítulo 3, cada clase que declaramos tenía un método llamado `Main`. La clase `LibroCalificaciones` también tiene un método: `MostrarMensaje` (líneas 8-11). Recuerde que `Main` es un método especial que siempre se llama de manera automática cuando se ejecuta una aplicación. La mayoría de los métodos no se llama de manera automática. Como pronto verá, deberá llamar al método `MostrarMensaje` para indicarle que realice su tarea.

La declaración del método empieza con la palabra clave `public` para indicar que el método está “disponible para el público”; es decir, los métodos de otras clases pueden llamar a este método desde fuera del cuerpo de la

```

1  // Fig. 4.1: LibroCalificaciones.cs
2  // Declaración de una clase con un método.
3  using System;
4
5  public class LibroCalificaciones
6  {
7      // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
8      public void MostrarMensaje()
9      {
10         Console.WriteLine( "¡Bienvenido al Libro de calificaciones!" );
11     } // fin del método MostrarMensaje
12 } // fin de la clase LibroCalificaciones

```

Figura 4.1 | Declaración de la clase con un método.

declaración de la clase. La palabra clave `void` (conocida como el *tipo de valor de retorno* del método) indica que este método no devolverá (o regresará) ninguna información al *método que lo llamó* cuando complete su tarea. Cuando se hace una llamada a un método que especifica un tipo de valor de retorno distinto de `void` y completa su tarea, el método devuelve un resultado al método que lo llamó. Por ejemplo, cuando usted va a un cajero automático (ATM) y solicita el saldo de su cuenta, espera que el ATM le devuelva un valor que represente su saldo. Si tiene un método llamado `Cuadrado` que devuelve el cuadrado de su argumento, sería de esperarse que la instrucción

```
int resultado = Cuadrado( 2 );
```

devolviera el valor 4 del método `Cuadrado` y lo asignara a la variable `resultado`. Si tiene un método llamado `Maximo` que devuelva el mayor de los tres argumentos tipo entero, sería de esperarse que la instrucción

```
int mayor = Maximo( 27, 114, 51 );
```

devolviera el valor 114 del método `Maximo` y asignara ese valor a la variable `mayor`. Ya hemos utilizado métodos que devuelven información; por ejemplo, en el capítulo 3 utilizamos el *método* `ReadLine de Console` para introducir una cadena escrita por el usuario mediante el teclado. Cuando `ReadLine` recibe un valor como entrada, devuelve ese valor para utilizarlo en la aplicación.

El nombre del método `MostrarMensaje` va después del tipo de valor de retorno (línea 8). Por convención, los nombres de los métodos empiezan con la primera letra en minúscula, y todas las palabras subsiguientes en ese nombre empiezan con letra mayúscula. Los paréntesis después del nombre del método indican que éste es un método. Un conjunto vacío de paréntesis, como el que se muestra en la línea 8, indica que este método no requiere información adicional para realizar su tarea. Por lo general, a la línea 8 se le conoce como el *encabezado del método*. El cuerpo de cada método está delimitado por las llaves izquierda y derecha, como en las líneas 9 y 11.

El cuerpo de un método contiene una o varias instrucciones que realizan la tarea de ese método. En este caso, el método contiene una instrucción (línea 10) que muestra el mensaje "¡Bienvenido al libro de calificaciones!", seguido de una nueva línea en la ventana de consola. Después de que se ejecuta esta instrucción, el método completa su tarea.

A continuación nos gustaría utilizar la clase `LibroCalificaciones` en una aplicación. Como aprendió en el capítulo 3, el método `Main` comienza la ejecución de toda aplicación. La clase `LibroCalificaciones` no puede comenzar una aplicación, ya que no contiene a `Main`. Esto no fue un problema en el capítulo 3, ya que todas las clases que declaramos tenían un método `Main`. Para corregir este problema para la clase `LibroCalificaciones`, debemos declarar una clase separada que contenga un método `Main` o colocar un método `Main` en la clase `LibroCalificaciones`. Para ayudarlo en su preparación para las aplicaciones más extensas con las que se encontrará más adelante en este libro y en la industria, utilizaremos una clase separada (en este ejemplo, `PruebaLibroCalificaciones`) que contiene el método `Main` para probar cada una de las nuevas clases que vayamos a crear en este capítulo.

Agregar una clase a un proyecto de Visual C#

Para cada uno de los ejemplos en este capítulo, agregará una clase a su aplicación de consola. Para ello, haga clic con el botón derecho en el nombre del proyecto dentro del **Explorador de soluciones** y seleccione **Agregar > Nuevo elemento...** del menú desplegable. En el cuadro de diálogo **Agregar nuevo elemento** que aparezca, seleccione **Archivo de código** y escriba el nombre de su nuevo archivo; en este caso, `PruebaLibroCalificaciones.cs`. A continuación se agregará un nuevo archivo en blanco a su proyecto. Agregue el código de la figura 4.2 a este archivo.

Clase PruebaLibroCalificaciones

La declaración de la clase `PruebaLibroCalificaciones` (figura 4.2) contiene el método `Main` que controla la ejecución de nuestra aplicación. Cualquier clase que contenga un método `Main` (como se muestra en la línea 7) puede utilizarse para ejecutar una aplicación. La declaración de esta clase comienza en la línea 4 y termina en la 15. La clase sólo contiene un método `Main`, que es común en muchas clases que sólo se utilizan para iniciar la ejecución de una aplicación.

Las líneas 7-14 declaran el método `Main`. Una parte clave de hacer que el método `Main` inicie la ejecución de la aplicación es la palabra clave `static` (línea 7), que establece que `Main` es un método estático. Un método

```

1 // Fig. 4.2: PruebaLibroCalificaciones.cs
2 // Crea un objeto LibroCalificaciones y llama a su método MostrarMensaje.
3
4 public class PruebaLibroCalificaciones
5 {
6     // El método Main comienza la ejecución del programa
7     public static void Main( string[] args )
8     {
9         // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
10        LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
11
12        // llama al método MostrarMensaje de miLibroCalificaciones
13        miLibroCalificaciones.MostrarMensaje();
14    } // fin de Main
15 } // fin de la clase PruebaLibroCalificaciones

```

¡Bienvenido al Libro de calificaciones!

Figura 4.2 | Crear un objeto LibroCalificaciones y llamar a su método MostrarMensaje.

`static` es especial, ya que puede llamarse sin necesidad de crear primero un objeto de la clase (en este caso, `PruebaLibroCalificaciones`) en la que está declarado. En el capítulo 7, Métodos: un análisis más detallado, explicaremos la función de los métodos `static`.

En esta aplicación queremos llamar al método `MostrarMensaje` de la clase `LibroCalificaciones` para mostrar el mensaje de bienvenida en la ventana de consola. Por lo general, no se puede llamar a un método que pertenezca a otra clase, sino hasta que se cree un objeto de esa clase, como se muestra en la línea 10. Para empezar, declaramos la variable `miLibroCalificaciones`. Observe que el tipo de la variable es `LibroCalificaciones`: la clase que declaramos en la figura 4.1. Cada nueva clase que usted crea se convierte en un nuevo tipo en C#, el cual puede utilizarse para declarar variables y crear objetos. Los nuevos tipos de clases son accesibles para todas las clases dentro del mismo proyecto. Puede declarar nuevos tipos de clases según lo requiera; ésta es una de las razones por las que C# se conoce como un *lenguaje extensible*.

La variable `miLibroCalificaciones` (línea 10) se inicializa con el resultado de la *expresión de creación de objeto* `new LibroCalificaciones()`. El operador `new` crea un nuevo objeto de la clase especificada a la derecha de la palabra clave (es decir, `LibroCalificaciones`). Los paréntesis a la derecha de `LibroCalificaciones` son requeridos. Como aprenderá en la sección 4.9, esos paréntesis en combinación con el nombre de una clase representan una llamada a un constructor, que es similar a un método pero se utiliza sólo cuando se crea el objeto, para inicializar sus datos. En esta sección veremos que pueden colocarse los datos entre paréntesis para especificar los valores iniciales para los datos del objeto. Por ahora, sólo dejaremos los paréntesis vacíos.

Podemos utilizar a `miLibroCalificaciones` para llamar a su método `MostrarMensaje`. La línea 13 llama al método `MostrarMensaje` (líneas 8-11 de la figura 4.1) mediante el uso de la variable `miLibroCalificaciones`, seguida de un *operador punto* (`.`), el nombre del método `MostrarMensaje` y un conjunto vacío de paréntesis. Esta llamada hace que el método `MostrarMensaje` realice su tarea. La llamada a este método es distinta a las llamadas a los métodos en el capítulo 3 que mostraban información en una ventana de consola; cada una de esas llamadas a métodos proporcionaban argumentos que especificaban los datos a mostrar. Al principio de la línea 13, “`miLibroCalificaciones`”, indica que `Main` debe utilizar el objeto `LibroCalificaciones` que se creó en la línea 10. Los paréntesis vacíos en la línea 8 de la figura 4.1 indican que el método `MostrarMensaje` no requiere información adicional para realizar su tarea. Por esta razón, la llamada al método (línea 13 de la figura 4.2) especifica un conjunto vacío de paréntesis después del nombre del método, para indicar que no se van a pasar argumentos al método `MostrarMensaje`. Cuando `MostrarMensaje` completa su tarea, el método `Main` continúa su ejecución en la línea 14. Como éste es el final del método `Main`, la aplicación termina.

Diagrama de clases de UML para la clase LibroCalificaciones

La figura 4.3 presenta un *diagrama de clases de UML* para la clase `LibroCalificaciones` de la figura 4.1. Recuerde que en la sección 1.9 vimos que UML es un lenguaje gráfico, utilizado por los programadores para

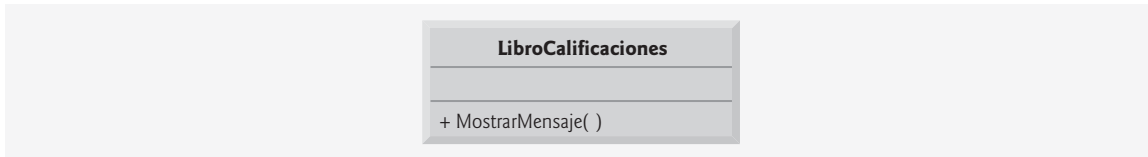


Figura 4.3 | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene una operación `public` llamada `MostrarMensaje`.

representar sus sistemas orientados a objetos de una manera estandarizada. En UML, cada clase se modela en un diagrama de clases en forma de un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado en forma horizontal y en negrita. El compartimiento de en medio contiene los atributos de la clase, que en C# corresponden a las variables de instancia y las propiedades. En la figura 4.3 el compartimiento de en medio está vacío, ya que la versión de la clase `LibroCalificaciones` en la figura 4.1 no tiene atributos. El compartimiento inferior contiene las operaciones de la clase, que en C# corresponden a los métodos. Para modelar las operaciones, UML lista el nombre de la operación seguido de un conjunto de paréntesis. La clase `LibroCalificaciones` tiene un método llamado `MostrarMensaje`, por lo que el compartimiento inferior de la figura 4.3 lista una operación con este nombre. El método `MostrarMensaje` no requiere información adicional para realizar sus tareas, por lo cual hay paréntesis vacíos después de `MostrarMensaje` en el diagrama de clases, de igual forma que como aparecieron en la declaración del método en la línea 8 de la figura 4.1. El signo más (+) que va antes del nombre de la operación indica que `MostrarMensaje` es una operación pública en UML (es decir, un método `public` en C#). Al signo más también se le conoce como *símbolo de visibilidad pública*. A menudo utilizaremos los diagramas de clases de UML para sintetizar los atributos y las operaciones de una clase.

4.4 Declaración de un método con un parámetro

En nuestra analogía del auto de la sección 4.2, hablamos sobre el hecho de que al oprimir el pedal del acelerador se envía un mensaje al auto para que realice una tarea: hacer que vaya más rápido. Pero, ¿qué tan rápido debería acelerar el auto? Como sabe, entre más oprima el pedal, mayor será la aceleración del auto. Por lo tanto, el mensaje para el auto en realidad incluye tanto la tarea a realizar como información adicional que ayuda al auto a realizar su tarea. A la información adicional se le conoce como *parámetro*; el valor del parámetro ayuda al auto a determinar qué tan rápido debe acelerar. De manera similar, un método puede requerir uno o más parámetros que representan la información adicional que necesita para realizar su tarea. La llamada a un método proporciona valores (llamados argumentos) para cada uno de los parámetros de ese método. Por ejemplo, el método `Console.WriteLine` requiere un argumento que especifica los datos a mostrar en una ventana de consola. Así mismo, para realizar un depósito en una cuenta bancaria, un método llamado `Deposito` especifica un parámetro que representa el monto a depositar. Cuando se hace una llamada al método `Deposito`, se asigna al parámetro del método un valor como argumento que representa el monto a depositar. Entonces el método realiza un depósito por ese monto e incrementa el balance de la cuenta.

Nuestro siguiente ejemplo declara la clase `LibroCalificaciones` (figura 4.4) con un método `MostrarMensaje` que señala el nombre del curso como parte del mensaje de bienvenida (en la figura 4.5 podrá ver la ejecución de ejemplo). El nuevo método `MostrarMensaje` requiere un parámetro que representa el nombre del curso a imprimir en pantalla.

Antes de hablar sobre las nuevas características de la clase `LibroCalificaciones`, veamos cómo se utiliza la nueva clase desde el método `Main` de la clase `PruebaLibroCalificaciones` (figura 4.5). La línea 12 crea un objeto de la clase `LibroCalificaciones` y lo asigna a la variable `miLibroCalificaciones`. La línea 15 pide al usuario que escriba el nombre de un curso. La línea 16 lee el nombre que introduce el usuario y lo asigna a la variable `nombreDelCurso`, mediante el uso del método `ReadLine` de `Console` para realizar la operación de entrada. El usuario escribe el nombre del curso y oprime `Intro` para enviarlo a la aplicación. Observe que al oprimir `Intro` se inserta un carácter de nueva línea al final de los caracteres escritos por el usuario. El método `ReadLine` lee los caracteres que escribió el usuario hasta encontrar el carácter de nueva línea y después devuelve un valor tipo `string` que contiene los caracteres hasta, pero sin incluir a, la nueva línea. El carácter de nueva línea se descarta.

```

1 // Fig. 4.4: LibroCalificaciones.cs
2 // Declaración de la clase con un método que tiene un parámetro.
3 using System;
4
5 public class LibroCalificaciones
6 {
7     // muestra un mensaje de bienvenida para el usuario del LibroCalificaciones
8     public void MostrarMensaje( string nombreCurso )
9     {
10         Console.WriteLine( "¡Bienvenido al libro de calificaciones para\n{0}!",
11                             nombreCurso );
12     } // fin del método MostrarMensaje
13 } // fin de la clase LibroCalificaciones

```

Figura 4.4 | Declaración de la clase con un método que tiene un parámetro.

```

1 // Fig. 4.5: LibroPruebaCalificaciones.cs
2 // Crea objeto LibroCalificaciones y pasa una cadena a
3 // su método MostrarMensaje.
4 using System;
5
6 public class LibroPruebaCalificaciones
7 {
8     // El método Main comienza la ejecución del programa
9     public static void Main( string[] args )
10    {
11        // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
12        LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
13
14        // pide el nombre del curso y lo recibe como entrada
15        Console.WriteLine( "Por favor escriba el nombre del curso:" );
16        string nombreDelCurso = Console.ReadLine(); // lee una línea de texto
17        Console.WriteLine(); // imprime en pantalla una línea en blanco
18
19        // llama al método MostrarMensaje de miLibroCalificaciones
20        // y pasa nombreDelCurso como argumento
21        miLibroCalificaciones.MostrarMensaje( nombreDelCurso );
22    } // fin de Main
23 } // fin de la clase LibroPruebaCalificaciones

```

Por favor escriba el nombre del curso:
CS101 Introducción a la programación en C#

¡Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C#!

Figura 4.5 | Creación de un objeto LibroCalificaciones y paso de una cadena a su método MostrarMensaje.

La línea 21 llama al método `MostrarMensaje` de `miLibroCalificaciones`. La variable `nombreDelCurso` entre paréntesis es el argumento que se pasa al método `MostrarMensaje` para que pueda realizar su tarea. El valor de la variable `nombreDelCurso` en `Main` se convierte en el valor del parámetro `nombreCurso` del método `MostrarMensaje` en la línea 8 de la figura 4.4. Al ejecutar esta aplicación, observe que el método `MostrarMensaje` imprime en pantalla el nombre que usted escribió como parte del mensaje de bienvenida (figura 4.5).



Observación de ingeniería de software 4.1

Por lo general, los objetos se crean mediante el uso de `new`. Una excepción es la literal de cadena que está encerrada entre comillas, tal como `"ho1a"`. Las literales de cadena son referencias a objetos `string` que C# crea de manera implícita.

Más sobre los argumentos y los parámetros

Al declarar un método, debe especificar en su declaración si éste requiere datos para realizar su tarea. Para ello hay que colocar información adicional en la **lista de parámetros** del método, la cual se encuentra en los paréntesis que van después del nombre del método. La lista de parámetros puede contener cualquier número de parámetros, incluso ninguno. Los paréntesis vacíos después del nombre del método (como en la figura 4.1, línea 8) indican que un método no requiere parámetros. En la figura 4.4, la lista de parámetros de `MostrarMensaje` (línea 8) declara que el método requiere un parámetro. Cada parámetro debe especificar un tipo y un identificador. En este caso, el tipo `string` y el identificador `nombreCurso` indican que el método `MostrarMensaje` requiere un objeto `string` para realizar su tarea. En el instante en que se llama al método, el valor del argumento en la llamada se asigna al parámetro correspondiente (en este caso, `nombreCurso`) en el encabezado del método. Después, el cuerpo del método utiliza el parámetro `nombreCurso` para acceder al valor. Las líneas 10-11 de la figura 4.4 muestran el valor del parámetro `nombreCurso`, mediante el uso del elemento de formato `{0}` en el primer argumento de `WriteLine`. Observe que el nombre de la variable de parámetro (figura 4.4, línea 8) puede ser igual o distinto al nombre de la variable de argumento (figura 4.5, línea 21).

Un método puede especificar múltiples parámetros; sólo hay que separar un parámetro de otro mediante una coma. El número de argumentos en la llamada a un método debe concordar con el número de parámetros en la lista de parámetros de la declaración del método que se llamó. Además, los tipos de los argumentos en la llamada a un método deben ser consistentes con los tipos de los parámetros correspondientes en la declaración del método (como veremos en capítulos posteriores, no siempre se requiere que el tipo de un argumento y el tipo de su correspondiente parámetro sean idénticos). En nuestro ejemplo, la llamada al método pasa un argumento de tipo `string` (`nombreDelCurso` se declara como `string` en la línea 16 de la figura 4.5) y la declaración del método especifica un parámetro de tipo `string` (línea 8 en la figura 4.4). Por lo tanto, el tipo del argumento en la llamada al método concuerda exactamente con el tipo del parámetro en el encabezado del método.



Error común de programación 4.1

Si el número de argumentos en la llamada a un método no concuerda con el número de parámetros en la declaración del método, se produce un error de compilación.



Error común de programación 4.2

Si los tipos de los argumentos en la llamada a un método no son consistentes con los tipos de los parámetros correspondientes en la declaración del método, se produce un error de compilación.

Diagrama de clases actualizado para la clase LibroCalificaciones

El diagrama de clases de UML de la figura 4.6 modela la clase `LibroCalificaciones` de la figura 4.4. Al igual que la figura 4.4, esta clase `LibroCalificaciones` contiene la operación `public` llamada `MostrarMensaje`. Sin embargo, esta versión de `MostrarMensaje` tiene un parámetro. La forma en que UML modela un parámetro es un poco distinta a la de C#, ya que lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre paréntesis, después del nombre de la operación. UML cuenta con varios tipos de datos que son similares a los tipos de C#. Por ejemplo, los tipos `String` e `Integer` de UML corresponden a los tipos `string` e `int` de C#. Por desgracia, UML no proporciona tipos que correspondan a todos los tipos de C#. Por esta razón y para evitar confusiones entre los tipos de UML y los de C#, sólo utilizamos tipos de C# en nuestros diagramas de UML. El método `MostrarMensaje` de la clase `LibroCalificaciones` (figura 4.4) tiene un parámetro `string` llamado `nombreCurso`, por lo que en la figura 4.6 se lista a `nombreCurso : string` entre los paréntesis que van después de `MostrarMensaje`.

Observaciones acerca del uso de las directivas

Observe la directiva `using` en la figura 4.5 (línea 4). Esto indica al compilador que la aplicación utiliza clases en el espacio de nombres `System`, como la clase `Console`. ¿Por qué necesitamos una directiva `using` para utilizar la clase `Console`, pero no la clase `LibroCalificaciones`? Existe una relación especial entre las clases que se compilan en el mismo proyecto, como las clases `LibroCalificaciones` y `PruebaLibroCalificaciones`. De manera predeterminada se considera que dichas clases se encuentran en el mismo espacio de nombres. No se requiere una directiva `using` cuando una clase en un espacio de nombres utiliza a otra dentro del mismo espacio de nombres;

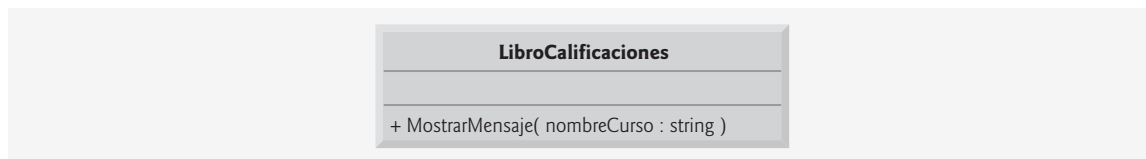


Figura 4.6 | Diagrama de clases de UML que indique que la clase `LibroCalificaciones` tiene una operación pública llamada `MostrarMensaje`, con un parámetro llamado `nombreCurso` de tipo `string`.

como cuando la clase `PruebaLibroCalificaciones` utiliza a la clase `LibroCalificaciones`. En la sección 9.14 verá cómo puede declarar sus propios espacios de nombres mediante la palabra clave `namespace`. Por cuestión de simpleza, nuestros ejemplos en este capítulo no declaran un espacio de nombres. Cualquier clase que no se coloque de manera explícita en un espacio de nombres se coloca de manera implícita en lo que se denomina **espacio de nombres global**.

En realidad no se requiere la directiva `using` en la línea 4 si nos referimos siempre a la clase `Console` como `System.Console`, con lo cual se incluye el espacio de nombres y el nombre de la clase completos. A esto se le conoce como el **nombre de clase completamente calificado**. Por ejemplo, podríamos escribir la línea 15 como

```
System.Console.WriteLine( "Por favor escriba el nombre del curso:" );
```

La mayoría de los programadores de C# considera que el uso de nombres completamente calificados es incómodo, por lo que prefieren utilizar directivas `using`. El código que genera el Diseñador de formularios de Visual C# utiliza nombres completamente calificados.

4.5 Variables de instancia y propiedades

En el capítulo 3 declaramos todas las variables de una aplicación en el método `Main` de esa aplicación. Las variables que se declaran en el cuerpo de un método específico se conocen como **variables locales** y sólo se pueden utilizar en ese método. Cuando termina un método, se pierden los valores de sus variables locales. En la sección 4.2 vimos que un objeto tiene atributos que lleva consigo cuando se utiliza en una aplicación. Dichos atributos existen antes de que se llame a un método de un objeto, y después de que el método completa su ejecución.

En la declaración de una clase, los atributos se representan como variables. A dichas variables se les denomina **campos** y se declaran dentro de la declaración de una clase, pero fuera de los cuerpos de las declaraciones de los métodos de esa clase. Cuando cada objeto de una clase mantiene su propia copia de un atributo, al campo que representa ese atributo se le denomina variable de instancia; cada objeto (instancia de la clase) tiene una instancia separada de esa variable en la memoria. [Nota: en el capítulo 9, Clases y objetos: un análisis más detallado, hablaremos sobre otro tipo de campo llamado variable `static`, en donde todos los objetos de la misma clase comparten una copia de la variable.]

Por lo general, una clase consiste de una o más propiedades que manipulan los atributos que pertenecen a un objeto específico de la clase. El ejemplo en esta sección demuestra una clase `LibroCalificaciones` que contiene una variable de instancia llamada `nombreCurso` para representar el nombre del curso de un objeto `LibroCalificaciones` específico, y una propiedad llamada `NombreCurso` para manipular a `nombreCurso`.

La clase `LibroCalificaciones` con una variable de instancia y una propiedad

En nuestra siguiente aplicación (figuras 4.7-4.8), la clase `LibroCalificaciones` (figura 4.7) mantiene el nombre del curso como una variable de instancia, de manera que se pueda utilizar o modificar en cualquier momento, durante la ejecución de una aplicación. La clase también contiene un método llamado `MostrarMensaje` (líneas 24-30) y una propiedad llamada `NombreCurso` (líneas 11-21). En el capítulo 2 vimos que las propiedades se utilizan para manipular los atributos de un objeto. Por ejemplo, en ese capítulo utilizamos la propiedad `Text` de un objeto `Label` para especificar el texto a mostrar en el objeto `Label`. En este ejemplo utilizamos una propiedad en el código, en vez de usarla en la ventana **Propiedades** del IDE. Para ello, primero declaramos una propiedad como miembro de la clase `LibroCalificaciones`. Como pronto verá, la propiedad `NombreCurso` de `LibroCalificaciones` puede utilizarse para almacenar el nombre de un curso en un objeto `LibroCalificaciones` (en la variable de instancia `nombreCurso`), o para recuperar el nombre del curso del objeto `LibroCalificaciones` (de

```

1  // Fig. 4.7: LibroCalificaciones.cs
2  // Clase LibroCalificaciones que contiene una variable de instancia cursoNombre
3  // y una propiedad para obtener (get) y establecer (set) su valor.
4  using System;
5
6  public class LibroCalificaciones
7  {
8      private string nombreCurso; // nombre del curso para este LibroCalificaciones
9
10     // propiedad para obtener (get) y establecer (set) el nombre del curso
11     public string NombreCurso
12     {
13         get
14         {
15             return nombreCurso;
16         } // fin de get
17         set
18         {
19             nombreCurso = value;
20         } // fin de set
21     } // fin de la propiedad NombreCurso
22
23     // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
24     public void MostrarMensaje()
25     {
26         // usa la propiedad NombreCurso para obtener el
27         // nombre del curso que representa este LibroCalificaciones
28         Console.WriteLine( "¡Bienvenido al libro de calificaciones para\n{0}!",
29                             NombreCurso ); // muestra la propiedad NombreCurso
30     } // fin del método MostrarMensaje
31 } // fin de la clase LibroCalificaciones

```

Figura 4.7 | La clase `LibroCalificaciones` contiene una variable de instancia `private` llamada `nombreCurso` y una propiedad `public` para obtener (`get`) y establecer (`set`) su valor.

la variable de instancia `nombreCurso`). El método `MostrarMensaje` (que ahora no especifica parámetros) sigue mostrando un mensaje de bienvenida, que incluye el nombre del curso. No obstante, ahora el método utiliza la propiedad `NombreCurso` para obtener el nombre del curso de la variable de instancia `nombreCurso`.

Un instructor común enseña más de un curso, cada uno con su propio nombre. La línea 8 que declara a `nombreCurso` como una variable de tipo `string`, es una declaración de una variable de instancia, ya que la variable se declara dentro del cuerpo de la clase (líneas 7-31), pero fuera de los cuerpos del método (líneas 24-30) y de la propiedad (líneas 11-21) de la clase. Cada instancia (es decir, objeto) de la clase `LibroCalificaciones` contiene una copia de cada una de las variables de instancia. Por ejemplo, si hay dos objetos `LibroCalificaciones`, cada uno tiene su propia copia de `nombreCurso` (una por objeto). Todos los métodos y las propiedades de la clase `LibroCalificaciones` pueden manipular en forma directa su variable de instancia `nombreCurso`, pero se considera una buena práctica que los métodos de una clase utilicen las propiedades de la misma para manipular las variables de instancia (como se hace en la línea 29 del método `MostrarMensaje`). Pronto se volverán más claras las razones de hacer esto, en relación con la ingeniería de software.

Modificadores de acceso `public` y `private`

La mayoría de las declaraciones de las variables de instancia va precedida por la palabra clave `private` (como en la línea 8). Al igual que `public`, la palabra clave `private` es un modificador de acceso. Las variables o los métodos que se declaran con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que están declarados. Por ende, la variable `nombreCurso` sólo se puede utilizar dentro de la propiedad `NombreCurso` y del método `MostrarMensaje` de la clase `LibroCalificaciones`.



Observación de ingeniería de software 4.2

Coloque un modificador de acceso antes de cada campo y dé la declaración de cada método. Como regla general, las variables de instancia deben declararse como `private`; los métodos y las propiedades como `public`. Si se omite el modificador de acceso antes del miembro de una clase, el miembro se declara de manera implícita como `private`. (Más adelante veremos que es apropiado declarar ciertos métodos `private`, sólo si otros métodos de la clase tendrán acceso a ellos.)



Buena práctica de programación 4.1

Es preferible listar primero los campos de una clase para que, cuando lea el código, pueda ver los nombres y los tipos de las variables antes de que se utilicen en los métodos de la clase. Es posible listar los campos de la clase en cualquier parte de ésta que sea afuera de las declaraciones de sus métodos, pero será más fácil leer el código si se agrupan en un solo lugar.



Buena práctica de programación 4.2

Al colocar una línea en blanco entre las declaraciones de los métodos y de las propiedades, se mejora la legibilidad de la aplicación.

Al proceso de declarar las variables de instancia con el modificador de acceso `private` se le conoce como **ocultamiento de información**. Cuando una aplicación crea (instancia) un objeto de la clase `LibroCalificaciones`, la variable `nombreCurso` se encapsula (oculta) en el objeto y sólo los métodos y las propiedades de la clase de ese objeto pueden tener acceso a ella. En la clase `LibroCalificaciones`, la propiedad `NombreClase` manipula la variable de instancia `nombreCurso`.

Cómo establecer y obtener los valores de las variables de instancia `private`

¿Cómo podemos permitir que un programa manipule las variables de instancia `private` de una clase y asegurar que permanezcan en un estado válido? Necesitamos proporcionar medios controlados para que los programadores puedan “obtener” (es decir, recuperar) y “establecer” (esto es, modificar) el valor en una variable de instancia. Para estos fines, los programadores que utilizan lenguajes distintos a C# utilizan por lo general métodos conocidos como *get* y *set*. Estos métodos casi siempre son `public` y proporcionan los medios para que el cliente acceda a los datos `private` o los modifique. Por cuestión de historia, estos métodos comienzan con las palabras “Get” (“Obtener”) y “Set” (“Establecer”); por ejemplo, en nuestra clase `LibroCalificaciones`, si quisiéramos utilizar dichos métodos podríamos llamarlos `ObtenerNombreCurso` y `EstablecerNombreCurso`, en forma respectiva.

Aunque es posible definir métodos como `ObtenerNombreCurso` y `EstablecerNombreCurso`, C# proporciona una solución más elegante. A continuación veremos cómo declarar y utilizar las propiedades.

La clase `LibroCalificaciones` con una propiedad

La **declaración de la propiedad** `NombreCurso` de la clase `LibroCalificaciones` se encuentra en las líneas 11-21 de la figura 4.7. La propiedad empieza en la línea 11 con un modificador de acceso (en este caso, `public`), seguido del tipo que representa la propiedad (`string`) y del nombre de la propiedad (`NombreCurso`). Por lo general los nombres de las propiedades empiezan con mayúsculas.

Las propiedades contienen **descriptores de acceso** que se encargan de todos los detalles relacionados con los procesos de devolver y modificar datos. La declaración de una propiedad puede contener un descriptor de acceso *get*, un descriptor de acceso *set* o ambos. El descriptor de acceso *get* (líneas 13-16) permite a un cliente leer el valor de la variable de instancia `private` llamada `nombreCurso`; el descriptor de acceso *set* (líneas 17-20) permite a un cliente modificar el valor de `nombreCurso`.

Después de definir una propiedad, puede utilizarla como una variable en su código. Por ejemplo, puede asignar un valor a una propiedad mediante el uso del operador `=` (asignación). Esto ejecuta el código en el descriptor de acceso *set* de la propiedad, para establecer el valor de la correspondiente variable de instancia. De manera similar, al hacer referencia a la propiedad para utilizar su valor (por ejemplo, para mostrarlo en pantalla) se ejecuta el código en el descriptor de acceso *get* de la propiedad para obtener el valor de la correspondiente variable de instancia. En breve le mostraremos cómo utilizar las propiedades. Por convención, nombramos cada propiedad con el nombre de la variable de instancia que manipula, empezando con letra mayúscula (por ejemplo, `NombreCurso`

es la propiedad que representa a la variable de instancia `nombreCurso`); como C# es sensible a mayúsculas/mi-núsculas, entonces éstos son identificadores distintos.

Descriptores de acceso `get` y `set`

Veamos más de cerca los descriptores de acceso `get` y `set` de la propiedad `NombreCurso` (figura 4.7). El descriptor de acceso `get` (líneas 13-16) empieza con el identificador **`get`** y está delimitado entre llaves. El cuerpo del descriptor de acceso contiene una ***instrucción de retorno***, que consta de la palabra clave **`return`**, seguida de una expresión. El valor de la expresión se devuelve al código del cliente que está usando la propiedad. En este ejemplo se devuelve el valor de `nombreCurso` cuando se hace referencia a la propiedad `NombreCurso`. Por ejemplo, la siguiente instrucción:

```
string elNombreDelCurso = libroCalificaciones.NombreCurso;
```

en donde `libroCalificaciones` es un objeto de la clase `LibroCalificaciones`, el cual ejecuta el descriptor de acceso `get` de la propiedad `NombreCurso`, que a su vez devuelve el valor de la variable de instancia `nombreCurso`. Después, ese valor se almacena en la variable `elNombreDelCurso`. Observe que la propiedad `NombreCurso` se puede utilizar al igual que una variable de instancia. La notación de la propiedad permite al cliente considerarla como los datos subyacentes. De nuevo, el cliente no puede manipular en forma directa la variable de instancia `nombreCurso`, ya que es `private`.

El descriptor de acceso `set` (líneas 17-20) empieza con el identificador **`set`** y está delimitado por llaves. Cuando aparece la propiedad `NombreCurso` en una instrucción de asignación, como en

```
libroCalificaciones.NombreCurso = "CS100 Introducción a las computadoras";
```

el texto "CS100 Introducción a las computadoras" se pasa a un parámetro implícito llamado `value` y se ejecuta el descriptor de acceso `set`. Observe que `value` se declara en forma implícita y se inicializa en el descriptor de acceso `set`; sería un error de compilación declarar una variable local llamada `value` en este cuerpo. La línea 19 almacena este valor en la variable de instancia `nombreCurso`. Los descriptores de acceso `set` no devuelven datos cuando completan sus tareas.

Las instrucciones dentro de la propiedad en las líneas 15 y 19 (figura 4.7) tienen acceso a `nombreCurso`, aun y cuando ésta se declaró fuera de la propiedad. Podemos utilizar la variable de instancia `nombreCurso` en los métodos y las propiedades de la clase `LibroCalificaciones`, ya que `nombreCurso` es una variable de instancia de la clase. El orden en el que se declaran los métodos y las propiedades en una clase no determina cuándo se van a llamar en tiempo de ejecución, por lo que podemos declarar el método `MostrarMensaje` (que utiliza la propiedad `NombreCurso`) antes de declarar la propiedad `NombreCurso`. Dentro de la propiedad en sí, los descriptores de acceso `get` y `set` pueden aparecer en cualquier orden, y es posible omitir cualquiera de ellos. En el capítulo 9 veremos cómo omitir un descriptor de acceso `get` o `set` para crear las denominadas propiedades de "sólo lectura" y "sólo escritura", en forma respectiva.

Uso de la propiedad `NombreCurso` en el método `MostrarMensaje`

El método `MostrarMensaje` (líneas 24-30 de la figura 4.7) no recibe parámetros. Las líneas 28-29 imprimen en pantalla un mensaje de bienvenida, que incluye el valor de la variable de instancia `nombreCurso`. No hacemos referencia a `nombreCurso` en forma directa, sino que accedemos a la propiedad `NombreCurso` (línea 29), que ejecuta su descriptor de acceso `get` y así devuelve el valor de `nombreCurso`.

La clase `PruebaLibroCalificaciones` para demostrar la clase `LibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 4.8) crea un objeto `LibroCalificaciones` y demuestra el uso de la propiedad `NombreCurso`. La línea 11 crea un objeto `LibroCalificaciones` y lo asigna a la variable local `miLibroCalificaciones`, de tipo `LibroCalificaciones`. Las líneas 14-15 muestran el nombre inicial del curso mediante el uso de la propiedad `NombreCurso` del objeto; con esto se ejecuta el descriptor de acceso `get` de la propiedad, el cual devuelve el valor `nombreCurso`.

Observe que la primera línea de la salida muestra un nombre vacío (marcado con ' '). A diferencia de las variables locales que no se inicializan en forma automática, cada campo tiene un ***valor inicial predeterminado***: un valor que proporciona C# cuando usted no especifica el valor inicial. Por ende, no se requiere inicializar los

```

1 // Fig. 4.8: PruebaLibroCalificaciones.cs
2 // Creación y manipulación de un objeto LibroCalificaciones.
3 using System;
4
5 public class PruebaLibroCalificaciones
6 {
7     // El método Main comienza la ejecución del programa
8     public static void Main( string[] args )
9     {
10         // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
11         LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
12
13         // muestra el valor inicial de NombreCurso
14         Console.WriteLine( "El nombre inicial del curso es: '{0}'\n",
15             miLibroCalificaciones.NombreCurso );
16
17         // pide y lee el nombre del curso
18         Console.WriteLine( "Por favor escriba el nombre del curso:" );
19         string elNombre = Console.ReadLine(); // lee una línea de texto
20         miLibroCalificaciones.NombreCurso = elNombre; // establece el nombre usando una
           propiedad
21         Console.WriteLine(); // imprime en pantalla una línea en blanco
22
23         // muestra el mensaje de bienvenida después de especificar el nombre del curso
24         miLibroCalificaciones.MostrarMensaje();
25     } // fin de Main
26 } // fin de la clase PruebaLibroCalificaciones

```

El nombre inicial del curso es: ''

Por favor escriba el nombre del curso:

CS101 Introducción a la programación en C#

¡Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C#!

Figura 4.8 | Creación y manipulación de un objeto LibroCalificaciones.

campos en forma explícita antes de utilizarlos en una aplicación; a menos que deban inicializarse con valores distintos de los predeterminados. El valor predeterminado para una variable de instancia de tipo `string` (como `nombreCurso`) es `null`. Cuando se muestra en pantalla una variable `string` que contiene el valor `null`, no se muestra ningún texto. En la sección 4.8 hablaremos sobre el significado de `null`.

La línea 18 pide al usuario que escriba el nombre para el curso. La variable `string` local `elNombre` (declarada en la línea 19) se inicializa con el nombre del curso que escribió el usuario, el cual se devuelve mediante la llamada a `ReadLine`. La línea 20 asigna `elNombre` a la propiedad `NombreCurso` del objeto `miLibroCalificaciones`. Cuando se asigna un valor a `NombreCurso`, el valor especificado (en este caso, `elNombre`) se asigna al parámetro implícito `value` del descriptor de acceso `set` de `NombreCurso` (líneas 17-20, figura 4.7). Después el parámetro `value` se asigna mediante el descriptor de acceso `set` a la variable de instancia `nombreCurso` (línea 19 de la figura 4.7). La línea 21 (figura 4.8) muestra en pantalla una línea en blanco y después la línea 24 llama al método `MostrarMensaje` de `miLibroCalificaciones` para mostrar el mensaje de bienvenida que contiene el nombre del curso.

4.6 Diagrama de clases de UML con una propiedad

La figura 4.9 contiene un diagrama de clases de UML actualizado para la versión de `LibroCalificaciones` de la figura 4.7. En UML las propiedades se modelan como atributos; la propiedad (en este caso, `NombreCurso`) se lista como un atributo público, como lo indica el signo más (+), y se le antepone la palabra “property” encerrada entre

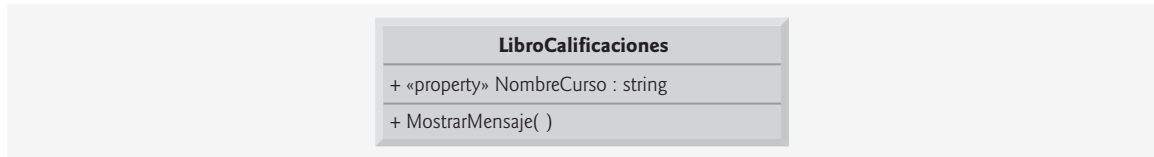


Figura 4.9 | Diagrama de clases de UML, en el que se indica que la clase `LibroCalificaciones` tiene una propiedad pública `NombreCurso` de tipo `string` y un método público.

los signos « y ». El uso de palabras descriptivas entre los signos « y » (lo que se conoce como *estereotipos* en UML) ayuda a distinguir las propiedades de otros atributos y operaciones. Para indicar el tipo de propiedad, UML coloca un signo de dos puntos y un tipo después del nombre de la propiedad. Los descriptores de acceso `get` y `set` de la propiedad están implícitos, por lo que no se listan en el diagrama de UML. La clase `LibroCalificaciones` también contiene un método `public` llamado `MostrarMensaje`, por lo que el diagrama de clases lista esta operación en el tercer compartimiento. Recuerde que el signo más (+) es el símbolo de visibilidad pública.

En la sección anterior aprendió a declarar una propiedad en código de C#. Vio que por lo general a una propiedad se le da el mismo nombre que la variable de instancia que manipula, pero con la primera letra en mayúscula (por ejemplo, la propiedad `NombreCurso` manipula a la variable de instancia `nombreCurso`). Un diagrama de clases le ayuda a diseñar una clase, por lo que no se requiere mostrar todos los detalles de implementación de la clase. Debido a que una variable de instancia manipulada por una propiedad es en realidad un detalle de implementación de esa propiedad, nuestro diagrama de clases no muestra la variable de instancia `nombreCurso`. Un programador que implemente la clase `LibroCalificaciones` con base en este diagrama de clases crearía la variable de instancia `nombreCurso` como parte del proceso de implementación (como hicimos en la figura 4.7).

En ocasiones, tal vez sea necesario modelar las variables de instancia `private` de una clase que no sean propiedades. Al igual que las propiedades, las variables de instancia son atributos de una clase y se modelan en el compartimiento de en medio de un diagrama de clases. Para representar las variables de instancia como atributos, UML lista el nombre del atributo, seguido de un signo de dos puntos y del tipo del atributo. Para indicar que un atributo es `private`, un diagrama de clases listaría el *símbolo de visibilidad privada* [un signo menos (-)] antes del nombre del atributo. Por ejemplo, la variable de instancia `nombreCurso` en la figura 4.7 se modelaría como “- `nombreCurso : string`” para indicar que es un atributo privado de tipo `string`.

4.7 Ingeniería de software con propiedades y los descriptores de acceso set y get

El uso de las propiedades como se describió antes en este capítulo parece violar la noción de los datos `private`. Aunque proporcionar una propiedad con descriptores de acceso `get` y `set` podría parecer lo mismo que hacer que su correspondiente variable de instancia sea `public`, en realidad no es así. Cualquier propiedad o método en el programa puede leer o escribir en una variable de instancia `public`. Si una variable de instancia es `private`, el código cliente puede acceder a esa variable de instancia sólo en forma indirecta, a través de las propiedades o métodos no `private` de la clase. Esto permite a la clase controlar la forma en la que se establecen o se devuelven los datos. Por ejemplo, los descriptores de acceso `get` y `set` pueden traducir entre el formato de los datos que utiliza el cliente y el formato almacenado en la variable de instancia `private`.

Considere una clase llamada `Reloj`, que representa la hora del día como una variable de instancia `private` `int` llamada `tiempo`, que contiene el número de segundos transcurridos desde medianoche. Suponga que la clase cuenta con una propiedad `Tiempo` de tipo `string` para manipular esta variable de instancia. Aunque por lo general los descriptores de acceso `get` devuelven los datos en la misma forma exacta en la que están almacenados en un objeto, no necesitan exponer los datos en este formato “puro”. Cuando un cliente hace referencia a la propiedad `Tiempo` de un objeto `Reloj`, el descriptor de acceso `get` de la propiedad puede utilizar la variable de instancia `tiempo` para determinar el número de horas, minutos y segundos transcurridos desde medianoche, para después devolver la hora como un objeto `string` de la forma “HH:MM:SS”. De manera similar, suponga que se asigna un objeto `string` de la forma “HH:MM:SS” a la propiedad `Tiempo` de un objeto `Reloj`. Mediante el uso de las capacidades de `string` que se presentan en el capítulo 16 y el método `Convert.ToInt32` presentado en la sección 3.6, el descriptor de acceso `set` de la propiedad `Tiempo` podría convertir este objeto `string` en un número `int`

de segundos transcurridos a partir de medianoche, y almacenar el resultado en la variable de instancia `private tiempo` del objeto `Reloj`. El descriptor de acceso `set` de la propiedad `Tiempo` también puede proporcionar capacidades de **validación de datos**, para escudriñar los intentos de modificar el valor de la variable de instancia y asegurar que el valor que reciba represente una hora válida (por ejemplo: "12:30:45" es válida, pero "42:85:70" no). En la sección 4.10 demostraremos el uso de la validación de datos. Así, aunque los descriptors de acceso de una propiedad permiten a los clientes manipular los datos `private`, controlan con cuidado esas manipulaciones y los datos `private` del objeto permanecen encapsulados (es decir, ocultos) en forma segura dentro del objeto. Esto no es posible con las variables de instancia `public`, que los clientes pueden establecer con facilidad y asignarles valores inválidos.

También es conveniente que los propios métodos de la clase utilicen las propiedades de la misma para manipular sus variables de instancia `private`, aun y cuando los métodos pueden acceder en forma directa a las variables de instancia `private`. Al acceder a una variable de instancia a través de los descriptors de acceso de una propiedad, como en el cuerpo del método `MostrarMensaje` (figura 4.7, líneas 28-29), se crea una clase más robusta que es más fácil de mantener y tiene menos probabilidad de fallar. Si decidimos modificar la representación de la variable de instancia `nombreCurso` de alguna forma, la declaración del método `MostrarMensaje` no requiere modificación; sólo tendrán que cambiar los cuerpos de los descriptors de acceso `get` y `set` de la propiedad `NombreCurso` que manipulan en forma directa a la variable de instancia. Por ejemplo, suponga que deseamos representar el nombre del curso como dos variables de instancia separadas: `numeroCurso` (por ejemplo, "CS101") y `tituloCurso` (por ejemplo, "Introducción a la programación en C#"). El método `MostrarMensaje` puede seguir utilizando el descriptor de acceso `get` de la propiedad `NombreCurso` para obtener el nombre completo del curso y mostrarlo como parte del mensaje de bienvenida. En este caso, el descriptor de acceso `get` tendría que crear y devolver un objeto `string` con el valor de `numeroCurso`, seguido del valor de `tituloCurso`. El método `MostrarMensaje` seguiría mostrando en pantalla el título completo del curso "CS101 Introducción a la programación en C#", debido a que no se ve afectado por la modificación de las variables de instancia de la clase.



Observación de ingeniería de software 4.3

Al acceder a los datos `private` a través de los descriptors de acceso `set` y `get` no sólo se protegen las variables de instancia de recibir valores inválidos, sino que también se oculta la representación interna de las variables de instancia de los clientes de esa clase. Por lo tanto, si cambia la representación de los datos (a menudo para reducir la cantidad de almacenamiento requerido o para mejorar el rendimiento), sólo necesitan cambiar las implementaciones de las propiedades; las implementaciones de los clientes no necesitan modificarse siempre y cuando se preserven los servicios que proporcionan las propiedades.

4.8 Comparación entre tipos por valor y tipos por referencia

En C#, los tipos se dividen en dos categorías: **por valor** y **por referencia**. Todos los tipos simples de C# son por valor; una variable de este tipo (como `int`) tan sólo contiene un valor de ese tipo. Por ejemplo, la figura 4.10 muestra una variable `int` llamada `conteo`, la cual contiene el valor 7.

En contraste, una variable de un tipo por referencia (también conocida como **referencia**) contiene la dirección de una ubicación en memoria, en donde se almacenan los datos a los que hace referencia. Se dice que dicha variable **hace referencia a un objeto** en el programa. La línea 11 de la figura 4.8 crea un objeto `LibroCalificaciones`, lo coloca en memoria y almacena la dirección de memoria de ese objeto en la variable de referencia `miLibroCalificaciones` de tipo `LibroCalificaciones`, como se muestra en la figura 4.11. Observe que el objeto `LibroCalificaciones` se muestra con su variable de instancia `nombreCurso`.

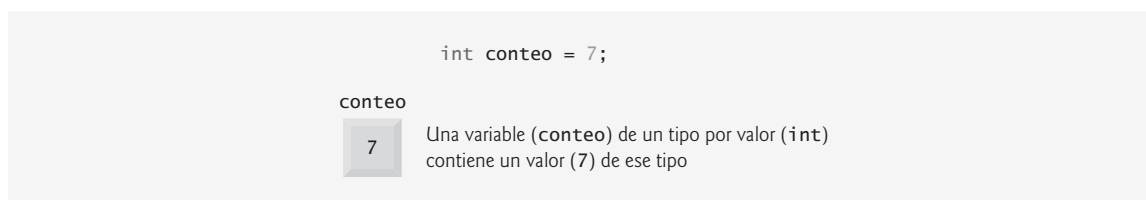


Figura 4.10 | Variable de tipo por valor.

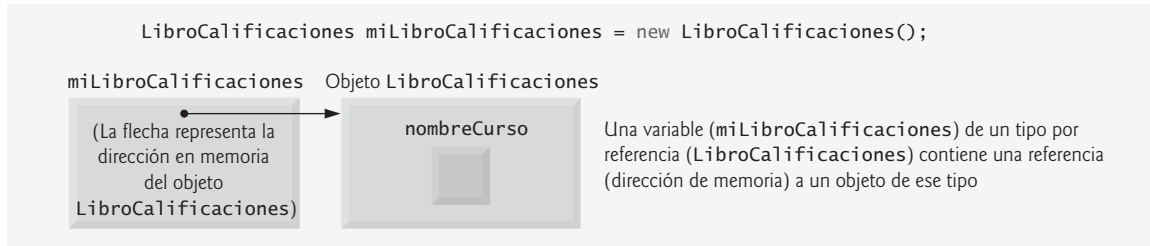


Figura 4.11 | Variable de tipo por referencia.

Las variables de instancia de tipo por referencia (como `miLibroCalificaciones` en la figura 4.11) se inicializan de manera predeterminada con el valor **null**. `string` es un tipo por referencia; por esta razón, la variable `string nombreCurso` se muestra en la figura 4.11 con un cuadro vacío, que representa a la variable con valor `null` en memoria.

Un cliente de un objeto debe utilizar una referencia a ese objeto para **invocar** (es decir, llamar) a los métodos del objeto y acceder a sus propiedades. En la figura 4.8, las instrucciones en `Main` utilizan la variable `miLibroCalificaciones`, que contiene la referencia al objeto `LibroCalificaciones`, para enviar mensajes al objeto `LibroCalificaciones`. Estos mensajes son llamadas a métodos (como `MostrarMensaje`) o referencias a propiedades (como `nombreCurso`), las cuales permiten al programa interactuar con objetos tipo `LibroCalificaciones`. Por ejemplo, la instrucción (en la línea 20 de la figura 4.8)

```
miLibroCalificaciones.NombreCurso = elNombre; // establece el nombre usando una propiedad
```

utiliza la referencia `miLibroCalificaciones` para establecer el nombre del curso mediante la asignación de un valor a la propiedad `NombreCurso`. Esto envía un mensaje al objeto `LibroCalificaciones` para que invoque al descriptor de acceso `set` de la propiedad `NombreCurso`. El mensaje incluye como argumento el valor `"CS101 Introducción a la programación en C#"` que requiere el descriptor de acceso `set` de `NombreCurso` para realizar su tarea. El descriptor de acceso `set` utiliza esta información para establecer la variable de instancia `nombreCurso`. En la sección 7.14 hablaremos con detalle acerca de los tipos por valor y los tipos por referencia.



Observación de ingeniería de software 4.4

El tipo declarado de una variable (por ejemplo, `int`, `double` o `LibroCalificaciones`) indica si la variable es de un tipo por valor o por referencia. Si el tipo de una variable no es uno de los trece tipos simples, o un tipo `enum` o `struct` (que veremos en la sección 7.10 y en el capítulo 16, en forma respectiva), entonces es un tipo por referencia. Por ejemplo, `Cuenta cuenta1` indica que `cuenta1` es una variable que puede hacer referencia a un objeto `Cuenta`.

4.9 Inicialización de objetos con constructores

Como mencionamos en la sección 4.5, cuando se crea un objeto de la clase `LibroCalificaciones` (figura 4.7), su variable de instancia `nombreCurso` se inicializa con `null` de manera predeterminada. ¿Qué pasa si usted desea proporcionar el nombre de un curso cuando crea un objeto `LibroCalificaciones`? Cada clase que usted declare puede proporcionar un **constructor**, que puede utilizarse para inicializar un objeto de una clase al momento de crear ese objeto. De hecho, C# requiere una llamada al constructor para cada objeto que se crea. El operador `new` llama al constructor de la clase para realizar la inicialización. La llamada al constructor se indica mediante el nombre de la clase, seguido de paréntesis. Por ejemplo, la línea 11 de la figura 4.8 primero utiliza `new` para crear un objeto `LibroCalificaciones`. Los paréntesis vacíos después de `"new LibroCalificaciones"` indican una llamada sin argumentos al constructor de la clase. De manera predeterminada, el compilador proporciona un **constructor predeterminado** sin parámetros, en cualquier clase que no incluya un constructor en forma explícita, por lo que *toda* clase tiene un constructor.

Cuando usted declara una clase, puede proporcionar su propio constructor para especificar una inicialización personalizada para los objetos de su clase. Por ejemplo, tal vez quiera especificar el nombre de un curso para un objeto `LibroCalificaciones` cuando se crea este objeto, como en


```

LibroCalificaciones miLibroCalificaciones =
    new LibroCalificaciones( "CS101 Introducción a la programación en C#" );

```

En este caso, el argumento "CS101 Introducción a la programación en C#" se pasa al constructor del objeto LibroCalificaciones y se utiliza para inicializar a nombreCurso. Cada vez que usted crea un objeto LibroCalificaciones distinto, puede proporcionar un nombre distinto para el curso. La instrucción anterior requiere que la clase proporcione un constructor con un parámetro string. La figura 4.12 contiene una clase LibroCalificaciones modificada con dicho constructor.

Las líneas 10-13 declaran el constructor para la clase LibroCalificaciones. Un constructor debe tener el mismo nombre que su clase. Al igual que un método, un constructor especifica en su lista de parámetros los datos que requiere para realizar su tarea. A diferencia de un método, un constructor no especifica un tipo de valor de retorno. Cuando usted crea un nuevo objeto (con new), coloca estos datos en los paréntesis que van después del nombre de la clase. La línea 10 indica que el constructor de la clase LibroCalificaciones tiene un parámetro llamado nombre, de tipo string. En la línea 12 del cuerpo del constructor, el nombre que se pasa al constructor se asigna a la variable de instancia nombreCurso mediante el descriptor de acceso set de la propiedad NombreCurso.

```

1 // Fig. 4.12: LibroCalificaciones.cs
2 // La clase LibroCalificaciones con un constructor para inicializar el nombre del curso.
3 using System;
4
5 public class LibroCalificaciones
6 {
7     private string nombreCurso; // nombre del curso para este LibroCalificaciones
8
9     // el constructor inicializa nombreCurso con el objeto string suministrado como
10    // argumento
11    public LibroCalificaciones( string nombre )
12    {
13        NombreCurso = nombre; // inicializa nombreCurso usando la propiedad
14    } // fin del constructor
15
16    // propiedad para obtener (get) y establecer (set) el nombre del curso
17    public string NombreCurso
18    {
19        get
20        {
21            return nombreCurso;
22        } // fin de get
23        set
24        {
25            nombreCurso = value;
26        } // fin de set
27    } // fin de la propiedad NombreCurso
28
29    // muestra un mensaje de bienvenida para el usuario del LibroCalificaciones
30    public void MostrarMensaje()
31    {
32        // usa la propiedad NombreCurso para obtener el
33        // nombre del curso que representa este LibroCalificaciones
34        Console.WriteLine( "Bienvenido al libro de calificaciones para\n{0}!",
35            NombreCurso );
36    } // fin del método MostrarMensaje
37 } // fin de la clase LibroCalificaciones

```

Figura 4.12 | La clase LibroCalificaciones con un constructor para inicializar el nombre del curso.

La figura 4.13 demuestra la inicialización de objetos `LibroCalificaciones` mediante el uso de este constructor. Las líneas 12-13 crean e inicializan un objeto `LibroCalificaciones`. El constructor de la clase `LibroCalificaciones` se llama con el argumento "CS101 Introducción a la programación en C#" para inicializar el nombre del curso. La expresión de creación de objeto a la derecha del signo `=` en las líneas 12-13 devuelve una referencia al nuevo objeto, el cual se asigna a la variable `libroCalificaciones1`. Las líneas 14-15 repiten este proceso para otro objeto `LibroCalificaciones`, pero esta vez pasan el argumento "CS102 Estructuras de datos en C#" para inicializar el nombre del curso para `libroCalificaciones2`. Las líneas 18-21 utilizan la propiedad `NombreCurso` de cada objeto para obtener los nombres de los cursos y mostrar que sin duda se inicializaron en el momento en el que se crearon. En la introducción a la sección 4.5, usted aprendió que cada instancia (es decir, objeto) de una clase contiene su propia copia de las variables de instancia de la clase. La salida confirma que cada `LibroCalificaciones` mantiene su propia copia de la variable de instancia `nombreCurso`.

Al igual que los métodos, los constructores también pueden recibir argumentos. No obstante, una importante diferencia entre los constructores y los métodos es que los primeros no pueden devolver valores; de hecho, no pueden especificar un tipo de valor de retorno (ni siquiera `void`). Por lo general, los constructores se declaran como `public`. Si una clase no incluye un constructor, las variables de instancia de esa clase se inicializan con sus valores predeterminados. Si usted declara uno o más constructores para una clase, C# no creará un constructor predeterminado para esa clase.



Tip de prevención de errores 4.1

A menos que sea aceptable la inicialización predeterminada de las variables de instancia de su clase, deberá proporcionar un constructor para asegurarse que las variables de instancia de su clase se inicialicen en forma apropiada con valores significativos, cuando se cree cada nuevo objeto de su clase.

```

1 // Fig. 4.13: PruebaLibroCalificaciones.cs
2 // El constructor LibroCalificaciones se utiliza para especificar el nombre del
3 // curso cada vez que se crea un objeto LibroCalificaciones.
4 using System;
5
6 public class PruebaLibroCalificaciones
7 {
8     // El método Main comienza la ejecución del programa
9     public static void Main( string[] args )
10    {
11        // crea el objeto LibroCalificaciones
12        LibroCalificaciones libroCalificaciones1 = new LibroCalificaciones( // invoca al
            constructor
13            "CS101 Introducción a la programación en C#" );
14        LibroCalificaciones libroCalificaciones2 = new LibroCalificaciones( // invoca al
            constructor
15            "CS102 Estructuras de datos en C#" );
16
17        // muestra el valor inicial de nombreCurso para cada LibroCalificaciones
18        Console.WriteLine( "El nombre del curso de libroCalificaciones1 es: {0}",
19            libroCalificaciones1.NombreCurso );
20        Console.WriteLine( "El nombre del curso de libroCalificaciones2 es: {0}",
21            libroCalificaciones2.NombreCurso );
22    } // fin de Main
23 } // fin de la clase PruebaLibroCalificaciones

```

```

El nombre del curso de libroCalificaciones1 es: CS101 Introducción a la programación en C#
El nombre del curso de libroCalificaciones2 es: CS102 Estructuras de datos en C#

```

Figura 4.13 | El constructor de `LibroCalificaciones` se utiliza para especificar el nombre del curso cada vez que se crea cada un objeto `LibroCalificaciones`.

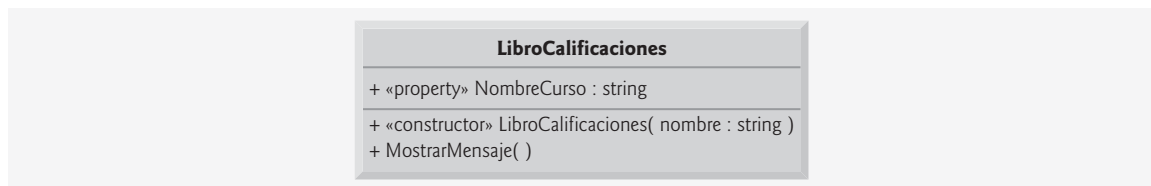


Figura 4.14 | Diagrama de clases de UML donde se indica que la clase `LibroCalificaciones` tiene un constructor con el parámetro `Nombre` del tipo `string`.

Agregar el constructor al diagrama de clases de UML de la clase `LibroCalificaciones`

El diagrama de clases de UML de la figura 4.14 modela la clase `LibroCalificaciones` de la figura 4.12, que tiene un constructor con un parámetro llamado `nombreCurso`, de tipo `string`. Al igual que las operaciones, el diagrama de clases UML modela a los constructores en el tercer compartimiento de una clase. Para diferenciar a un constructor de las operaciones de una clase, UML coloca la palabra “constructor” entre los signos « y » antes del nombre del constructor. Es costumbre listar los constructores antes de otras operaciones en el tercer compartimiento.

4.10 Los números de punto flotante y el tipo `decimal`

En nuestra siguiente aplicación dejaremos por un momento nuestro caso de estudio con la clase `LibroCalificaciones` para declarar una clase llamada `Cuenta`, que mantiene el saldo de una cuenta bancaria. La mayoría de los saldos de las cuentas no son números enteros (por ejemplo, 0, -22 y 1024). Por esta razón, la clase `Cuenta` representa el saldo de las cuentas como un número real (es decir, un número con un punto decimal, como 7.33, 0.0975 o 1000.12345). C# proporciona tres tipos simples para almacenar números reales en memoria: `float`, `double` y `decimal`. A los tipos `float` y `double` se les llama tipos de **punto flotante**. La principal diferencia entre ellos y `decimal` es que las variables tipo `decimal` almacenan un rango limitado de números reales con precisión, mientras que las variables de punto flotante sólo almacenan aproximaciones de números reales, pero a través de un rango mucho mayor de valores. Además, las variables `double` pueden almacenar números con mayor magnitud y detalle (es decir, más dígitos a la derecha del punto decimal; también se le conoce como la **precisión** del número) que las variables `float`. Una aplicación clave del tipo `decimal` es para representar cantidades monetarias.

Precisión de los números reales y requerimientos de memoria

Las variables de tipo `float` representan **números de punto flotante de precisión simple** y tienen siete dígitos significativos. Las variables de tipo `double` representan **números de punto flotante de precisión doble**. Estos requieren el doble de memoria que las variables `float` y proporcionan de 15 a 16 dígitos significativos; aproximadamente el doble de precisión de las variables `float`. Lo que es más, las variables de tipo `decimal` requieren hasta el doble de memoria que las variables `double` y proporcionan de 28 a 29 dígitos significativos. Para el rango de valores requeridos por la mayoría de las aplicaciones debe bastar con las variables de tipo `float` para las aproximaciones, pero podemos utilizar variables tipo `double` o `decimal` para estar seguros. En algunas aplicaciones, incluso hasta las variables de tipo `double` y `decimal` serán inadecuadas; dichas aplicaciones se encuentran más allá del alcance de este libro.

La mayoría de los programadores representa números de punto flotante con el tipo `double`. De hecho, C# trata a todos los números reales que uno escribe en el código fuente de una aplicación (como 7.33 y 0.0975) como valores `double` de manera predeterminada. Dichos valores en el código fuente se conocen como **literales de punto flotante**. Para escribir una **literal decimal**, debe escribir la letra “M” o “m” al final de un número real (por ejemplo, 7.33M es una literal `decimal` en vez de `double`). Las literales enteras se convierten de manera implícita en literales tipo `float`, `double` o `decimal` cuando se asignan a una variable de uno de estos tipos. En el apéndice L, Los tipos simples, podrá consultar los rangos de los valores para los tipos `float`, `double`, `decimal` y todos los demás tipos simples.

Aunque los números de punto flotante no son siempre 100% precisos, tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal “normal” de 36.8, no necesitamos una precisión con un número extenso de dígitos. Cuando leemos la temperatura en un termómetro como 36.8, en realidad podría ser 36.7999473210643. Si consideramos a este número simplemente como 36.8 está bien para la mayoría de las

aplicaciones en las que se trabaja con las temperaturas corporales. Debido a la naturaleza imprecisa de los números de punto flotante, se prefiere el tipo decimal a los tipos de punto flotante siempre que los cálculos necesitan ser exactos, como en los cálculos monetarios. En los casos en donde es suficiente con las aproximaciones, se prefiere el tipo double al tipo float ya que las variables double pueden representar los números de punto flotante con más precisión. Por esta razón, utilizaremos el tipo decimal a lo largo de este libro para tratar con cantidades monetarias y el tipo double para los demás números reales.

Los números reales también surgen como resultado de la división. Por ejemplo, en la aritmética convencional, cuando dividimos 10 entre 3 el resultado es 3.3333333..., y la secuencia de números 3 se repite en forma indefinida. La computadora asigna sólo una cantidad fija de espacio para almacenar un valor de este tipo, por lo que es evidente que el valor de punto flotante almacenado sólo puede ser una aproximación.



Error común de programación 4.3

El uso de números de punto flotante de una manera en la que se asuma que se representan con precisión puede producir errores lógicos.

La clase Cuenta con una variable de instancia de tipo decimal

Nuestra siguiente aplicación (figuras 4.15-4.16) contiene una clase muy simplificada llamada Cuenta (figura 4.15), que mantiene el saldo de una cuenta bancaria. Un banco ordinario da servicio a muchas cuentas, cada una con su propio balance, por lo que la línea 7 declara una variable de instancia llamada saldo de tipo decimal. La variable saldo es una variable de instancia, ya que está declarada en el cuerpo de la clase (líneas 6-36) pero fuera del método de la clase y de las declaraciones de las propiedades (líneas 10-13, 16-19 y 22-35). Cada instancia (es decir, objeto) de la clase Cuenta contiene su propia copia de saldo.

La clase Cuenta contiene un constructor, un método y una propiedad. Como es común que alguien abra una cuenta para depositar dinero de inmediato, el constructor (líneas 10-13) recibe un parámetro llamado saldoInicial de tipo decimal, que representa el saldo inicial de la cuenta. La línea 12 asigna saldoInicial a la propiedad Saldo, con lo cual invoca el descriptor de acceso set de Saldo para inicializar la variable de instancia saldo.

El método Acredita (líneas 16-19) no devuelve ningún dato cuando completa su tarea, por lo que su tipo de valor de retorno es void. El método recibe un parámetro llamado monto: un valor decimal que se suma a la propiedad Saldo. La línea 18 utiliza los descriptors de acceso get y set de Saldo. La expresión Saldo + monto invoca al descriptor de acceso get de Saldo para obtener el valor actual de la variable de instancia saldo y después le suma la variable monto. Posteriormente asignamos el resultado a la variable de instancia saldo mediante la invocación del descriptor de acceso set de la propiedad Saldo (con lo cual se sustituye el valor anterior de saldo).

La propiedad Saldo (líneas 22-35) cuenta con un descriptor de acceso get, que permite a los clientes de la clase (es decir, otras clases que utilicen esta clase) obtener el valor del saldo del objeto de una Cuenta específica. La propiedad tiene el tipo decimal (línea 22). Saldo también cuenta con un descriptor de acceso set mejorado.

En la sección 4.5 presentamos las propiedades cuyos descriptors de acceso set permiten a los clientes de la clase modificar el valor de una variable de instancia private. En la figura 4.7, la clase LibroCalificaciones define el descriptor de acceso set de la propiedad NombreCurso para asignar el valor que recibe en su parámetro value a la variable de instancia nombreCurso (línea 19). Esta propiedad NombreCurso no asegura que nombreCurso sólo contenga datos válidos.

La aplicación de las figuras 4.15 a la 4.16 mejora el descriptor de acceso set de la propiedad Saldo de la clase Cuenta para realizar esta validación (lo que también se conoce como **comprobación de validez**). La línea 32 (figura 4.15) asegura que el valor no sea negativo. Si el valor es mayor o igual a 0, el monto almacenado en value se asigna a la variable de instancia saldo en la línea 33. De no ser así, saldo permanece sin cambios.

La clase PruebaCuenta que utiliza a la clase Cuenta

La clase PruebaCuenta (figura 4.16) crea dos objetos Cuenta (líneas 10-11) y los inicializa en forma respectiva con 50.00M y -7.53M (las literales decimales que representan los números reales 50.00 y -7.53). Observe que el constructor de Cuenta (líneas 10-13 de la figura 4.15) hace referencia a la propiedad Saldo para inicializar

```

1  // Fig. 4.15: Cuenta.cs
2  // La clase Cuenta con un constructor para
3  // inicializar la variable de instancia saldo.
4
5  public class Cuenta
6  {
7      private decimal saldo; // variable de instancia que almacena el saldo
8
9      // constructor
10     public Cuenta( decimal saldoInicial )
11     {
12         Saldo = saldoInicial; // establece el saldo usando la propiedad
13     } // fin del constructor de Cuenta
14
15     // acredita (suma) un monto a la cuenta
16     public void Acredita( decimal monto )
17     {
18         Saldo = Saldo + monto; // suma monto al saldo
19     } // fin del método Acredita
20
21     // una propiedad para obtener (get) y establecer (set) el saldo de una cuenta
22     public decimal Saldo
23     {
24         get
25         {
26             return saldo;
27         } // end get
28         set
29         {
30             // valida que el valor sea mayor o igual a 0;
31             // si no lo es, el saldo permanece sin cambios
32             if ( value >= 0 )
33                 saldo = value;
34         } // fin de set
35     } // fin de la propiedad Saldo
36 } // fin de la clase Cuenta

```

Figura 4.15 | La clase Cuenta con un constructor para inicializar la variable de instancia saldo.

saldo. En ejemplos anteriores, el beneficio de hacer referencia a la propiedad en el constructor no era evidente. Sin embargo, ahora el constructor aprovecha la validación que proporciona el descriptor de acceso set de la propiedad Saldo. Y sólo asigna un valor a Saldo, en vez de duplicar el código de validación del descriptor de acceso set. Cuando la línea 11 de la figura 4.16 pasa un saldo inicial de -7.53 al constructor de Cuenta, el constructor pasa este valor al descriptor de acceso set de la propiedad Saldo, en donde ocurre la inicialización en sí. Este valor es menor que 0, por lo que el descriptor de acceso set no modifica a saldo y deja a esta variable de instancia con su valor predeterminado de 0.

Las líneas de la 14 a la 17 en la figura 4.16 imprimen en pantalla el saldo en cada Cuenta mediante el uso de la propiedad Saldo de Cuenta. Cuando se utiliza Saldo para cuenta1 (línea 15), el descriptor de acceso get devuelve el valor del saldo de cuenta1 en la línea 26 de la figura 4.15 y la instrucción Console.WriteLine lo muestra en pantalla (figura 4.16, líneas 14-15). De manera similar, cuando se llama a la propiedad Saldo para cuenta2 desde la línea 17, se devuelve el valor del saldo de cuenta2 desde la línea 26 de la figura 4.15 y la instrucción Console.WriteLine la imprime en pantalla (figura 4.16, líneas 16-17). Observe que el saldo de cuenta2 es 0, ya que el constructor se aseguró que la cuenta no pudiera comenzar con un saldo negativo. El valor se imprime en pantalla mediante WriteLine con el elemento de formato {0:C}, el cual da formato al saldo de la cuenta como una cantidad monetaria. El signo : después del 0 indica que el siguiente carácter representa un *especificador de formato*, y el especificador de formato C antes del signo : especifica una cantidad monetaria (C es para moneda).

```

1 // Fig. 4.16: PruebaCuenta.cs
2 // Creación y manipulación de un objeto Cuenta.
3 using System;
4
5 public class PruebaCuenta
6 {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10         Cuenta cuenta1 = new Cuenta( 50.00M ); // crea el objeto Cuenta
11         Cuenta cuenta2 = new Cuenta( -7.53M ); // crea el objeto Cuenta
12
13         // muestra el saldo inicial de cada objeto usando una propiedad
14         Console.WriteLine( "Saldo de cuenta1: {0:C}",
15             cuenta1.Saldo ); // muestra la propiedad Saldo
16         Console.WriteLine( "Saldo de cuenta2: {0:C}\n",
17             cuenta2.Saldo ); // muestra la propiedad Saldo
18
19         decimal montoDeposito; // deposita la cantidad que se leyó del usuario
20
21         // pide y obtiene la entrada del usuario
22         Console.Write( "Escriba el monto a depositar para la cuenta1: " );
23         montoDeposito = Convert.ToDecimal( Console.ReadLine() );
24         Console.WriteLine( "se sumó {0:C} al saldo de cuenta1\n",
25             montoDeposito );
26         cuenta1.Acuenta( montoDeposito ); // se suma al saldo de cuenta1
27
28         // muestra los saldos
29         Console.WriteLine( "Saldo de cuenta1: {0:C}",
30             cuenta1.Saldo );
31         Console.WriteLine( "Saldo de cuenta2: {0:C}\n",
32             cuenta2.Saldo );
33
34         // pide y obtiene la entrada del usuario
35         Console.Write( "Escriba la cantidad a depositar para la cuenta2: " );
36         montoDeposito = Convert.ToDecimal( Console.ReadLine() );
37         Console.WriteLine( "se sumó {0:C} al saldo de cuenta2\n",
38             montoDeposito );
39         cuenta2.Acuenta( montoDeposito ); // se suma al saldo de cuenta2
40
41         // muestra los saldos
42         Console.WriteLine( "Saldo de cuenta1: {0:C}", cuenta1.Saldo );
43         Console.WriteLine( "Saldo de cuenta2: {0:C}", cuenta2.Saldo );
44     } // fin de Main
45 } // fin de la clase PruebaCuenta

```

```

Saldo de cuenta1: $50.00
Saldo de cuenta2: $0.00

Escriba el monto a depositar para la cuenta1: 49.99
se sumó $49.99 al saldo de cuenta1

Saldo de cuenta1: $99.99
Saldo de cuenta2: $0.00

Escriba la cantidad a depositar para la cuenta2: 123.21
se sumó $123.21 al saldo de cuenta2

Saldo de cuenta1: $99.99
Saldo de cuenta2: $123.21

```

Figura 4.16 | Creación y manipulación de un objeto Cuenta.

Especificador de formato	Descripción
C o c	Da formato al objeto string como moneda. Coloca antes del número un símbolo de moneda apropiado (\$) en Estados Unidos y en México). Separa los dígitos con un carácter separador apropiado (coma en Estados Unidos y en México) y establece el número de caracteres decimales en 2, de manera predeterminada.
D o d	Da formato al objeto string como decimal. Muestra el número como un entero.
N o n	Da formato al objeto string con comas y un valor predeterminado de dos lugares decimales.
E o e	Da formato al objeto string utilizando notación científica, con un valor predeterminado de seis lugares decimales.
F o f	Da formato al objeto string con un número fijo de lugares decimales (el valor predeterminado es de dos).
G o g	General. Por lo general da formato al número con lugares decimales o utilizando notación científica, dependiendo del contexto. Si un elemento de formato no contiene un especificador de formato, se asume el formato G de manera implícita.
X o x	Da formato al objeto string como hexadecimal.

Figura 4.17 | Especificadores de formato para objetos string.

Las configuraciones culturales en el equipo del usuario determinan el formato para mostrar cantidades monetarias. Por ejemplo, en Estados Unidos, 50 se muestra como \$50.00; mientras que en Alemania, como 50,00€. La figura 4.17 muestra algunos especificadores de formato, además de C.

La línea 19 declara la variable local `montoDeposito` para almacenar cada monto a depositar que escriba el usuario. A diferencia de la variable de instancia `saldo` en la clase `Cuenta`, la variable local `montoDeposito` en `Main` no se inicializa con 0 de manera predeterminada. No obstante, esta variable no necesita inicializarse aquí, ya que su valor se determinará en base a la entrada del usuario.

La línea 22 pide al usuario que introduzca un monto a depositar para `cuenta1`. La línea 23 obtiene la entrada del usuario mediante una llamada al método `ReadLine` de la clase `Console`, y después pasa el objeto string que introdujo el usuario al método `ToDecimal` de la clase `Convert`, el cual devuelve el valor `decimal` en este objeto string. Las líneas 24-25 muestran en pantalla el monto de depósito. La línea 26 llama al método `Acredita` del objeto `cuenta1` y suministra `montoDeposito` como argumento para el método. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro `monto` del método `Acredita` (líneas 16-19 de la figura 4.15) y después este método suma ese valor al `saldo` (línea 18 de la figura 4.15). Las líneas 29-32 (figura 4.16) imprimen en pantalla los saldos de ambos objetos `Cuenta` otra vez, para mostrar que sólo cambió el saldo de `cuenta1`.

La línea 35 pide al usuario que introduzca un monto a depositar en `cuenta2`. La línea 36 obtiene la entrada del usuario mediante una llamada al método `ReadLine` de la clase `Console` y pasa el valor de retorno al método `ToDecimal` de la clase `Convert`. Las líneas 37-38 muestran en pantalla el monto de depósito. La línea 39 llama al método `Acredita` del objeto `cuenta2` y suministra `montoDeposito` como argumento para el método, después el método `Acredita` suma ese valor al saldo. Por último, las líneas 42-43 imprimen en pantalla los saldos de ambos objetos `Cuenta` otra vez, para mostrar que sólo cambió el saldo de `cuenta2`.

Descriptores set y get con distintos modificadores de acceso

De manera predeterminada, los descriptores de acceso `get` y `set` de una propiedad tienen el mismo acceso que la propiedad; por ejemplo, para una propiedad `public`, los descriptores de acceso son `public`. Es posible declarar los descriptores de acceso `get` y `set` con distintos modificadores de acceso. En este caso, uno de los descriptores de acceso debe tener de manera implícita el mismo acceso que la propiedad, y el otro debe declararse con un modificador de acceso más restrictivo que el de la propiedad. Por ejemplo, en una propiedad `public`, el descriptor de acceso `get` podría ser `public` y el descriptor de acceso `set` podría ser `private`. En la sección 9.6 demostraremos esta característica.

**Tip de prevención de errores 4.2**

Los beneficios de la integridad de datos no son automáticos sólo porque las variables de instancia sean `private`; usted debe proporcionar una comprobación de validez apropiada y reportar los errores.

**Tip de prevención de errores 4.3**

Los descriptores de acceso `set` que establecen los valores de datos `private` deben verificar que los nuevos valores deseados sean apropiados; si no lo son, las variables de instancia deben permanecer sin cambio y se genera un error. En el capítulo 12, Manejo de excepciones, demostraremos cómo generar errores con sutileza.

Diagrama de clases de UML para la clase Cuenta

El diagrama de clases de UML en la figura 4.18 modela la clase Cuenta de la figura 4.15. El diagrama modela la propiedad Saldo como un atributo de UML de tipo decimal (ya que la propiedad correspondiente en C# tiene el tipo decimal). El diagrama modela el constructor de la clase Cuenta con un parámetro saldoInicial de tipo decimal en el tercer compartimiento de la clase. El diagrama modela la operación Acredita en el tercer compartimiento con un parámetro monto de tipo decimal (ya que el método correspondiente tiene un parámetro monto de tipo decimal en C#).

4.11 (Opcional) Caso de estudio de ingeniería de software: identificación de las clases en el documento de requerimientos del ATM

Ahora empezaremos a diseñar el sistema ATM que presentamos en el capítulo 3. En esta sección identificaremos las clases necesarias para crear el sistema ATM; para ello es necesario analizar los sustantivos y las frases nominales que aparecen en el documento de requerimientos. Introducimos los diagramas de clases de UML para modelar las relaciones entre estas clases. Este primer paso es importante para definir la estructura de nuestro sistema.

Identificación de las clases en un sistema

Para comenzar nuestro proceso de diseño orientado a objetos (DOO), identificaremos las clases requeridas para crear el sistema ATM. Más adelante describiremos estas clases mediante el uso de los diagramas de clases de UML y las implementaremos en C#. Primero debemos revisar el documento de requerimientos de la sección 3.10, para encontrar sustantivos y frases nominales clave que nos ayuden a identificar las clases que conformarán el sistema ATM. Tal vez decidamos que algunos de estos sustantivos y frases nominales sean atributos de otras clases en el sistema. Tal vez también concluyamos que algunos de los sustantivos y frases nominales no corresponden a ciertas partes del sistema y, por ende, no deben modelarse. A medida que avancemos por el proceso de diseño podemos ir descubriendo clases adicionales. La figura 4.19 lista los sustantivos y frases nominales en el documento de requerimientos.

Crearemos clases sólo para los sustantivos y frases nominales que tengan importancia en el sistema ATM. No necesitamos modelar “banco” como una clase, ya que el banco no es una parte del sistema ATM; el banco sólo quiere que nosotros construyamos el ATM. “Usuario” y “cliente” también representan entidades fuera del sistema; son importantes debido a que interactúan con nuestro sistema ATM, pero no necesitamos modelarlos como clases en el sistema ATM. Recuerde que modelamos un usuario del ATM (es decir, un cliente bancario) como el actor en el diagrama de caso-uso de la figura 3.31.

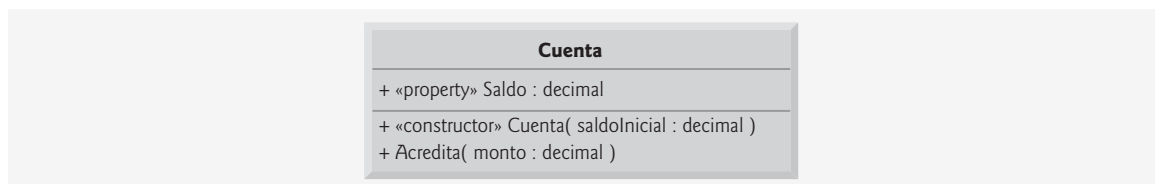


Figura 4.18 | Diagrama de clases de UML que indica que la clase Cuenta tiene una propiedad pública llamada Saldo de tipo decimal, un constructor y un método.

Sustantivos y frases nominales en el documento de requerimientos		
banco	dinero / fondos	número de cuenta
ATM	pantalla	NIP
usuario	teclado numérico	base de datos del banco
cliente	dispensador de efectivo	solicitud de saldo
transacción	billete de \$20 / efectivo	retiro
cuenta	ranura de depósito	depósito
saldo	sobre de depósito	

Figura 4.19 | Sustantivos y frases nominales en el documento de requerimientos.

No necesitamos modelar “billete de \$20” ni “sobre de depósito” como clases. Éstos son objetos físicos en el mundo real, pero no forman parte de lo que se automatizará. Podemos representar en forma adecuada la presencia de billetes en el sistema, mediante el uso de un atributo de la clase que modela el dispensador de efectivo (en la sección 5.12 asignaremos atributos a las clases). Por ejemplo, el dispensador de efectivo mantiene un conteo del número de billetes que contiene. El documento de requerimientos no dice nada acerca de lo que debe hacer el sistema con los sobres de depósito después de recibirlos. Podemos suponer que con sólo admitir la recepción de un sobre (una *operación* que realiza la clase que modela la ranura de depósito) es suficiente para representar la presencia de un sobre en el sistema (en la sección 7.15 asignaremos operaciones a las clases).

En nuestro sistema ATM simplificado, lo más apropiado sería representar varios montos de “dinero”, incluyendo el “saldo” de una cuenta, como atributos de otras clases. De igual forma, los sustantivos “número de cuenta” y “NIP” representan piezas importantes de información en el sistema ATM. Son atributos importantes de una cuenta bancaria. Sin embargo, no ofrecen comportamientos. Por ende, podemos modelarlos de la manera más apropiada como atributos de una clase de cuenta.

Aunque con frecuencia el documento de requerimientos describe una “transacción” en un sentido general, no modelaremos la amplia noción de una transacción financiera en este momento. En vez de ello, modelaremos los tres tipos de transacciones (es decir, “solicitud de saldo”, “retiro” y “depósito”) como clases individuales. Estas clases poseen los atributos específicos necesarios para ejecutar las transacciones que representan. Por ejemplo, para un retiro se necesita conocer el monto de dinero que el usuario desea retirar. Sin embargo, una solicitud de saldo no requiere datos adicionales. Lo que es más, las tres clases de transacciones exhiben comportamientos únicos. Para un retiro se requiere entregar efectivo al usuario, mientras que para un depósito se requiere recibir un sobre de depósito del usuario. [Nota: en la sección 11.9, “factorizaremos” las características comunes de todas las transacciones en una clase de “transacción” general, mediante el uso de los conceptos orientados a objetos de las clases abstractas y la herencia.]

Determinaremos las clases para nuestro sistema con base en los sustantivos y frases nominales restantes de la figura 4.19. Cada una de ellas se refiere a uno o varios de los siguientes elementos:

- ATM.
- pantalla.
- teclado numérico.
- dispensador de efectivo.
- ranura de depósito.
- cuenta.
- base de datos del banco.
- solicitud de saldo.
- retiro.
- depósito.

Es probable que los elementos de esta lista sean clases que necesitaremos implementar en nuestro sistema, aunque es demasiado pronto en nuestro proceso de diseño como para decir que la lista está completa.

Ahora podemos modelar las clases en nuestro sistema, con base en la lista que hemos creado. En el proceso de diseño escribimos los nombres de las clases con la primera letra en mayúscula (una convención de UML), como lo haremos cuando escribamos el código de C# para implementar nuestro diseño. Si el nombre de una clase contiene más de una palabra, juntaremos todas las palabras y escribiremos la primera letra de cada una de ellas en mayúscula (por ejemplo, NombreConVariasPalabras). Con estas convenciones, crearemos las clases ATM, Pantalla, Teclado, DispensadorEfectivo, RanuraDeposito, Cuenta, BaseDatosBanco, SolicitudSaldo, Retiro y Deposito. Construiremos nuestro sistema mediante el uso de todas estas clases como bloques de construcción. Sin embargo, antes de empezar a construir el sistema, debemos comprender mejor la forma en que las clases se relacionan entre sí.

Modelado de las clases

UML nos permite modelar, a través de los **diagramas de clases**, las clases en el sistema ATM y sus interrelaciones. La figura 4.20 representa a la clase ATM. En UML, cada clase se modela como un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado en forma horizontal y en negrita. El compartimiento de en medio contiene los atributos de la clase (en las secciones 5.12 y 6.9 hablaremos sobre los atributos). El compartimiento inferior contiene las operaciones de la clase (que veremos en la sección 7.15). En la figura 4.20 los compartimientos de en medio e inferior están vacíos, ya que no hemos determinado los atributos y operaciones de esta clase todavía.

Los diagramas de clases también muestran las relaciones entre las clases del sistema. La figura 4.21 muestra cómo nuestras clases ATM y Retiro se relacionan una con la otra. Por el momento modelaremos sólo este subconjunto de las clases del ATM, por cuestión de simpleza. Más adelante en esta sección, presentaremos un diagrama de clases más completo. Observe que los rectángulos que representan a las clases en este sistema no están subdivididos en compartimientos. UML permite suprimir los atributos y las operaciones de una clase de esta forma, cuando sea apropiado, para crear diagramas más legibles. Un diagrama de este tipo se denomina **diagrama con elementos omitidos (elided diagram)**: tanto su información como el contenido de los compartimientos segundo y tercero, no se modela. En las secciones 5.12 y 7.15 colocaremos información en estos compartimientos.

En la figura 4.21, la línea sólida que conecta a las dos clases representa una **asociación**: una relación entre clases. Los números cerca de cada extremo de la línea son valores de **multiplicidad**; éstos indican cuántos objetos de cada clase participan en la asociación. En este caso, al seguir la línea de un extremo al otro se revela que, en un momento dado, un objeto ATM participa en una asociación con cero o con un objeto Retiro; cero si el usuario actual no está realizando una transacción o si ha solicitado un tipo distinto de transacción, y uno si el usuario ha solicitado un retiro. UML puede modelar muchos tipos de multiplicidad. La figura 4.22 explica los tipos de multiplicidad.

Una asociación puede tener nombre. Por ejemplo, la palabra Ejecuta por encima de la línea que conecta a las clases ATM y Retiro en la figura 4.21 indica el nombre de esa asociación. Esta parte del diagrama se lee así: “un objeto de la clase ATM ejecuta cero o un objeto de la clase Retiro”. Los nombres de las asociaciones son direc-

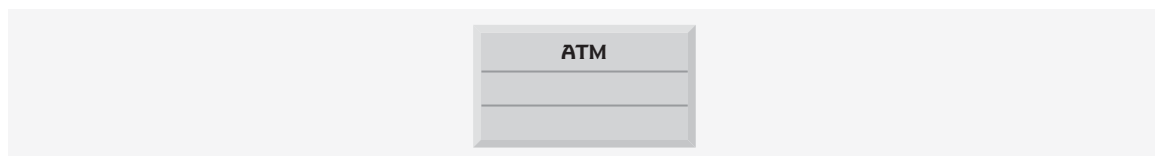


Figura 4.20 | Representación de una clase en UML mediante un diagrama de clases.

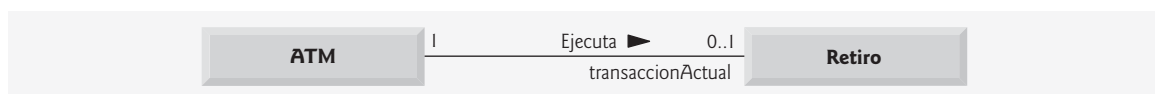


Figura 4.21 | Diagrama de clases que muestra una asociación entre clases.

Símbolo	Descripción
0	Ninguno
1	Uno
m	Un valor entero
0..1	Cero o uno
m, n	m o n
$m..n$	Cuando menos m , pero no más que n
*	Cualquier entero no negativo (cero o más)
0..*	Cero o más (idéntico a *)
1..*	Uno o más

Figura 4.22 | Tipos de multiplicidad.

cionales, como lo indica la punta de flecha rellena; por lo tanto, sería inapropiado, por ejemplo, leer la anterior asociación de derecha a izquierda como “cero o un objeto de la clase Retiro ejecuta un objeto de la clase ATM”.

La palabra `transaccionActual` en el extremo de Retiro de la línea de asociación en la figura 4.21 es un **nombre de rol**, el cual identifica el rol que desempeña el objeto Retiro en su relación con el ATM. Un nombre de rol agrega significado a una asociación entre clases, ya que identifica el rol que desempeña una clase dentro del contexto de una asociación. Una clase puede desempeñar varios roles en el mismo sistema. Por ejemplo, en un sistema de personal de una universidad, una persona puede desempeñar el rol de “profesor” con respecto a los estudiantes. La misma persona puede desempeñar el rol de “colega” cuando participa en una relación con otro profesor, y de “entrenador” cuando entrena a los atletas estudiantes. En la figura 4.21, el nombre de rol `transaccionActual` indica que el objeto Retiro que participa en la asociación Ejecuta con un objeto de la clase ATM representa a la transacción que está procesando el ATM en ese momento. En otros contextos, un objeto Retiro puede desempeñar otros roles (por ejemplo, la transacción anterior). Observe que no especificamos un nombre de rol para el extremo del ATM de la asociación Ejecuta. A menudo, los nombres de los roles se omiten en los diagramas de clases cuando el significado de una asociación está claro sin ellos.

Además de indicar relaciones simples, las asociaciones pueden especificar relaciones más complejas, como cuando los objetos de una clase están compuestos de objetos de otras clases. Considere un cajero automático real. ¿Qué “piezas” reúne un fabricante para construir un ATM funcional? Nuestro documento de requerimientos nos indica que el ATM está compuesto de una pantalla, un teclado, un dispensador de efectivo y una ranura de depósito.

En la figura 4.23, los **diamantes sólidos** que se adjuntan a las líneas de asociación de la clase ATM indican que esta clase tiene una relación de **composición** con las clases Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito. La composición implica una relación en todo/en parte. La clase que tiene el símbolo de composición (el diamante sólido) en su extremo de la línea de asociación es el todo (en este caso, ATM) y las clases en el otro extremo de las líneas de asociación son las partes; en este caso, las clases Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito. Las composiciones en la figura 4.23 indican que un objeto de la clase ATM está formado de un objeto de la clase Pantalla, un objeto de la clase DispensadorEfectivo, un objeto de la clase Teclado y un objeto de la clase RanuraDeposito; el ATM “tiene una” pantalla, un teclado, un dispensador de efectivo y una ranura de depósito. La **relación “tiene un”** define la composición (en la sección del Caso de estudio de ingeniería de software del capítulo 11 veremos que la relación “es un” define la herencia).

De acuerdo con la especificación del UML, las relaciones de composición tienen las siguientes propiedades:

1. Sólo una clase en la relación puede representar el todo (es decir, el diamante puede colocarse sólo en un extremo de la línea de asociación). Por ejemplo, la pantalla es parte del ATM o el ATM es parte de la pantalla, pero la pantalla y el ATM no pueden representar ambos el todo dentro de la relación.
2. Las partes en la relación de composición existen sólo mientras exista el todo, y el todo es responsable de la creación y destrucción de sus partes. Por ejemplo, el acto de construir un ATM incluye la manufactu-

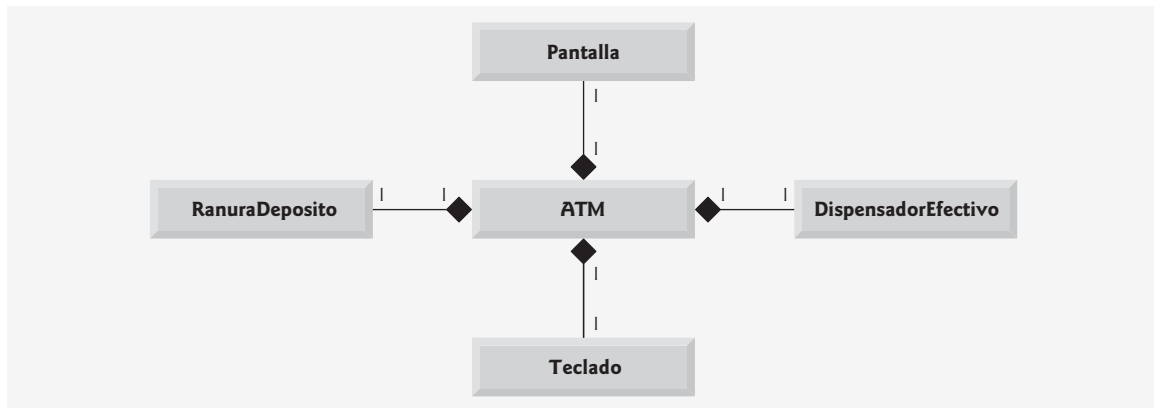


Figura 4.23 | Diagrama de clases que muestra las relaciones de composición.

ra de sus partes. Lo que es más, si el ATM se destruye, también se destruyen su pantalla, teclado, dispensador de efectivo y ranura de depósito.

3. Una parte puede pertenecer sólo a un todo a la vez, aunque esa parte puede quitarse y unirse a otro todo, el cual entonces asumirá la responsabilidad de esa parte.

Los diamantes sólidos en nuestros diagramas de clases indican las relaciones de composición que cumplen con estas tres propiedades. Si una relación “es un” no satisface uno o más de estos criterios, UML especifica que se deben adjuntar diamantes sin relleno a los extremos de las líneas de asociación para indicar una **agregación**: una forma más débil de la composición. Por ejemplo, una computadora personal y un monitor de computadora participan en una relación de agregación: la computadora “tiene un” monitor, pero las dos partes pueden existir en forma independiente, y el mismo monitor puede conectarse a varias computadoras a la vez, con lo cual se violan las propiedades segunda y tercera de la composición.

La figura 4.24 muestra un diagrama de clases para el sistema ATM. Este diagrama modela la mayoría de las clases que identificamos antes en esta sección, así como las asociaciones entre ellas que podemos inferir del documento de requerimientos. [Nota: las clases *SolicitudSaldo* y *Deposito* participan en asociaciones similares a las de la clase *Retiro*, por lo que preferimos omitirlas en este diagrama por cuestión de simpleza. En el capítulo 11 expandiremos nuestro diagrama de clases para incluir todas las clases en el sistema ATM.]

La figura 4.24 presenta un modelo gráfico de la estructura del sistema ATM. Este diagrama de clases incluye las clases *BaseDatosBanco* y *Cuenta*, junto con varias asociaciones que no presentamos en las figuras 4.21 o 4.23. El diagrama de clases muestra que la clase *ATM* tiene una **relación de uno a uno** con la clase *BaseDatosBanco*: un objeto *ATM* autentica a los usuarios contra un objeto *BaseDatosBanco*. En la figura 4.24 también modelamos el hecho de que la base de datos del banco contiene información sobre muchas cuentas; un objeto de la clase *BaseDatosBanco* participa en una relación de composición con cero o más objetos de la clase *Cuenta*. Recuerde que en la figura 4.22 se muestra que el valor de multiplicidad $0..*$ en el extremo de la clase *Cuenta* de la asociación entre las clases *BaseDatosBanco* y *Cuenta* indica que cero o más objetos de la clase *Cuenta* participan en la asociación. La clase *BaseDatosBanco* tiene una **relación de uno a varios** con la clase *Cuenta*; *BaseDatosBanco* puede contener muchos objetos *Cuenta*. De manera similar, la clase *Cuenta* tiene una **relación de varios a uno** con la clase *BaseDatosBanco*; puede haber muchos objetos *Cuenta* en *BaseDatosBanco*. Si recuerda la figura 4.22, el valor de multiplicidad $*$ es idéntico a $0..*$.

La figura 4.24 también indica que si el usuario va a realizar un retiro, “un objeto de la clase *Retiro* accede a/modifica un saldo de cuenta a través de un objeto de la clase *BaseDatosBanco*”. Podríamos haber creado una asociación en forma directa entre la clase *Retiro* y la clase *Cuenta*. No obstante, el documento de requerimientos indica que el “ATM debe interactuar con la base de datos de información de las cuentas del banco” para realizar transacciones. Una cuenta de banco contiene información delicada, por lo que los ingenieros de sistemas deben considerar siempre la seguridad de los datos personales al diseñar un sistema. Por ello, sólo *BaseDatosBanco* puede acceder a una cuenta y manipularla en forma directa. Todas las demás partes del sistema deben

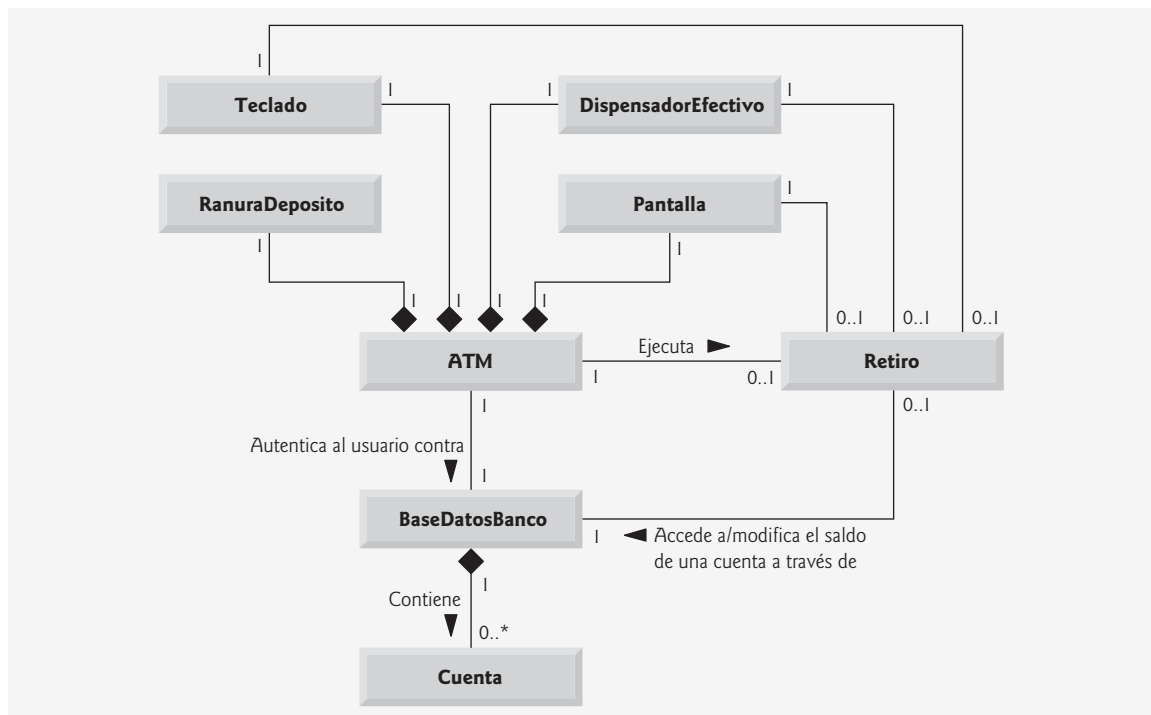


Figura 4.24 | Diagrama de clases para el modelo del sistema ATM.

interactuar con la base de datos para recuperar o actualizar la información de las cuentas (por ejemplo, el saldo de una cuenta).

El diagrama de clases de la figura 4.24 también modela las asociaciones entre la clase *Retiro* y las clases *Pantalla*, *DispensadorEfectivo* y *TecLado*. Una transacción de retiro implica pedir al usuario que seleccione el monto a retirar; también implica recibir entrada numérica. Estas acciones requieren el uso de la pantalla y del teclado, en forma respectiva. Para entregar efectivo al usuario se requiere acceso al dispensador de efectivo.

Aunque no se muestran en la figura 4.24, las clases *SolicitudSaldo* y *Deposito* participan en varias asociaciones con las otras clases del sistema ATM. Al igual que la clase *Retiro*, cada una de estas clases se asocia con las clases *ATM* y *BaseDatosBanco*. Un objeto de la clase *SolicitudSaldo* también se asocia con un objeto de la clase *Pantalla* para mostrar al usuario el saldo de una cuenta. La clase *Deposito* se asocia con las clases *Pantalla*, *TecLado* y *RanuraDeposito*. Al igual que los retiros, las transacciones de depósito requieren el uso de la pantalla y el teclado para mostrar mensajes y recibir entradas, en forma respectiva. Para recibir un sobre de depósito, un objeto de la clase *Deposito* se asocia con un objeto de la clase *RanuraDeposito*.

Hemos identificado las clases en nuestro sistema ATM, aunque tal vez descubramos otras a medida que avancemos con el diseño y la implementación. En la sección 5.12 determinaremos los atributos para cada una de estas clases, y en la sección 6.9 utilizaremos estos atributos para examinar la forma en que cambia el sistema con el tiempo. En la sección 7.15 determinaremos las operaciones de las clases en nuestro sistema.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

4.1 Suponga que tenemos una clase llamada *Auto*, la cual representa a un auto. Piense en algunas de las distintas piezas que podría reunir un fabricante para producir un auto completo. Cree un diagrama de clases (similar a la figura 4.23) que modele algunas de las relaciones de composición de la clase *Auto*.

4.2 Suponga que tenemos una clase llamada *Archivo*, la cual representa un documento electrónico en una computadora independiente, sin conexión de red, representada por la clase *Computadora*. ¿Qué tipo de asociación existe entre la clase *Computadora* y la clase *Archivo*?

a) La clase *Computadora* tiene una relación de uno a uno con la clase *Archivo*.

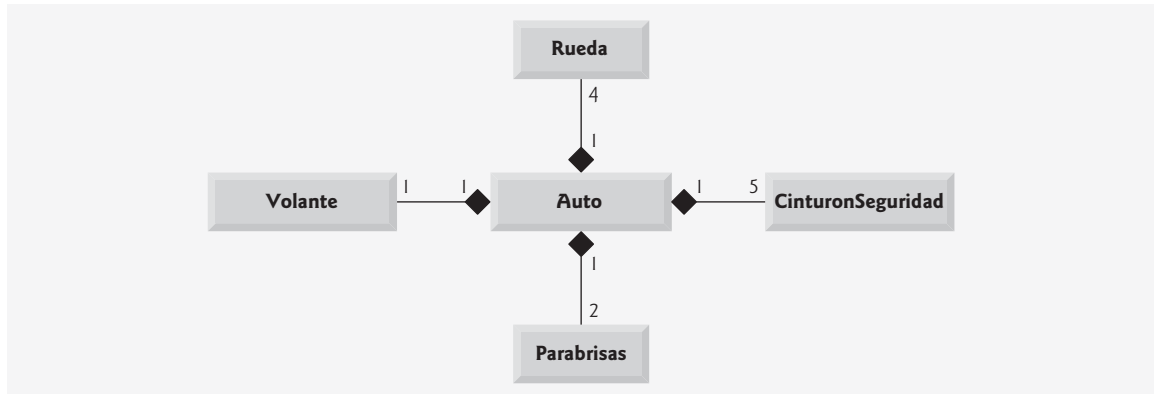


Figura 4.25 | Diagrama de clases que muestra algunas relaciones de composición de una clase **Auto**.

- b) La clase **Computadora** tiene una relación de varios a uno con la clase **Archivo**.
- c) La clase **Computadora** tiene una relación de uno a varios con la clase **Archivo**.
- d) La clase **Computadora** tiene una relación de varios a varios con la clase **Archivo**.

4.3 Indique si la siguiente aseveración es *verdadera* o *falsa*. Si es *falsa*, explique por qué: un diagrama de clases de UML en el que no se modelan los compartimientos segundo y tercero se denomina diagrama con elementos omitidos (elided diagram).

4.4 Modifique el diagrama de clases de la figura 4.24 para incluir la clase **Deposito**, en vez de la clase **Retiro**.

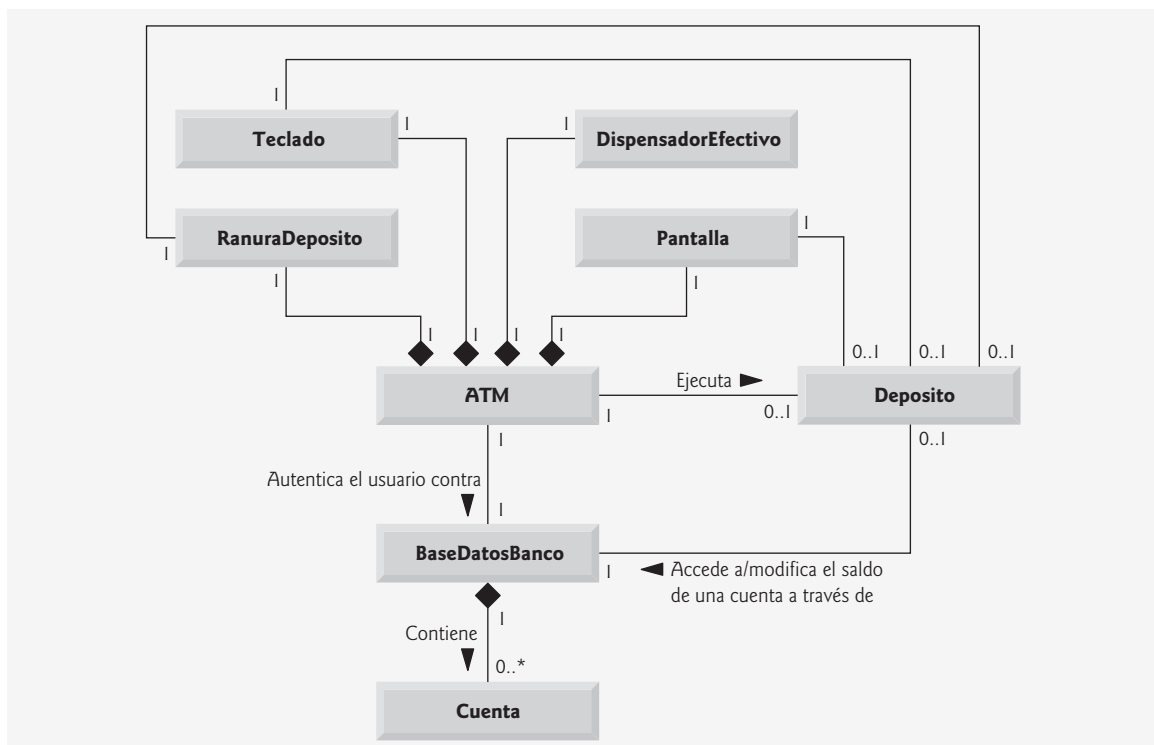


Figura 4.26 | Diagrama de clases para el modelo del sistema ATM, incluyendo la clase **Deposito**.

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

4.1 La figura 4.25 presenta un diagrama de clases que muestra algunas de las relaciones de composición de una clase `Auto`.

4.2 c. En una computadora con conexión de red, esta relación podría ser de varios a varios.

4.3 Verdadera.

4.4 La figura 4.26 presenta un diagrama de clases para el ATM, en el cual se incluye la clase `Deposito` en vez de la clase `Retiro` (como en la figura 4.24). Observe que la clase `Deposito` no se asocia con la clase `DispensadorEfectivo`, sino que se asocia con la clase `RanuraDeposito`.

4.12 Conclusión

En este capítulo aprendió los conceptos básicos orientados a objetos de las clases, los objetos, métodos, las variables de instancia y las propiedades; utilizaremos estos conceptos en aplicaciones de C# más robustas que crearemos. Aprendió a declarar variables de instancia de una clase para mantener los datos de cada objeto de la clase, a declarar métodos que operan sobre esos datos y cómo declarar propiedades para obtener y establecer esos datos. Demostramos cómo llamar a un método para decirle que realice su tarea y cómo pasar información a los métodos en forma de argumentos. Vimos la diferencia entre una variable local de un método y una variable de instancia de una clase, y que sólo las variables de instancia se inicializan en forma automática. También aprendió a utilizar el constructor de una clase para especificar los valores iniciales para las variables de instancia de un objeto. Vimos algunas de las diferencias entre los tipos por valor y los tipos por referencia. Aprendió acerca de los tipos por valor `float`, `double` y `decimal` para almacenar números reales.

A lo largo de este capítulo le mostramos cómo puede utilizarse el UML para crear diagramas de clases que modelen los constructores, métodos, propiedades y atributos de las clases. Aprendió el valor de declarar las variables de instancia como `private`, y a utilizar las propiedades `public` para manipularlas. Por ejemplo, demostramos cómo los descriptores de acceso `set` en las propiedades pueden utilizarse para validar los datos de un objeto y asegurar que ese objeto se mantenga en un estado consistente. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de una aplicación. Utilizará estas instrucciones en sus métodos para especificar cómo deben realizar sus tareas.

5

Instrucciones de control: parte I

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Utilizar las instrucciones de selección `if` e `if...else` para elegir una de varias acciones alternativas.
- Utilizar la instrucción de repetición `while` para ejecutar instrucciones en una aplicación repetidas veces.
- Utilizar la repetición controlada por un contador y la repetición controlada por un centinela.
- Utilizar los operadores de incremento, decremento y de asignación compuestos.

Avancemos todos un lugar.

—Lewis Carroll

La rueda dio una vuelta completa.

—William Shakespeare

¡Cuántas manzanas cayeron sobre la cabeza de Newton antes de que captara el mensaje!

—Robert Frost

Toda la evolución que conocemos avanza de lo impreciso a lo definido.

—Charles Sanders Peirce

- 5.1 Introducción
- 5.2 Estructuras de control
- 5.3 Instrucción de selección simple `if`
- 5.4 Instrucción de selección doble `if...else`
- 5.5 Instrucción de repetición `while`
- 5.6 Cómo formular algoritmos: repetición controlada por un contador
- 5.7 Cómo formular algoritmos: repetición controlada por un centinela
- 5.8 Cómo formular algoritmos: instrucciones de control anidadas
- 5.9 Operadores de asignación compuestos
- 5.10 Operadores de incremento y decremento
- 5.11 Tipos simples
- 5.12 (Opcional) Caso de estudio de ingeniería de software: identificación de los atributos de las clases en el sistema ATM
- 5.13 Conclusión

5.1 Introducción

En este capítulo introduciremos las instrucciones de control `if`, `if...else` y `while` de C#. Dedicaremos una parte del capítulo (y los capítulos 6 y 8) para desarrollar más la clase `LibroCalificaciones` que presentamos en el capítulo 4. En especial, agregaremos un método a la clase `LibroCalificaciones` que utiliza instrucciones de control para calcular el promedio de un conjunto de calificaciones de estudiantes. Otro ejemplo demostrará formas adicionales para combinar instrucciones de control para resolver un problema similar. Presentaremos los operadores de asignación compuestos de C# y exploraremos sus operadores de incremento y decremento. Por último, presentaremos las generalidades acerca de los tipos simples de C#.

5.2 Estructuras de control

Por lo general, las instrucciones en una aplicación se ejecutan una después de la otra, en el orden en que se escribieron. A este proceso se le conoce como *ejecución secuencial*. Varias instrucciones de C# (que veremos pronto) le permiten especificar que la siguiente instrucción a ejecutar no es necesariamente la siguiente en la secuencia. A esto se le conoce como *transferencia de control*.

Durante la década de los sesenta, se hizo evidente que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la *instrucción goto* (utilizada en la mayoría de los lenguajes de programación de ese entonces), la cual permite a los programadores especificar la transferencia de control a uno de los muchos posibles destinos dentro de una aplicación (con lo que se creaba lo que se conoce comúnmente como “código espagueti”). La noción de la llamada *programación estructurada* se hizo casi un sinónimo de la “eliminación del goto”. Le recomendamos evitar la instrucción `goto` de C#.

Las investigaciones de Bohm y Jacopini¹ demostraron que las aplicaciones podrían escribirse sin instrucciones `goto`. El reto de la época para los programadores fue cambiar sus estilos a una “programación sin goto”. No fue sino hasta la década de los setenta cuando los programadores tomaron en serio la programación estructurada. Los resultados fueron impresionantes, a medida que los grupos de desarrollo de software reportaron: reducciones en los tiempos de desarrollo, mayor incidencia de entregas de sistemas a tiempo y más proyectos de software finalizados sin salirse del presupuesto. La clave para estos logros fue que las aplicaciones estructuradas eran más claras, más fáciles de depurar y modificar, y había más probabilidad de que estuvieran libres de errores desde el principio.

El trabajo de Bohm y Jacopini demostró que todas las aplicaciones podían escribirse en términos de tres estructuras de control solamente: la *estructura de secuencia*, la *estructura de selección* y la *estructura de repetición*. El término “estructuras de control” proviene del campo de las ciencias computacionales. Cuando presentemos las

1. Bohm, C. y G. Jacopini, “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules”, *Communications of the ACM*, vol. 9, núm. 5, mayo de 1966, páginas 336-371.

implementaciones de las estructuras de control en C#, nos referiremos a ellas en términos de la *Especificación del lenguaje C#* como “instrucciones de control”.

Estructura de secuencia en C#

La estructura de secuencia está integrada en C#. A menos que se indique lo contrario, la computadora ejecuta las instrucciones en C# una después de otra, en el orden en que estén escritas (esto es, en secuencia). El *diagrama de actividad* de UML en la figura 5.1 ilustra una estructura de secuencia típica, en la que se realizan dos cálculos en orden. C# le permite tener todas las acciones que desee en una estructura de secuencia. Como pronto veremos, en cualquier lugar en donde se pueda colocar una sola acción, se pueden colocar varias acciones en secuencia.

Un diagrama de actividad modela el *flujo de trabajo* (también conocido como la *actividad*) de una parte de un sistema de software. Dichos flujos de trabajo pueden incluir una porción de un algoritmo, como la estructura de secuencia de la figura 5.1. Los diagramas de actividad están compuestos por símbolos de propósito especial, como los *símbolos de estado de acción* (rectángulos en los que sus lados izquierdo y derecho se reemplazan con arcos hacia afuera), *rombos (diamantes)* y *pequeños círculos*. Estos símbolos se conectan mediante *flechas de transición*, que representan el flujo de la actividad; esto es, el orden en que deben ocurrir las acciones.

Los diagramas de actividad le ayudan a desarrollar y representar algoritmos. Muestran con claridad la forma en que deben operar las estructuras de control. Considere el diagrama de actividad para la estructura de secuencia de la figura 5.1, que contiene dos *estados de acción* que representan las acciones a realizar. Cada estado de acción contiene una *expresión de acción* (por ejemplo, “sumar calificación a total” o “sumar 1 a contador”) que especifica una acción particular a realizar. Otras acciones podrían incluir cálculos u operaciones de entrada/salida. Las flechas en el diagrama de actividad representan *transiciones*, que indican el orden en el que ocurren las acciones representadas por los estados de acción. La porción de la aplicación que implementa las actividades ilustradas por el diagrama de la figura 5.1 primero suma calificación a total y después suma 1 a contador.

El *círculo relleno* que se encuentra en la parte superior del diagrama de actividad representa el *estado inicial* de la actividad: el inicio del flujo de trabajo antes de que la aplicación realice las acciones modeladas. El *círculo relleno rodeado por un círculo vacío* que aparece en la parte inferior del diagrama representa el *estado final*: el final del flujo de trabajo después de que la aplicación realiza sus acciones.

La figura 5.1 también incluye rectángulos que tienen la esquina superior derecha doblada. En UML, a estos rectángulos se les llama *notas* (como los comentarios en C#): comentarios con explicaciones que describen el propósito de los símbolos en el diagrama. La figura 5.1 utiliza notas de UML para mostrar el código en C# asociado con cada uno de los estados de acción en el diagrama de actividades. Una *línea punteada* conecta cada nota con el elemento que ésta describe. Los diagramas de actividad por lo general no muestran el código en C# que implementa la actividad. En este libro utilizamos las notas con el propósito de mostrar cómo se relaciona el diagrama con el código en C#. Para obtener más información sobre UML vea nuestro caso de estudio opcional, que aparece en las secciones de Caso de estudio de ingeniería de software al final de los capítulos 1, 3 a 9 y 11; también puede visitar el sitio www.uml.org.

Estructuras de selección en C#

C# cuenta con tres tipos de estructuras de selección, que de aquí en adelante denominaremos *instrucciones de selección*. Hablaremos sobre estas estructuras en este capítulo y en el 6. La *instrucción if* realiza (selecciona) una acción

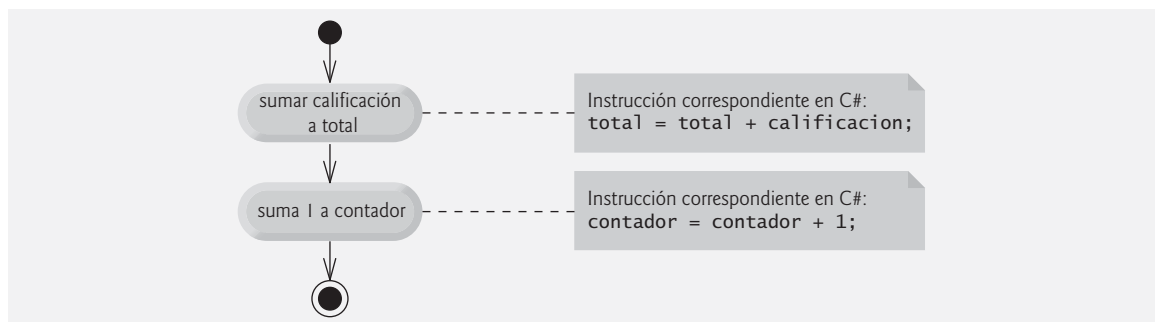


Figura 5.1 | Diagrama de actividad de una estructura de secuencia.

si una condición es verdadera, o ignora la acción si la condición es falsa. La instrucción **if...else** realiza una acción si una condición es verdadera o realiza una acción distinta si la condición es falsa. La instrucción **switch** (capítulo 6) realiza una de varias acciones distintas, dependiendo del valor de una expresión.

A la instrucción **if** se le llama *instrucción de selección simple* debido a que selecciona o ignora una acción individual (o, como pronto veremos, un grupo individual de acciones). A la instrucción **if...else** se le llama *instrucción de selección doble* debido a que selecciona una de dos acciones distintas (o grupos de acciones). A la instrucción **switch** se le llama *instrucción de selección múltiple* debido a que selecciona una de varias acciones distintas (o grupos de acciones).

Estructuras de repetición en C#

C# cuenta con cuatro estructuras de repetición, que de aquí en adelante denominaremos *instrucciones de repetición* (también se les conoce como *instrucciones de iteración* o *ciclos*). Las instrucciones de repetición permiten a las aplicaciones ejecutar instrucciones repetidas veces, dependiendo del valor de una *condición de continuación de ciclo*. Las instrucciones de repetición son: **while**, **do...while**, **for** y **foreach** (en el capítulo 6 presentamos las instrucciones **do...while** y **for**; y en el 8 hablaremos sobre la instrucción **foreach**). Las instrucciones **while**, **for** y **foreach** realizan la acción (o grupo de acciones) contenida en su cuerpo cero o más veces; si la condición de continuación de ciclo es falsa al principio, no se ejecutará la acción (o grupo de acciones). La instrucción **do...while** realiza la acción (o grupo de acciones) contenida en su cuerpo una o más veces.

Las palabras **if**, **else**, **switch**, **while**, **do**, **for** y **foreach** son palabras clave de C# y no pueden utilizarse como identificadores, como en los nombres de las variables. En la figura 3.2 aparece una lista completa de palabras clave de C#.

Resumen de las instrucciones de control en C#

C# sólo cuenta con tres tipos de instrucciones de control estructuradas: la instrucción de secuencia, la de selección (tres tipos) y la de repetición (cuatro tipos). Toda aplicación se forma mediante la combinación de todas las instrucciones de secuencia, de selección y de repetición que sean apropiadas para el algoritmo que implemente la aplicación. Al igual que con la instrucción de secuencia de la figura 5.1, podemos modelar cada instrucción de control como un diagrama de actividad. Cada diagrama contiene un estado inicial y uno final, los cuales representan el punto de entrada y el punto de salida de una instrucción de control, en forma respectiva. Las *instrucciones de control de una entrada/una salida* facilitan la creación de aplicaciones; las instrucciones de control se “conectan” una a otra mediante la conexión del punto de salida de una con el punto de entrada de la siguiente. Este procedimiento es similar a la forma en la que un niño apila bloques de construcción, por lo cual le llamamos *apilamiento de estructuras de control*. En breve aprenderemos que sólo hay una manera alternativa en la que pueden conectarse las instrucciones de control: *anidamiento de instrucciones de control*, en el cual una estructura de control aparece dentro de otra. Por lo tanto, los algoritmos en las aplicaciones en C# se crean a partir de sólo tres tipos de instrucciones de control estructuradas, que se combinan en sólo dos formas. Ésta es la esencia de la simpleza.

5.3 Instrucción de selección simple **if**

Las aplicaciones utilizan instrucciones de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es de 60. La instrucción

```
if ( calif >= 60 )
    Console.WriteLine( "Aprobado" );
```

determina si la condición **calif >= 60** es verdadera o falsa. Si la condición es verdadera se imprime "Aprobado" y se ejecuta la siguiente instrucción en la secuencia. Si la condición es falsa, no se imprime ningún mensaje y se ejecuta la siguiente instrucción en la secuencia.

La figura 5.2 ilustra la instrucción **if** de selección simple. Este diagrama de actividad contiene lo que tal vez sea el símbolo más importante en un diagrama de actividad: el rombo, o *símbolo de decisión*, el cual indica que se tomará una decisión. El flujo de trabajo continuará a lo largo de una ruta determinada por las *condiciones de guardia* asociadas del símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que emerge de un símbolo de decisión tiene una condición de guardia (especificada entre corchetes, enseguida de la flecha de transición). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de

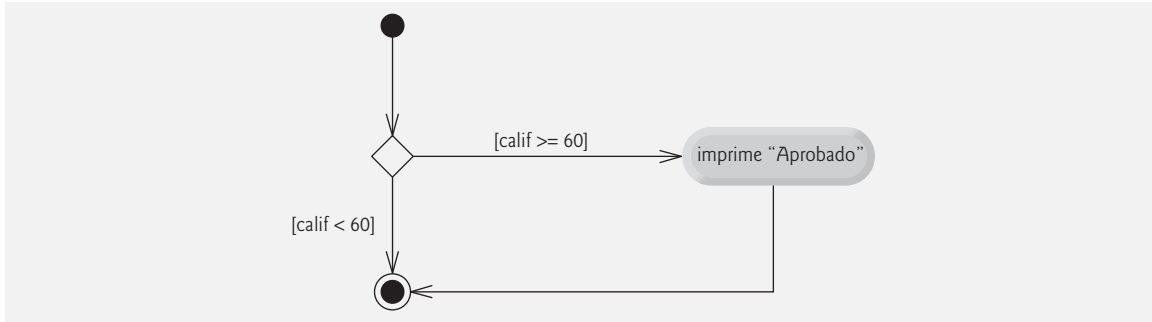


Figura 5.2 | Diagrama de actividad de UML para la instrucción de selección simple `if`.

transición. En la figura 5.2, si la calificación es mayor o igual a 60, la aplicación imprime “Aprobado” y después pasa al estado final de esta actividad. Si la calificación es menor a 60, la aplicación pasa de inmediato al estado final, sin mostrar un mensaje.

La instrucción `if` es una instrucción de control de una entrada/una salida. En breve veremos que los diagramas de actividad para el resto de las instrucciones de control también contienen estados iniciales, flechas de transición, estados de acción que indican las acciones a realizar y símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar, y los estados finales.

5.4 Instrucción de selección doble `if...else`

La instrucción de selección simple `if` realiza una acción indicada sólo cuando la condición es verdadera; en caso contrario se omite la acción. La instrucción de selección doble `if...else` nos permite especificar una acción a realizar cuando la condición es verdadera y una acción distinta cuando la condición es falsa. Por ejemplo, la instrucción

```

if ( calif >= 60 )
    Console.WriteLine( "Aprobado" );
else
    Console.WriteLine( "Reprobado" );
  
```

imprime “Aprobado” si la calificación es mayor o igual a 60, pero imprime “Reprobado” si es menor a 60. En cualquier caso, después de realizar la impresión se ejecuta la siguiente instrucción en la secuencia.

La figura 5.3 ilustra el flujo de control en la instrucción `if...else`. Una vez más, los símbolos en el diagrama de actividad de UML (además del estado inicial, las flechas de transición y el estado final) representan los estados de acción y una decisión.

Operador condicional (`?:`)

C# cuenta con el *operador condicional* (`?:`), que puede utilizarse en lugar de una instrucción `if...else`. Éste es el único *operador ternario* en C#; es decir, que utiliza tres operandos. En conjunto, los operandos y los símbolos `?:` forman una *expresión condicional*. El primer operando (a la izquierda del `?`) es una expresión *booleana*; es decir,

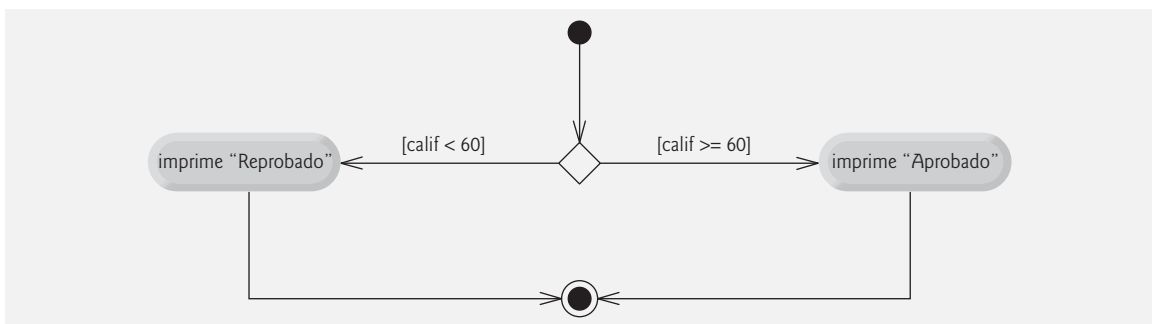


Figura 5.3 | Diagrama de actividad de UML para la instrucción de selección doble `if...else`.

una expresión que se evalúa como un valor tipo `bool`: **true** (verdadero) o **false** (falso). El segundo operando (entre el `?` y el `:`) es el valor de la expresión condicional si la expresión booleana es `true` y el tercer operando (a la derecha del `:`) es el valor de la expresión condicional si la expresión booleana es `false`. Por ejemplo, la instrucción

```
Console.WriteLine( calif >= 60 ? "Aprobado" : "Reprobado" );
```

imprime el valor del argumento de la expresión condicional de `WriteLine`. La expresión condicional en esta instrucción se evalúa como la cadena "Aprobado" si la expresión booleana `calif >= 60` es `true` (verdadera) y se evalúa como la cadena "Reprobado" si la expresión condicional es `false` (falsa). Por lo tanto, esta instrucción con el operador condicional realiza en esencia la misma función que la instrucción `if...else` que mostramos antes en esta sección. Más adelante veremos que las expresiones condicionales pueden utilizarse en algunas situaciones en las que no se pueden utilizar las instrucciones `if...else`.



Buena práctica de programación 5.1

Las expresiones condicionales son más difíciles de leer que las instrucciones `if...else`, por lo que deben utilizarse sólo para sustituir instrucciones `if...else` simples que elijan uno de dos valores.



Buena práctica de programación 5.2

Cuando una expresión condicional se encuentra dentro de una expresión más grande, es una buena práctica colocarla entre paréntesis para mejorar la legibilidad. Al agregar los paréntesis también se pueden evitar los problemas de precedencia de operadores que podrían provocar errores de sintaxis.

Instrucciones `if...else` anidadas

Una aplicación puede evaluar varios casos colocando instrucciones `if...else` dentro de otras instrucciones `if...else`, para crear las **instrucciones `if...else` anidadas**. Por ejemplo, la siguiente instrucción `if...else` anidada imprime A para las calificaciones de los exámenes que sean mayores o iguales a 90, B para el rango de 80 a 89, C para el rango de 70 a 79, D para el rango de 60 a 69 y F para todas las demás:

```
if ( calif >= 90 )
    Console.WriteLine( "A" );
else
    if ( calif >= 80 )
        Console.WriteLine( "B" );
    else
        if ( calif >= 70 )
            Console.WriteLine( "C" );
        else
            if ( calif >= 60 )
                Console.WriteLine( "D" );
            else
                Console.WriteLine( "F" );
```

Si `calif` es mayor o igual a 90, las primeras cuatro condiciones serán verdaderas pero sólo se ejecutará la instrucción en la parte `if` de la primera instrucción `if...else`. Una vez que se ejecuta esa instrucción, se omite la parte `else` de la instrucción `if...else` más "externa". La mayoría de los programadores de C# prefiere escribir la anterior instrucción `if...else` de la siguiente manera:

```
if ( calif >= 90 )
    Console.WriteLine( "A" );
else if ( calif >= 80 )
    Console.WriteLine( "B" );
else if ( calif >= 70 )
    Console.WriteLine( "C" );
else if ( calif >= 60 )
    Console.WriteLine( "D" );
else
    Console.WriteLine( "F" );
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular, ya que evita usar mucha sangría hacia la derecha del código; dicha sangría a menudo deja poco espacio en una línea de código, forzando a que las líneas se dividan y empeorando la legibilidad.

Problema del `else` suelto

El compilador de C# siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves (`{ }`). Este comportamiento puede ocasionar lo que se conoce como el *problema del `else` suelto*. Por ejemplo,

```
if ( x > 5 )
    if ( y > 5 )
        Console.WriteLine( "x e y son > 5" );
else
    Console.WriteLine( "x es <= 5" );
```

parece indicar que si `x` es mayor que 5, la instrucción `if` anidada determina si `y` es también mayor que 5. De ser así, se produce como resultado la cadena "x e y son > 5". De lo contrario, parece ser que si `x` no es mayor que 5, la instrucción `else` que es parte del `if...else` produce como resultado la cadena "x es <= 5".

¡Cuidado! Esta instrucción `if...else` anidada no se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así:

```
if ( x > 5 )
    if ( y > 5 )
        Console.WriteLine( "x e y son > 5" );
else
    Console.WriteLine( "x es <= 5" );
```

en donde el cuerpo del primer `if` es un `if...else` anidado. La instrucción `if` más externa evalúa si `x` es mayor que 5. De ser así, la ejecución continúa evaluando si `y` es también mayor que 5. Si la segunda condición es verdadera, se muestra la cadena apropiada ("x e y son > 5"). No obstante, si la segunda condición es falsa se muestra la cadena "x es <= 5", aun y cuando sabemos que `x` es mayor que 5.

Para forzar a que la instrucción `if...else` anidada se ejecute como se tenía pensado originalmente, debemos escribirla de la siguiente manera:

```
if ( x > 5 )
{
    if ( y > 5 )
        Console.WriteLine( "x e y son > 5" );
}
else
    Console.WriteLine( "x es <= 5" );
```

Las llaves (`{ }`) indican al compilador que la segunda instrucción `if` se encuentra en el cuerpo del primer `if` y que el `else` está asociado con el *primer* `if`.

Bloques

Por lo general, la instrucción `if` espera sólo una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo del `else` en una instrucción `if...else`), encierre las instrucciones entre llaves (`{ }`). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama *bloque*. Un bloque puede colocarse en cualquier parte de un programa en donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte `else` de una instrucción `if...else`:

```
if ( calif >= 60 )
    Console.WriteLine ( "Aprobado" );
else
{
    Console.WriteLine( "Reprobado" );
    Console.WriteLine( "Debe tomar este curso otra vez." );
}
```

En este caso, si `calif` es menor a 60, la aplicación ejecuta ambas instrucciones en el cuerpo del `else` e imprime

```
Reprobado.  
Debe tomar este curso otra vez.
```

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Estas llaves son importantes. Sin ellas, la instrucción

```
Console.WriteLine( "Debe tomar este curso otra vez." );
```

estaría fuera del cuerpo de la parte `else` de la instrucción `if...else` y se ejecutaría sin importar que la calificación fuera menor que 60.



Buena práctica de programación 5.3

Colocar siempre las llaves en una instrucción `if...else` (o cualquier estructura de control) ayuda a evitar que se omitan de manera accidental, en especial cuando se agregan posteriormente instrucciones a la parte `if` o `else`. Para evitar que esto suceda, algunos programadores prefieren escribir la llave inicial y la final de los bloques antes de escribir las instrucciones individuales dentro de ellas.

Así como un bloque puede colocarse en cualquier parte en donde pueda colocarse una sola instrucción, también es posible tener una instrucción vacía. En la sección 3.9 vimos que la instrucción vacía se representa colocando un punto y coma (;) en donde normalmente iría una instrucción.



Error común de programación 5.1

Colocar un punto y coma después de la condición en una instrucción `if` o `if...else` produce un error lógico en las instrucciones `if` de selección simple y un error de sintaxis en las instrucciones `if...else` de selección doble (cuando la parte del `if` contiene una instrucción en el cuerpo).

5.5 Instrucción de repetición `while`

Una *instrucción de repetición* le permite especificar que una aplicación debe repetir una acción mientras cierta condición sea verdadera. Como un ejemplo de la *instrucción de repetición `while`* de C#, considere un segmento de código diseñado para encontrar la primera potencia de 3 que sea mayor que 100. Suponga que la variable `int` llamada `producto` se inicializa con 3. Cuando termina de ejecutarse la siguiente instrucción `while`, `producto` contiene el resultado:

```
int producto = 3;  
  
while ( producto <= 100 )  
    producto = 3 * producto;
```

Cuando esta instrucción `while` comienza a ejecutarse, el valor de la variable `producto` es 3. Cada repetición de la instrucción `while` multiplica `producto` por 3, por lo que `producto` toma los subsiguientes valores 9, 27, 81 y 243 en forma sucesiva. Cuando la variable `producto` se convierte en 243, la condición de la instrucción `while` (`producto <= 100`) se vuelve falsa. Esto termina la repetición, por lo que el valor final de `producto` es 243. En este punto, la ejecución de la aplicación continúa con la siguiente instrucción después de la instrucción `while`.



Error común de programación 5.2

*Si no se proporciona, en el cuerpo de una instrucción `while`, una acción que ocasione que en algún momento la condición del `while` se torne falsa, por lo general se producirá un error lógico conocido como **ciclo infinito**, en el que el ciclo nunca terminará.*

El diagrama de actividad de UML de la figura 5.4 muestra el flujo de control que corresponde a la instrucción `while` anterior. Este diagrama también introduce el *símbolo de fusión*. UML representa tanto al símbolo de fusión como al de decisión como rombos. El símbolo de fusión une dos flujos de actividad en uno solo. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que ambas

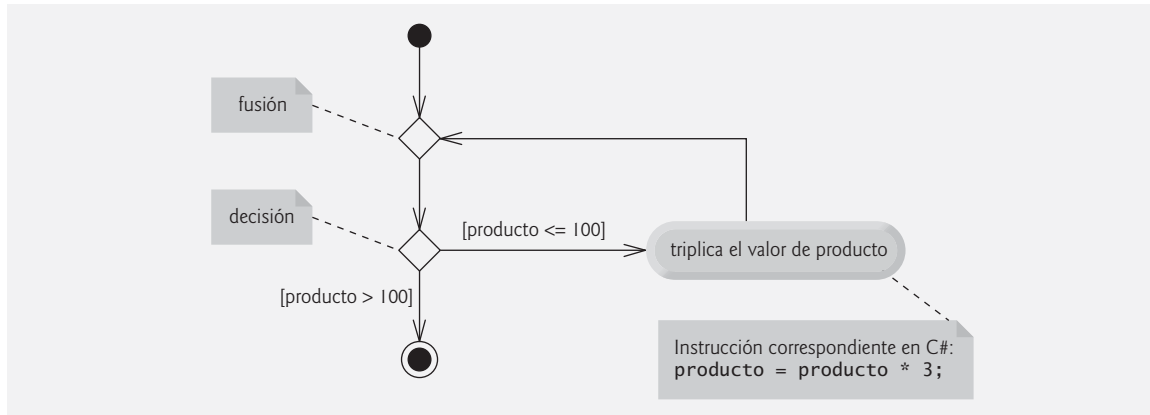


Figura 5.4 | Diagrama de actividad de UML para la instrucción de repetición `while`.

fluyan en la decisión que determina si el ciclo debe empezar a ejecutarse (o seguir ejecutándose). Los símbolos de decisión y de fusión pueden diferenciarse por el número de flechas de transición “entrantes” y “salientes”. Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo y dos o más que apuntan hacia fuera del rombo, para indicar las posibles transiciones desde ese punto. Cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo y sólo una flecha de transición que apunta hacia fuera del rombo, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. Ninguna de las flechas de transición asociadas con un símbolo de fusión tiene condiciones de guardia.

La figura 5.4 muestra con claridad la repetición de la instrucción `while` que vimos antes en esta sección. La flecha de transición que emerge del estado de acción apunta de vuelta a la fusión, desde la cual el flujo del programa se dirige nuevamente hacia la decisión que se evalúa al principio de cada repetición del ciclo. El ciclo continúa ejecutándose hasta que la condición de guardia `producto > 100` se torna verdadera. Entonces, termina la instrucción `while` (llega a su estado final) y el control pasa a la siguiente instrucción en la secuencia de la aplicación.

5.6 Cómo formular algoritmos: repetición controlada por un contador

Para ilustrar cómo se desarrollan los algoritmos, modificamos la clase `LibroCalificaciones` del capítulo 4 para resolver dos variaciones de un problema que promedia las calificaciones de los estudiantes. Considere el siguiente enunciado del problema:

A una clase de 10 estudiantes se les aplicó un examen. Las calificaciones (enteros en el rango de 0 a 100) de este examen están disponibles para su análisis. Determine el promedio de la clase en el examen.

El promedio de la clase es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada calificación, llevar el registro del total de todas las calificaciones introducidas, realizar el cálculo del promedio e imprimir el resultado.

Emplearemos la **repetición controlada por un contador** para introducir las calificaciones, una a la vez. Esta técnica utiliza una variable llamada *contador* (o *variable de control*) para controlar el número de veces que se ejecutará un conjunto de instrucciones. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta una versión de la clase `LibroCalificaciones` (figura 5.5) que implementa el algoritmo en un método de C#. Después se presenta una aplicación (figura 5.6) que demuestra el algoritmo en acción.

Implementación de la repetición controlada por un contador en la clase `LibroCalificaciones`

La clase `LibroCalificaciones` (figura 5.5) contiene un constructor (líneas 11-14) que asigna un valor a la variable de instancia `nombreCurso` (declarada en la línea 8) de la clase, mediante el uso de la propiedad `NombreCurso`. Las líneas 17-27 y 30-35 declaran la propiedad `NombreCurso` y el método `MostrarMensaje`, en forma respectiva.

```

1  // Fig. 5.5: LibroCalificaciones.cs
2  // Clase LibroCalificaciones que resuelve el problema del promedio de la clase
3  // utilizando la repetición controlada por un contador.
4  using System;
5
6  public class LibroCalificaciones
7  {
8      private string nombreCurso; // nombre del curso que representa este LibroCalificaciones
9
10     // el constructor inicializa nombreCurso
11     public LibroCalificaciones( string nombre )
12     {
13         NombreCurso = nombre; // inicializa nombreCurso usando la propiedad
14     } // fin del constructor
15
16     // propiedad para obtener (get) y establecer (set) el nombre del curso
17     public string NombreCurso
18     {
19         get
20         {
21             return nombreCurso;
22         } // fin de get
23         set
24         {
25             nombreCurso = value; // set debería validar
26         } // fin de set
27     } // fin de la propiedad NombreCurso
28
29     // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
30     public void MostrarMensaje()
31     {
32         // la propiedad NombreCurso obtiene el nombre del curso
33         Console.WriteLine( "Bienvenido al libro de calificaciones de\n{0}!\n",
34                             NombreCurso );
35     } // fin del método MostrarMensaje
36
37     // determina el promedio de la clase con base en las 10 calificaciones introducidas
38     // por el usuario
39     public void DeterminarPromedioClase()
40     {
41         int total; // suma de las calificaciones introducidas por el usuario
42         int contadorCalif; // número de la siguiente calificación a introducir
43         int calificacion; // valor de la calificación introducida por el usuario
44         int promedio; // promedio de las calificaciones
45
46         // fase de inicialización
47         total = 0; // inicializa el total
48         contadorCalif = 1; // inicializa el contador del ciclo
49
50         // fase de procesamiento
51         while ( contadorCalif <= 10 ) // itera 10 veces
52         {
53             Console.Write( "Escriba calificación: " ); // mensaje para el usuario
54             calificacion = Convert.ToInt32( Console.ReadLine() ); // lee calificación
55             total = total + calificacion; // suma la calificación al total
56             contadorCalif = contadorCalif + 1; // incrementa el contador en 1
57         } // fin de while
58     }
59 }

```

Figura 5.5 | La clase LibroCalificaciones que resuelve el problema del promedio de la clase, mediante el uso de la repetición controlada por un contador. (Parte I de 2).


```

58      // fase de terminación
59      promedio = total / 10; // división entera produce resultado entero
60
61      // muestra el total y el promedio de las calificaciones
62      Console.WriteLine( "\nEl total de las 10 calificaciones es {0}", total );
63      Console.WriteLine( "El promedio de la clase es {0}", promedio );
64  } // fin del método DeterminarPromedioClase
65 } // fin de la clase LibroCalificaciones

```

Figura 5.5 | La clase `LibroCalificaciones` que resuelve el problema del promedio de la clase, mediante el uso de la repetición controlada por un contador. (Parte 2 de 2).

Las líneas 38-64 declaran el método `DeterminarPromedioClase`, el cual implementa el algoritmo para calcular el promedio de la clase.

Las líneas 40-43 declaran las variables locales `total`, `contadorCalif`, `calificacion` y `promedio` de tipo `int`. La variable `calificacion` almacena los valores introducidos por el usuario.

Observe que las declaraciones (en las líneas 40-43) aparecen en el cuerpo del método `DeterminarPromedioClase`. Recuerde que las variables que se declaran en el cuerpo de un método son variables locales, y sólo pueden utilizarse desde la línea de su declaración en el método hasta la llave derecha de cierre de la declaración del método. La declaración de una variable local debe aparecer antes de utilizarla en ese método. Una variable local no puede utilizarse fuera del método en el que está declarada.

En las versiones de la clase `LibroCalificaciones` en este capítulo, sólo leemos y procesamos un conjunto de calificaciones. El cálculo del promedio se realiza en el método `DeterminarPromedioClase` mediante el uso de variables locales; no preservamos ningún tipo de información sobre las calificaciones de los estudiantes en variables de instancia de la clase. En versiones posteriores de esta clase (en el capítulo 8, Arreglos) almacenaremos las calificaciones en memoria mediante el uso de una variable de instancia que haga referencia a una estructura de datos conocida como arreglo. Esto permite a un objeto `LibroCalificaciones` realizar varios cálculos sobre el mismo conjunto de calificaciones, sin necesidad de que el usuario tenga que repetir varias veces el proceso de introducir las calificaciones.

Decimos que una variable se *asigna definitivamente* cuando la variable se asigna en todos los posibles flujos de control. Observe que cada variable local declarada en las líneas 40-43 se asigna definitivamente antes de utilizarse en los cálculos. Las asignaciones (en las líneas 46-47) inicializan `total` a 0 y `contadorCalif` a 1. Las variables `calificacion` y `promedio` (para la entrada del usuario y el promedio calculado, respectivamente) no necesitan inicializarse aquí; sus valores se asignan a medida que se introducen como entrada o se calculan más adelante en el método.



Error común de programación 5.3

Si se utiliza el valor de una variable local antes de que se asigne definitivamente, se produce un error de compilación. Todas las variables locales deben asignarse definitivamente antes de utilizar sus valores en una expresión.



Tip de prevención de errores 5.1

Inicialice cada contador y total, ya sea en su declaración o en una instrucción de asignación. Por lo general, los totales se inicializan a 0. Los contadores se inicializan comúnmente a 0 o 1, dependiendo de la forma en que se utilicen (en breve mostraremos ejemplos de cada uno).

La línea 50 indica que la instrucción `while` debe continuar iterando mientras que el valor de `contadorCalif` sea menor o igual a 10. Mientras esta condición siga siendo verdadera, la instrucción `while` ejecutará repetidas veces las instrucciones entre las llaves que delimitan su cuerpo (líneas 51-56).

La línea 52 muestra el mensaje "Escriba calificación: " en la ventana de consola. La línea 53 lee la calificación introducida por el usuario y la asigna a la variable `calificacion`. Después la línea 54 suma la nueva `calificacion` introducida por el usuario al `total` y asigna el resultado a `total`, con lo cual se sustituye su valor anterior.

La línea 55 suma 1 a `contadorCalif` para indicar que la aplicación ha procesado una calificación y está lista para recibir como entrada la siguiente calificación de parte del usuario. El proceso de incrementar `contadorCalif`

en un momento dado hará que `contadorCalif` exceda a 10. En ese punto terminará el ciclo `while`, ya que su condición (línea 50) se volverá falsa.

Cuando termina el ciclo, la línea 59 realiza el cálculo del promedio y asigna su resultado a la variable `promedio`. La línea 62 utiliza el método `WriteLine` de `Console` para mostrar el texto "El total de las 10 calificaciones es ", seguido del valor de la variable `total`. Después la línea 63 utiliza `WriteLine` para mostrar el texto "El promedio de la clase es ", seguido del valor de la variable `promedio`. El método `DeterminarPromedioClase` devuelve el control al método que lo llamó (es decir, `Main` en `PruebaLibroCalificaciones` de la figura 5.6) después de llegar a la línea 64.

La clase `PruebaLibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 5.6) crea un objeto de la clase `LibroCalificaciones` (figura 5.5) y demuestra sus capacidades. Las líneas 9-10 de la figura 5.6 crean un nuevo objeto `LibroCalificaciones` y lo asignan a la variable `miLibroCalificaciones`. El objeto `string` de la línea 10 se pasa al constructor de `LibroCalificaciones` (líneas 11-14 de la figura 5.5). La línea 12 llama al método `MostrarMensaje` de `miLibroCalificaciones` para mostrar un mensaje de bienvenida al usuario. Después la línea 13 llama al método `DeterminarPromedioClase` de `miLibroCalificaciones` para permitir que el usuario introduzca 10 calificaciones, después de lo cual el método calcula e imprime el promedio; el método ejecuta el algoritmo que se muestra en la figura 5.5.

Notas acerca de la división de enteros y el truncamiento

El cálculo del promedio que realiza el método `DeterminarPromedioClase` en respuesta a la llamada del método de la línea 13 de la figura 5.6 produce un resultado entero. La salida de la aplicación indica que la suma de los

```

1  // Fig. 5.6: PruebaLibroCalificaciones.cs
2  // Crea el objeto LibroCalificaciones e invoca a su método DeterminarPromedio.
3  public class PruebaLibroCalificaciones
4  {
5      public static void Main( string[] args )
6      {
7          // crea el objeto miLibroCalificaciones de LibroCalificaciones y
8          // pasa el nombre del curso al constructor
9          LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
10             "CS101 Introducción a la programación en C#" );
11
12         miLibroCalificaciones.MostrarMensaje(); // muestra el mensaje de bienvenida
13         miLibroCalificaciones.DeterminarPromedioClase(); // encuentra el promedio de 10
14             calificaciones
15     } // fin de Main
16 } // fin de la clase PruebaLibroCalificaciones

```

Bienvenido al libro de calificaciones de
CS101 Introducción a la programación en C#!

Escriba calificación: 88
Escriba calificación: 79
Escriba calificación: 95
Escriba calificación: 100
Escriba calificación: 48
Escriba calificación: 88
Escriba calificación: 92
Escriba calificación: 83
Escriba calificación: 90
Escriba calificación: 85

El total de las 10 calificaciones es 848
El promedio de la clase es 84

Figura 5.6 | Crear el objeto `LibroCalificaciones` e invocar a su método `DeterminarPromedioClase`.

valores de las calificaciones en la ejecución de ejemplo es 848, que al dividirse entre 10 debería producir el número de punto flotante 84.8. No obstante, el resultado del cálculo `total / 10` (línea 59 de la figura 5.5) es el entero 84, ya que `total` y 10 son ambos enteros. La división de dos enteros produce una *división entera*: se pierde cualquier fracción del cálculo (es decir, se *trunca*, no se redondea). En la siguiente sección veremos cómo obtener un resultado de punto flotante a partir del cálculo de un promedio.



Error común de programación 5.4

Suponer que la división entera se redondea (en vez de truncarse) puede producir resultados incorrectos. Por ejemplo, $7 \div 4$, cuyo resultado es 1.75 en la aritmética convencional, se trunca a 1 en la aritmética de enteros, en vez de redondearse a 2.

5.7 Cómo formular algoritmos: repetición controlada por un centinela

Generalicemos el problema del promedio de la clase de la sección 5.6. Considere el siguiente problema:

Desarrolle una aplicación para calcular el promedio de una clase, que procese las calificaciones para un número arbitrario de estudiantes cada vez que se ejecute.

En el ejemplo anterior del promedio de la clase, el enunciado del problema especificaba el número de estudiantes, por lo que se conocía de antemano el número de calificaciones (10). En este ejemplo no se da indicación alguna de cuántas calificaciones introducirá el usuario durante la ejecución de la aplicación. Esta aplicación debe procesar un número arbitrario de calificaciones.

Una manera de resolver este problema es mediante el uso de un valor especial llamado *valor centinela* (también conocido como *valor de señal*, *valor de señuelo* o *valor de bandera*) para indicar el “fin de la entrada de datos”. A esto se le conoce como *repetición controlada por un centinela*. El usuario introduce calificaciones hasta que se hayan introducido todas las calificaciones legítimas. Después escribe el valor centinela para indicar que ya no va a introducir más calificaciones.

Es evidente que debe elegirse un valor centinela que no pueda confundirse con un valor de entrada aceptable. Las calificaciones en un examen son enteros no negativos, por lo que `-1` es un valor centinela aceptable para este problema. Por ende, la ejecución de la aplicación del promedio de la clase podría procesar un flujo de entradas como 95, 96, 75, 74, 89 y `-1`. Después la aplicación calcularía e imprimiría el promedio de la clase para las calificaciones 95, 96, 75, 74 y 89. Como `-1` es el valor centinela, no se utiliza en el cálculo del promedio.

Cómo implementar la repetición controlada por un centinela en la clase `LibroCalificaciones`

La figura 5.7 muestra la clase `LibroCalificaciones` en C#, la cual contiene el método `DeterminarPromedioClase` que utiliza la repetición controlada por un centinela. Aunque cada calificación es de tipo entero, existe la posibilidad de que el cálculo del promedio produzca como resultado un número con un punto decimal; en otras palabras, un número real o de punto flotante. El tipo `int` no puede representar dicho número, por lo que esta clase utiliza el tipo `double` para hacerlo.

La instrucción `while` (líneas 54-62) va seguida en secuencia por una instrucción `if...else` (líneas 66-77). La mayor parte del código en esta aplicación es idéntico al de la figura 5.5, por lo que nos concentraremos en las nuevas características y cuestiones.

La línea 42 declara la variable `double` llamada `promedio`. Esta variable nos permite almacenar el promedio calculado de la clase como un número de punto flotante. La línea 46 inicializa `contadorCalif` a 0, ya que todavía no se han introducido calificaciones. Recuerde que esta aplicación utiliza la repetición controlada por un centinela para recibir como entrada las calificaciones de parte del usuario. Para llevar un registro preciso del número de calificaciones introducidas, la aplicación incrementa `contadorCalif` sólo cuando el usuario introduce una calificación válida.

Comparación entre la lógica del programa para la repetición controlada por un centinela y la repetición controlada por un contador

Compare la lógica del programa para la repetición controlada por un centinela en esta aplicación, con la de la repetición controlada por un contador de la figura 5.5. En la repetición controlada por un contador, cada repetición de

```

1 // Fig. 5.7: LibroCalificaciones.cs
2 // Clase LibroCalificaciones que resuelve el problema del promedio de la clase
3 // usando la repetición controlada por un centinela.
4 using System;
5
6 public class LibroCalificaciones
7 {
8     private string nombreCurso; // nombre del curso que representa este LibroCalificaciones
9
10    // el constructor inicializa nombreCurso
11    public LibroCalificaciones( string nombre )
12    {
13        NombreCurso = nombre; // inicializa nombreCurso utilizando la propiedad
14    } // fin del constructor
15
16    // propiedad para obtener (get) y establecer (set) el nombre del curso
17    public string NombreCurso
18    {
19        get
20        {
21            return nombreCurso;
22        } // fin de get
23        set
24        {
25            nombreCurso = value; // set debería validar
26        } // fin de set
27    } // fin de la propiedad NombreCurso
28
29    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
30    public void MostrarMensaje()
31    {
32        Console.WriteLine( "Bienvenido al libro de calificaciones para\n{0}!\n",
33            NombreCurso );
34    } // fin del método MostrarMensaje
35
36    // determina el promedio de un número arbitrario de calificaciones
37    public void DeterminaPromedioClase()
38    {
39        int total; // suma de las calificaciones
40        int contadorCalif; // número de calificaciones introducidas
41        int calificacion; // valor de la calificación
42        double promedio; // número con punto decimal para el promedio
43
44        // fase de inicialización
45        total = 0; // inicializa total
46        contadorCalif = 0; // inicializa contador de ciclo
47
48        // fase de procesamiento
49        // mensaje para recibir como entrada la calificación del usuario
50        Console.Write( "Escriba calificación o -1 para salir: " );
51        calificacion = Convert.ToInt32( Console.ReadLine() );
52
53        // itera hasta que se lee el valor centinela del usuario
54        while ( calificacion != -1 )
55        {
56            total = total + calificacion; // suma calificacion a total
57            contadorCalif = contadorCalif + 1; // incrementa el contador
58

```

Figura 5.7 | La clase LibroCalificaciones, que resuelve el problema del promedio de la clase usando la repetición controlada por un centinela. (Parte I de 2).

```

59      // mensaje para recibir como entrada la siguiente calificación del usuario
60      Console.Write( "Escriba calificación o -1 para salir: " );
61      calificacion = Convert.ToInt32( Console.ReadLine() );
62  } // fin de while
63
64      // fase de terminación
65      // si el usuario introdujo cuando menos una calificación...
66      if ( contadorCalif != 0 )
67      {
68          // calcula el promedio de todas las calificaciones introducidas
69          promedio = ( double ) total / contadorCalif;
70
71          // muestra el total y el promedio (con dos dígitos de precisión)
72          Console.WriteLine( "\nEl total de las {0} calificaciones introducidas es {1}",
73              contadorCalif, total );
74          Console.WriteLine( "El promedio de la clase es {0:F2}", promedio );
75      } // fin del if
76      else // no se introdujeron calificaciones, entonces mostrar mensaje de error
77          Console.WriteLine( "No se introdujeron calificaciones" );
78  } // fin del método DeterminaPromedioClase
79 } // fin de la clase LibroCalificaciones

```

Figura 5.7 | La clase LibroCalificaciones, que resuelve el problema del promedio de la clase usando la repetición controlada por un centinela. (Parte 2 de 2).

la instrucción `while` (por ejemplo, las líneas 50-56 de la figura 5.5) lee un valor del usuario, durante el número especificado de repeticiones. En la repetición controlada por un centinela, la aplicación lee el primer valor (líneas 50-51 de la figura 5.7) antes de llegar al `while`. Este valor determina si el flujo de control de la aplicación debe entrar al cuerpo del `while`. Si la condición del `while` es falsa, significa que el usuario introdujo el valor centinela, por lo que no se ejecuta el cuerpo del `while` (debido a que no se introdujeron calificaciones). Si, por otro lado, la condición es verdadera, el cuerpo comienza a ejecutarse y el ciclo suma el valor de `calificacion` al `total` (línea 56), y suma 1 a `contadorCalif` (línea 57). Después las líneas 60-61 en el cuerpo del ciclo reciben como entrada el siguiente valor de parte del usuario. A continuación, el control del programa llega a la llave derecha de cierre del cuerpo en la línea 62, por lo que la ejecución continúa con la evaluación de la condición del `while` (línea 54). La condición utiliza la entrada más reciente de `calificacion` recibida del usuario para determinar si el cuerpo del ciclo debe ejecutarse otra vez. Observe que el valor de la variable `calificacion` siempre se recibe como entrada del usuario, justo antes de que la aplicación evalúe la condición del `while`. Esto permite a la aplicación determinar si el valor que acaba de recibir como entrada es el valor centinela *antes* de que la aplicación procese ese valor (es decir, que lo sume al `total`). Si se introduce el valor centinela, el ciclo termina; la aplicación *no* suma -1 al `total`.



Buena práctica de programación 5.4

En un ciclo controlado por centinela, los mensajes que solicitan datos deben recordar explícitamente al usuario el valor centinela.

Una vez que termina el ciclo, se ejecuta la instrucción `if...else` de las líneas 66-77. La condición en la línea 66 determina si se introdujeron calificaciones. Si no se introdujo ninguna, se ejecuta la parte `else` (línea 76-77) de la instrucción `if...else` y se muestra en pantalla el mensaje "No se introdujeron calificaciones", y el método devuelve el control al método que hizo la llamada.

Conversiones explícitas e implícitas entre los tipos simples

Si se introdujo cuando menos una calificación, la línea 69 de la figura 5.7 calcula el promedio de las calificaciones. En la figura 5.5 vimos que la división entera produce un resultado entero. Aun y cuando la variable `promedio` se declara como `double` (línea 42), el cálculo

```
promedio = total / contadorCalif;
```

pierde la parte fraccional del cociente antes de asignar el resultado de la división a la variable promedio. Esto ocurre debido a que `total` y `contadorCalif` son variables enteras, y la división entera produce un resultado entero. Para realizar un cálculo de punto flotante con valores enteros, debemos tratar estos valores temporalmente como números de punto flotante para utilizarlos en el cálculo. C# cuenta con el **operador de conversión unario** para realizar esta tarea. La línea 69 utiliza el operador de conversión (**`double`**) (un operador unario) para crear una copia *temporal* de punto flotante de su operando `total` (el cual aparece a la derecha del operador). Al uso de un operador de conversión de esta manera se le llama **conversión explícita**. El valor almacenado en `total` sigue siendo un entero.

El cálculo ahora consiste de un valor de punto flotante (la versión `double` temporal de `total`) dividido entre el entero `contadorCalif`. C# sabe cómo evaluar sólo expresiones aritméticas en las que los tipos de los operandos sean idénticos. Para asegurar que los operandos sean del mismo tipo, C# realiza una operación conocida como **promoción** (o **conversión implícita**) sobre los operandos seleccionados. Por ejemplo, en una expresión que contiene valores de los tipos `int` y `double`, los valores `int` se promueven a valores `double` para usarlos en la expresión. En este ejemplo, el valor de `contadorCalif` se promueve al tipo `double`, después se realiza la división de punto flotante y el resultado del cálculo se asigna a `promedio`. Siempre que el operador de conversión (`double`) se aplique a cualquier variable en el cálculo, éste producirá un resultado `double`. Más adelante en este capítulo veremos todos los tipos simples. En la sección 7.7 aprenderá más acerca de las reglas de promoción.



Error común de programación 5.5

El operador de conversión puede utilizarse para convertir entre los tipos numéricos simples, como `int` y `double`, y entre los tipos por referencia relacionados (como veremos en el capítulo 11, Polimorfismo, interfaces y sobrecarga de operadores). La conversión al tipo incorrecto puede ocasionar errores de compilación o errores en tiempo de ejecución.

Hay operadores de conversión disponibles para todos los tipos simples (en el capítulo 11 hablaremos sobre los operadores de conversión para los tipos por referencia). Para formar el operador de conversión, se colocan paréntesis alrededor del nombre de un tipo. Los operadores de conversión asocian de derecha a izquierda y tienen la misma precedencia que otros operadores unarios, como el `+` y el `-` unarios. Esta precedencia es un nivel mayor que la de los **operadores multiplicativos** `*`, `/` y `%`. (En el apéndice A podrá consultar la tabla de precedencia de operadores.) En nuestras tablas de precedencia indicamos el operador de conversión con la notación (*tipo*), para demostrar que puede utilizarse cualquier nombre de tipo para formar un operador de conversión.

La línea 74 muestra en pantalla el promedio de la clase mediante el uso del método `WriteLine` de `Console`. En este ejemplo decidimos que sería mejor mostrar el promedio de la clase redondeado al centésimo más cercano y mostrar el promedio con exactamente dos dígitos a la derecha del punto decimal. El especificador de formato `F` en el elemento de formato de `WriteLine` (línea 74) indica que el valor de la variable `promedio` debe mostrarse como un número real. El número después del especificador de formato `F` representa el número de lugares decimales (en este caso, 2) que deben mostrarse a la derecha del punto decimal en el número de punto flotante; a esto se le conoce también como la **precisión** del número. Cualquier valor de punto flotante que se imprima en pantalla con `F2` se redondeará a la posición de las centésimas; por ejemplo, 123.457 se redondearía a 123.46 y 27.333 se redondearía a 27.33. En esta aplicación, las tres calificaciones que se introducen durante la ejecución de ejemplo de la clase `PruebaLibroCalificaciones` (figura 5.8) dan un total de 263, lo cual produce el promedio 87.66666... El elemento de formato redondea el promedio a la posición de las centésimas, por lo que el promedio se muestra en pantalla como 87.67.

5.8 Cómo formular algoritmos: instrucciones de control anidadas

En este caso de estudio, vamos a **anidar** una estructura de control dentro de otra. Considere el siguiente enunciado del problema:

Una universidad ofrece un curso que prepara a los estudiantes para el examen de la licencia estatal para los agentes de bienes raíces. El año pasado, 10 de los estudiantes que completaron este curso tomaron el examen. La universidad desea saber qué tan bien se desempeñaron los estudiantes en el examen. A usted se le ha pedido que escriba

```

1 // Fig. 5.8: PruebaLibroCalificaciones.cs
2 // Crea el objeto LibroCalificaciones e invoca a su método DeterminarPromedioClase.
3 public class PruebaLibroCalificaciones
4 {
5     public static void Main( string[] args )
6     {
7         // crea el objeto miLibroCalificaciones de la clase LibroCalificaciones y
8         // pasa el nombre del curso al instructor
9         LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
10             "CS101 Introducción a la programación en C#" );
11
12         miLibroCalificaciones.MostrarMensaje(); // muestra mensaje de bienvenida
13         miLibroCalificaciones.DeterminarPromedioClase(); // encuentra el promedio de las
            calificaciones
14     } // fin de Main
15 } // fin de la clase PruebaLibroCalificaciones

```

Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C#!

Escriba calificación o -1 para salir: 96
Escriba calificación o -1 para salir: 88
Escriba calificación o -1 para salir: 79
Escriba calificación o -1 para salir: -1

El total de las 3 calificaciones introducidas es 263
El promedio de la clase es 87.67

Figura 5.8 | Crear el objeto LibroCalificaciones e invocar a su método DeterminarPromedioClase.

una aplicación para sintetizar los resultados. Para ello se le ha dado una lista de estos 10 estudiantes. En seguida de cada uno de los nombres está escrito un 1 si el estudiante aprobó el examen o un 2 si el estudiante reprobó.

Su aplicación debe analizar los resultados del examen, de la siguiente manera:

1. Recibir como entrada cada resultado de la prueba (es decir, un 1 o un 2). Mostrar el mensaje "Escriba el resultado" en la pantalla cada vez que la aplicación requiera otro resultado de la prueba.
2. Contar el número de resultados de la prueba de cada tipo.
3. Mostrar un resumen de los resultados de la prueba, indicando el número de estudiantes que aprobaron y el número de estudiantes que reprobaron.
4. Si más de ocho estudiantes aprobaron el examen, imprimir el mensaje "Aumentar colegiatura".

Después de leer el enunciado del problema, podemos hacer las siguientes observaciones:

1. La aplicación debe procesar los resultados de la prueba para 10 estudiantes. Puede utilizarse un ciclo controlado por un contador, ya que se conoce de antemano el número de resultados de la prueba.
2. Cada resultado de la prueba tiene un valor numérico: ya sea 1 o 2. Cada vez que la aplicación lee un resultado de la prueba, debe determinar si el número es 1 o 2. En nuestro algoritmo evaluamos para 1. Si el número no es 1, suponemos que es 2.
3. Se utilizan dos contadores para llevar el registro de los resultados del examen: uno para contar el número de estudiantes que aprobaron y otro para contar el número de estudiantes que reprobaron.
4. Una vez que la aplicación ha procesado todos los resultados, debe determinar si más de ocho estudiantes aprobaron el examen.

En la figura 5.9 se muestra la clase en C# que implementa el algoritmo y en la figura 5.10 aparecen dos ejecuciones de ejemplo.

```

1  // Fig. 5.9: Analisis.cs
2  // Análisis de los resultados del examen, usando instrucciones de control anidadas.
3  using System;
4
5  public class Analisis
6  {
7      public void ProcesarResultadosExamen()
8      {
9          // inicializa las variables en las declaraciones
10         int aprobados = 0; // número de aprobados
11         int reprobados = 0; // número de reprobados
12         int contadorEstudiantes = 1; // contador de estudiantes
13         int resultado; // un resultado del examen del usuario
14
15         // procesa 10 estudiantes usando la repetición controlada por un contador
16         while ( contadorEstudiantes <= 10 )
17         {
18             // pide la entrada al usuario y obtiene el valor del usuario
19             Console.Write( "Escriba resultado (1 = aprobado, 2 = reprobado): " );
20             resultado = Convert.ToInt32( Console.ReadLine() );
21
22             // if...else anidada en el while
23             if ( resultado == 1 ) // si el resultado es 1,
24                 aprobados = aprobados + 1; // incrementa aprobados
25             else // en caso contrario el resultado no es 1, por lo que
26                 reprobados = reprobados + 1; // incrementa reprobados
27
28             // incrementa contadorEstudiantes para que el ciclo termine en un momento dado
29             contadorEstudiantes = contadorEstudiantes + 1;
30         } // fin de while
31
32         // fase de terminación; prepara y muestra los resultados
33         Console.WriteLine( "Aprobados: {0}\nReprobados: {1}", aprobados, reprobados );
34
35         // determina si aprobaron más de 8 estudiantes
36         if ( aprobados > 8 )
37             Console.WriteLine( "Aumentar colegiatura" );
38     } // fin del método ProcesarResultadosExamen
39 } // fin de la clase Analisis

```

Figura 5.9 | Análisis de los resultados de un examen, usando instrucciones de control anidadas.

Las líneas 10-13 de la figura 5.9 declaran las variables que utiliza el método `ProcesarResultadosExamen` de la clase `Analisis` para procesar los resultados del examen. Varias de estas declaraciones utilizan la habilidad de C# de incorporar la inicialización de variables en las declaraciones (se asigna un 0 a `aprobados`, se asigna un 0 a `reprobados` y se asigna un 1 a `contadorEstudiantes`). Las aplicaciones con ciclos pueden requerir la inicialización al inicio de cada repetición; por lo general dicha inicialización se lleva a cabo mediante instrucciones de asignación, en vez de hacerlo en las declaraciones.

La instrucción `while` (líneas 16-30) itera 10 veces. Durante cada repetición, el ciclo recibe como entrada y procesa un resultado del examen. Observe que la instrucción `if...else` (líneas 23-26) para procesar cada resultado está anidada en la instrucción `while`. Si `resultado` es 1, la instrucción `if...else` incrementa `aprobados`; en caso contrario, supone que el resultado es 2 e incrementa `reprobados`. La línea 29 incrementa `contadorEstudiantes` antes de evaluar de nuevo la condición del ciclo en la línea 16. Una vez que se introducen 10 valores, el ciclo termina y la línea 33 muestra en pantalla el número de `aprobados` y el número de `reprobados`. La instrucción `if` en las líneas 36-37 determina si más de ocho estudiantes aprobaron el examen y, de ser así, imprime en pantalla el mensaje "Aumentar colegiatura".



Tip de prevención de errores 5.2

Inicializar las variables locales al momento de declararlas le ayuda a evitar errores de compilación que podrían surgir al tratar de utilizar datos sin inicializar. Aunque C# no requiere que las inicializaciones de las variables locales se incorporen en las declaraciones, sí requiere que las variables locales se inicialicen antes de utilizar sus valores en una expresión.

La clase `PruebaAnalisis` que demuestra el uso de la clase `Analisis`

La clase `PruebaAnalisis` (figura 5.10) crea un objeto `Analisis` (línea 7) e invoca al método `ProcesarResultadosExamen` de ese objeto (línea 8) para procesar un conjunto de resultados de un examen introducidos por el usuario. La figura 5.10 muestra la entrada y la salida de dos ejecuciones de ejemplo de la aplicación. Durante la primera ejecución de ejemplo, la condición en la línea 36 del método `ProcesarResultadosExamen` de la figura 5.9 es verdadera; más de ocho estudiantes aprobaron el examen, por lo que la aplicación imprime en pantalla un mensaje indicando que se debe aumentar la colegiatura.

```

1  // Fig. 5.10: PruebaAnalisis.cs
2  // Aplicación de prueba para la clase Analisis.
3  public class PruebaAnalisis
4  {
5      public static void Main( string[] args )
6      {
7          Analisis aplicacion = new Analisis(); // crea un objeto Analisis
8          aplicacion.ProcesarResultadosExamen(); // llama al método p/procesar resultados
9      } // fin de Main
10 } // fin de la clase PruebaAnalisis

```

```

Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 2
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Aprobados: 9
Reprobados: 1
Aumentar colegiatura

```

```

Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 2
Escriba resultado (1 = aprobado, 2 = reprobado): 2
Escriba resultado (1 = aprobado, 2 = reprobado): 2
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 1
Escriba resultado (1 = aprobado, 2 = reprobado): 2
Escriba resultado (1 = aprobado, 2 = reprobado): 2
Aprobados: 5
Reprobados: 5

```

Figura 5.10 | Aplicación de prueba para la clase `Analisis`.

5.9 Operadores de asignación compuestos

C# cuenta con varios *operadores de asignación compuestos* para abreviar las expresiones de asignación. Cualquier instrucción de la forma

```
variable = variable operador expresión;
```

en donde *operador* sea uno de los operadores binarios +, -, *, / o % (u otros que veremos más adelante en el libro) puede escribirse de la siguiente forma:

```
variable operador= expresión;
```

Por ejemplo, podemos abreviar la instrucción

```
c = c + 3;
```

con el *operador de asignación compuesto de suma*, +=, de la siguiente forma:

```
c += 3;
```

El operador += suma el valor de la expresión a la derecha del operador con el valor de la variable a la izquierda del operador, y almacena el resultado en la variable que está a la izquierda del operador. Así, la expresión de asignación c += 3 suma 3 a c. La figura 5.11 muestra los operadores de asignación compuestos aritméticos, expresiones de ejemplo del uso de estos operadores y una explicación de qué es lo que hacen.

5.10 Operadores de incremento y decremento

C# cuenta con dos operadores unarios para sumar 1 o restar 1 al valor de una variable numérica. Éstos son el *operador de incremento* unario, ++ y el *operador de decremento* unario, --, los cuales se sintetizan en la figura 5.12. Una aplicación puede incrementar en 1 el valor de una variable llamada c mediante el uso del operador de incremento, ++, en vez de usar la expresión c = c + 1 o c += 1. Un operador de incremento o decremento que se agrega como prefijo (antepone) a una variable se denomina *operador de incremento prefijo* u *operador de decremento prefijo*, respectivamente. Un operador de incremento o decremento que se agrega como postfijo a (se coloca después de) una variable se denomina *operador de incremento postfijo* u *operador de decremento postfijo*, en forma respectiva.

Al incrementar (o decrementar) una variable con el operador de incremento prefijo (o de decremento prefijo), ésta se incrementa (o decrementa) en 1 y después se utiliza el nuevo valor de la variable en la expresión en la que aparece. Al incrementar (o decrementar) la variable con el operador de incremento postfijo (o de decremento postfijo), se utiliza el valor actual de la variable en la expresión en la que aparece y después se incrementa (o decrementa) en 1.

Operador de asignación	Expresión de ejemplo	Explicación	Asignaciones
<i>Suponga que: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
+=	c += 7	c = c + 7	10 a c
-=	d -= 4	d = d - 4	1 a d
*=	e *= 5	e = e * 5	20 a e
/=	f /= 3	f = f / 3	2 a f
%=	g %=9	g = g % 9	3 a g

Figura 5.11 | Operadores de asignación compuestos aritméticos.

Operador	Se llama	Expresión de ejemplo	Explicación
++	incremento prefijo	++a	Incrementa a en 1 y después utiliza el nuevo valor de a en la expresión en la que reside.
++	incremento postfijo	a++	Usa el valor actual de a en la expresión en la que reside y después incrementa a en 1.
--	decremento prefijo	--b	Decrementa b en 1 y después utiliza el nuevo valor de b en la expresión en la que reside.
--	decremento postfijo	b--	Usa el valor actual de b en la expresión en la que reside y después decrementa b en 1.

Figura 5.12 | Operadores de incremento y decremento.



Buena práctica de programación 5.5

A diferencia de los operadores binarios, los operadores unarios de incremento y decremento deben (por convención) colocarse enseguida de sus operandos, sin espacios entre ellos.

La figura 5.13 demuestra la diferencia entre las versiones de incremento prefijo e incremento postfijo del operador de incremento ++. El operador de decremento (--) funciona de manera similar. Observe que este ejemplo contiene sólo una clase, en donde el método Main se encarga de todo el trabajo de la clase. En este capítulo y en el 4 hemos visto ejemplos que consisten en dos clases: una clase contiene los métodos que realizan tareas útiles y el otro contiene el método Main, que crea un objeto de la otra clase y llama a sus métodos. En este ejemplo sólo queremos demostrar la mecánica del operador ++, por lo que utilizaremos una sola declaración de clase que contiene el método Main. En ocasiones, cuando no tenga sentido tratar de crear una clase reutilizable para demostrar un concepto simple, utilizaremos un ejemplo mecánico, contenido por completo dentro del método Main de una sola clase.

```
1 // Fig. 5.13: Incremento.cs
2 // Los operadores de incremento prefijo e incremento postfijo.
3 using System;
4
5 public class Incremento
6 {
7     public static void Main( string[] args )
8     {
9         int c;
10
11         // demuestra el uso del operador de incremento postfijo
12         c = 5; // asigna 5 a c
13         Console.WriteLine( c ); // imprime 5
14         Console.WriteLine( c++ ); // imprime 5 otra vez, después incrementa
15         Console.WriteLine( c ); // imprime 6
16
17         Console.WriteLine(); // salta una línea
18
19         // demuestra el uso del operador de incremento prefijo
20         c = 5; // asigna 5 a c
21         Console.WriteLine( c ); // imprime 5
22         Console.WriteLine( ++c ); // incrementa y después imprime 6
23         Console.WriteLine( c ); // imprime 6 otra vez
24     } // fin de Main
25 } // fin de la clase Incremento
```

Figura 5.13 | Los operadores de incremento prefijo e incremento postfijo. (Parte I de 2).

```

5
5
6

5
6
6

```

Figura 5.13 | Los operadores de incremento prefijo e incremento postfijo. (Parte 2 de 2).

La línea 12 inicializa la variable `c` con 5 y la línea 13 imprime en pantalla el valor inicial de `c`. La línea 14 imprime en pantalla el valor de la expresión `c++`. Esta expresión realiza la operación de incremento postfijo sobre la variable `c`, por lo que el valor original de `c` (5) se imprime en pantalla y después se incrementa el valor de `c`. Por ende, la línea 14 muestra de nuevo en pantalla el valor inicial de `c` (5). La línea 15 imprime en pantalla el nuevo valor de `c` (6) para demostrar que, en definitiva, se incrementó el valor de la variable en la línea 14.

La línea 20 restablece el valor de `c` a 5 y la línea 21 imprime en pantalla el valor de `c`. La línea 22 imprime el valor de la expresión `++c`. Esta expresión ejecuta la operación de incremento prefijo sobre `c`, por lo que se incrementa su valor y después se imprime el nuevo valor (6). La línea 23 imprime el valor de `c` otra vez para mostrar que sigue siendo 6 después de que se ejecuta la línea 22.

Los operadores de asignación compuestos aritméticos y los operadores de incremento y decremento pueden utilizarse para simplificar las instrucciones. Por ejemplo, las tres instrucciones de asignación en la figura 5.9 (líneas 24, 26 y 29)

```

aprobados = aprobados + 1;
reprobados = reprobados + 1;
contadorEstudiantes = contadorEstudiantes + 1;

```

pueden escribirse en forma más concisa con los operadores de asignación compuestos, de la siguiente manera:

```

aprobados += 1;
reprobados += 1;
contadorEstudiantes += 1;

```

e incluso pueden escribirse en forma aún más concisa con los operadores de incremento prefijo, de la siguiente manera:

```

++aprobados;
++reprobados;
++contadorEstudiantes;

```

o con los operadores de incremento postfijo, de la siguiente forma:

```

aprobados++;
reprobados++;
contadorEstudiantes++;

```

Al incrementar o decrementar una variable en una instrucción por sí sola, las formas de incremento prefijo y de incremento postfijo tienen el mismo efecto, y las formas de decremento prefijo y de decremento postfijo también tienen el mismo efecto. Sólo cuando aparece una variable dentro del contexto de una expresión más grande, el incremento prefijo y el incremento postfijo tienen distintos efectos (y lo mismo ocurre para el decremento prefijo y el decremento postfijo).



Error común de programación 5.6

Tratar de utilizar el operador de incremento o decremento en una expresión que no sea una en la que se pueda asignar un valor es un error de sintaxis. Por ejemplo, si escribimos `++(x + 1)` es un error de sintaxis, ya que `(x + 1)` no es una variable.

La figura 5.14 muestra la precedencia y la asociatividad de los operadores que hemos presentado hasta este momento. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia. La segunda columna describe la asociatividad de los operadores en cada nivel de precedencia. El operador condicional (?:); los operadores unarios de incremento prefijo (++) y de decremento prefijo (--), el más (+) y el menos (-); los operadores de conversión y los operadores de asignación =, +=, -=, *=, /= y %= se asocian de derecha a izquierda. Todos los demás operadores en la tabla de precedencia de operadores de la figura 5.14 se asocian de izquierda a derecha. La tercera columna nombra los grupos de operadores.

5.11 Tipos simples

La tabla en el apéndice L, Los tipos simples, lista los trece *tipos simples* en C#. Al igual que sus lenguajes antecesores C y C++, C# requiere que todas las variables tengan un tipo. Por esta razón, C# se conoce como un *lenguaje fuertemente tipificado*.

En C y C++, los programadores tenían que escribir con frecuencia versiones separadas de las aplicaciones para dar soporte a distintas plataformas de computadoras, ya que no se garantiza que los tipos simples sean idénticos de una computadora a otra. Por ejemplo, un valor `int` en una máquina podría representarse por 16 bits (2 bytes) de memoria, mientras que un valor `int` en otra máquina podría representarse por 32 bits (4 bytes) de memoria. En C#, los valores `int` siempre son de 32 bits (4 bytes). De hecho, *todos* los tipos numéricos en C# tienen tamaños fijos, como se muestra en el apéndice L, Los tipos simples.

Cada uno de los tipos en el apéndice L se lista con su tamaño en bits (hay ocho bits en un byte) y su rango de valores. Como los diseñadores de C# querían que fuera lo más portable posible, utilizaron estándares con reconocimiento internacional para los formatos de caracteres (Unicode; para obtener más información, visite www.unicode.org) y los números de punto flotante (IEEE 754; para obtener más información, visite [group. ieee.org/groups/754/](http://group.ieee.org/groups/754/)).

En la sección 4.5 vimos que las variables de tipos simples que se declaran fuera de un método como campos de una clase reciben de manera automática valores predeterminados, a menos que se inicialicen en forma explícita. Las variables de instancia de los tipos `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` y `decimal` reciben el valor 0 de manera predeterminada. Las variables de instancia de tipo `bool` reciben el valor `false` de manera predeterminada. De manera similar, las variables de instancia de tipos por referencia se inicializan de manera predeterminada con el valor `null`.

5.12 (Opcional) Caso de estudio de ingeniería de software: identificación de los atributos de las clases en el sistema ATM

En la sección 4.11 empezamos la primera etapa de un diseño orientado a objetos (DOO) para nuestro sistema ATM: el análisis del documento de requerimientos y la identificación de las clases necesarias para implementar el sistema. Listamos los sustantivos y las frases nominales en el documento de requerimientos e identificamos una clase separada para cada uno de los que desempeña un papel importante en el sistema ATM. Después modelamos

Operadores	Asociatividad	Tipo
· new ++(<i>postfijo</i>) --(<i>postfijo</i>)	izquierda a derecha	mayor precedencia
++ -- + - (<i>tipo</i>)	derecha a izquierda	prefijo unario
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
?:	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación

Figura 5.14 | Precedencia y asociatividad de los operadores descritos hasta ahora.

las clases y sus relaciones en un diagrama de clases de UML (figura 4.24). Las clases tienen atributos (datos) y operaciones (comportamientos). En los programas en C#, los atributos de las clases se implementan como variables de instancia y propiedades, y las operaciones de las clases se implementan como métodos y propiedades. En esta sección determinaremos muchos de los atributos necesarios en el sistema ATM. En la sección 6.9 examinaremos la forma en que estos atributos representan el estado de un objeto. En la sección 7.15 determinaremos las operaciones para nuestras clases.

Identificación de los atributos

Considere los atributos de algunos objetos reales: los atributos de una persona incluyen su altura, peso y si es zurdo, diestro o ambidiestro. Los atributos de un radio incluyen la estación seleccionada, el volumen seleccionado y si está en AM o FM. Los atributos de un auto incluyen las lecturas de su velocímetro y odómetro, la cantidad de gasolina en su tanque y la velocidad de marcha en la que se encuentra. Los atributos de una computadora personal incluyen su fabricante (por ejemplo, Dell, Gateway, Sun, Apple o IBM), el tipo de pantalla (por ejemplo, LCD o CRT), el tamaño de su memoria principal y el de su disco duro.

Podemos identificar muchos atributos de las clases en nuestro sistema analizando palabras y frases descriptivas en el documento de requerimientos. Por cada palabra o frase que descubramos que desempeña un rol importante en el sistema ATM, creamos un atributo y lo asignamos a una o más de las clases identificadas en la sección 4.11. También creamos atributos para representar los datos adicionales que pueda necesitar una clase, ya que dichas necesidades se van aclarando a lo largo del proceso de diseño.

La figura 5.15 lista las palabras o frases del documento de requerimientos que describen a cada una de las clases. Por ejemplo, el documento de requerimientos describe los pasos que se llevan a cabo para obtener un “monto de retiro”, por lo que listamos “monto” enseguida de la clase *Retiro*.

La figura 5.15 nos conduce a crear un atributo de la clase ATM. Esta clase mantiene información acerca del estado del ATM. La frase “el usuario es autenticado” describe un estado del ATM (en la sección 6.9 hablaremos con detalle sobre los estados), por lo que incluimos `usuarioAutenticado` como un *atributo* `bool` (es decir, un atributo que tiene un valor de `true` o `false`). Este atributo indica si el ATM autenticó con éxito al usuario actual o no; `usuarioAutenticado` debe ser `true` para que el sistema permita al usuario realizar transacciones y acceder a la información de la cuenta. Este atributo nos ayuda a cerciorarnos de la seguridad de los datos en el sistema.

Clase	Palabras y frases descriptivas
ATM	el usuario es autenticado
SolicitudSaldo	número de cuenta
Retiro	número de cuenta monto
Deposito	número de cuenta monto
BaseDatosBanco	[no hay palabras o frases descriptivas]
Cuenta	número de cuenta NIP saldo
Pantalla	[no hay palabras o frases descriptivas]
Teclado	[no hay palabras o frases descriptivas]
DispensadorEfectivo	empieza cada día cargado con 500 billetes de \$20
RanuraDeposito	[no hay palabras o frases descriptivas]

Figura 5.15 | Palabras y frases descriptivas del documento de requerimientos del ATM.

Las clases `SolicitudSaldo`, `Retiro` y `Deposito` comparten un atributo. Cada transacción implica un “número de cuenta” que corresponde a la cuenta del usuario que realiza la transacción. Asignamos el atributo entero `numeroCuenta` a cada clase de transacción para identificar la cuenta a la que se aplica un objeto de la clase.

Las palabras y frases descriptivas en el documento de requerimientos también sugieren ciertas diferencias en los atributos requeridos por cada clase de transacción. El documento de requerimientos indica que para retirar efectivo o depositar fondos, los usuarios deben introducir un “monto” específico de dinero para retirar o depositar, respectivamente. Por ende, asignamos a las clases `Retiro` y `Deposito` un atributo llamado `monto` para almacenar el valor suministrado por el usuario. Los montos de dinero relacionados con un retiro y un depósito son características que definen estas transacciones, que el sistema requiere para que se lleven a cabo. Recuerde que C# representa las cantidades monetarias con el tipo `decimal`. La clase `SolicitudSaldo` no necesita datos adicionales para realizar su tarea; sólo requiere un número de cuenta para indicar la cuenta cuyo saldo hay que obtener.

La clase `Cuenta` tiene varios atributos. El documento de requerimientos establece que cada cuenta de banco tiene un “número de cuenta” y un “NIP”, que el sistema utiliza para identificar las cuentas y autenticar a los usuarios. A la clase `Cuenta` le asignamos dos atributos enteros: `numeroCuenta` y `nip`. El documento de requerimientos también especifica que una cuenta debe mantener un “saldo” del monto de dinero que hay en la cuenta, y que el dinero que el usuario deposita no estará disponible para su retiro sino hasta que el banco verifique la cantidad de efectivo en el sobre de depósito y cualquier cheque que contenga. Sin embargo, una cuenta debe registrar de todas formas el monto de dinero que deposita un usuario. Por lo tanto, decidimos que una cuenta debe representar un saldo utilizando dos atributos `decimal`: `saldoDisponible` y `saldoTotal`. El atributo `saldoDisponible` rastrea el monto de dinero que un usuario puede retirar de la cuenta. El atributo `saldoTotal` se refiere al monto total de dinero que el usuario tiene “en depósito” (es decir, el monto de dinero disponible más el monto de depósitos en efectivo o la cantidad de cheques esperando a ser verificados). Por ejemplo, suponga que un usuario del ATM deposita \$50.00 en efectivo en una cuenta vacía. El atributo `saldoTotal` se incrementa a \$50.00 para registrar el depósito, pero el `saldoDisponible` permanece en \$0 hasta que un empleado del banco cuente el monto de efectivo en el sobre y confirme el total. [Nota: estamos suponiendo que el banco actualiza el atributo `saldoDisponible` de una `Cuenta` poco después de que se realiza la transacción del ATM, en respuesta a la confirmación de que se encontró un monto equivalente a \$50.00 en efectivo en el sobre de depósito. Asumimos que esta actualización se realiza a través de una transacción que realiza el empleado del banco mediante el uso de un sistema bancario distinto al del ATM. Por ende, no hablaremos sobre esta transacción en nuestro caso de estudio.]

La clase `DispensadorEfectivo` tiene un atributo. El documento de requerimientos establece que el dispensador de efectivo “empieza cada día cargado con 500 billetes de \$20”. Éste debe llevar el registro del número de billetes que contiene para determinar si hay suficiente efectivo disponible para satisfacer la demanda de los retiros. Asignamos a la clase `DispensadorEfectivo` el atributo entero `conteo`, el cual se establece al principio en 500.

Para los verdaderos problemas en la industria, no existe garantía alguna de que el documento de requerimientos será lo suficientemente robusto y preciso como para que el diseñador de sistemas orientados a objetos determine todos los atributos, o inclusive todas las clases. La necesidad de clases, atributos y comportamientos adicionales puede irse aclarando a medida que avance el proceso de diseño. A medida que progresemos a través de este caso de estudio, nosotros también seguiremos agregando, modificando y eliminando información acerca de las clases en nuestro sistema.

Modelado de los atributos

El diagrama de clases de la figura 5.16 lista algunos de los atributos para las clases en nuestro sistema; las palabras y frases descriptivas en la figura 5.15 nos ayudaron a identificar estos atributos. Por cuestión de simpleza, la figura 5.16 no muestra las asociaciones entre las clases; en la figura 4.24 mostramos estas asociaciones. Por lo general, los diseñadores de sistemas se encargan de esto. Recuerde que en UML, los atributos de una clase se colocan en el compartimiento de en medio del rectángulo de la clase. Listamos el nombre de cada atributo y su tipo separados por un signo de dos puntos (:), seguido en algunos casos de un signo de igual (=) y de un valor inicial.

Considere el atributo `usuarioAutenticado` de la clase `ATM`:

```
usuarioAutenticado : bool = false
```

La declaración de este atributo contiene tres piezas de información acerca del atributo. El *nombre del atributo* es `usuarioAutenticado`. El *tipo del atributo* es `bool`. En C#, un atributo puede representarse mediante un tipo simple, tal como `bool`, `int`, `double` o `decimal`, o por un tipo de clase (como vimos en el capítulo 4). Hemos

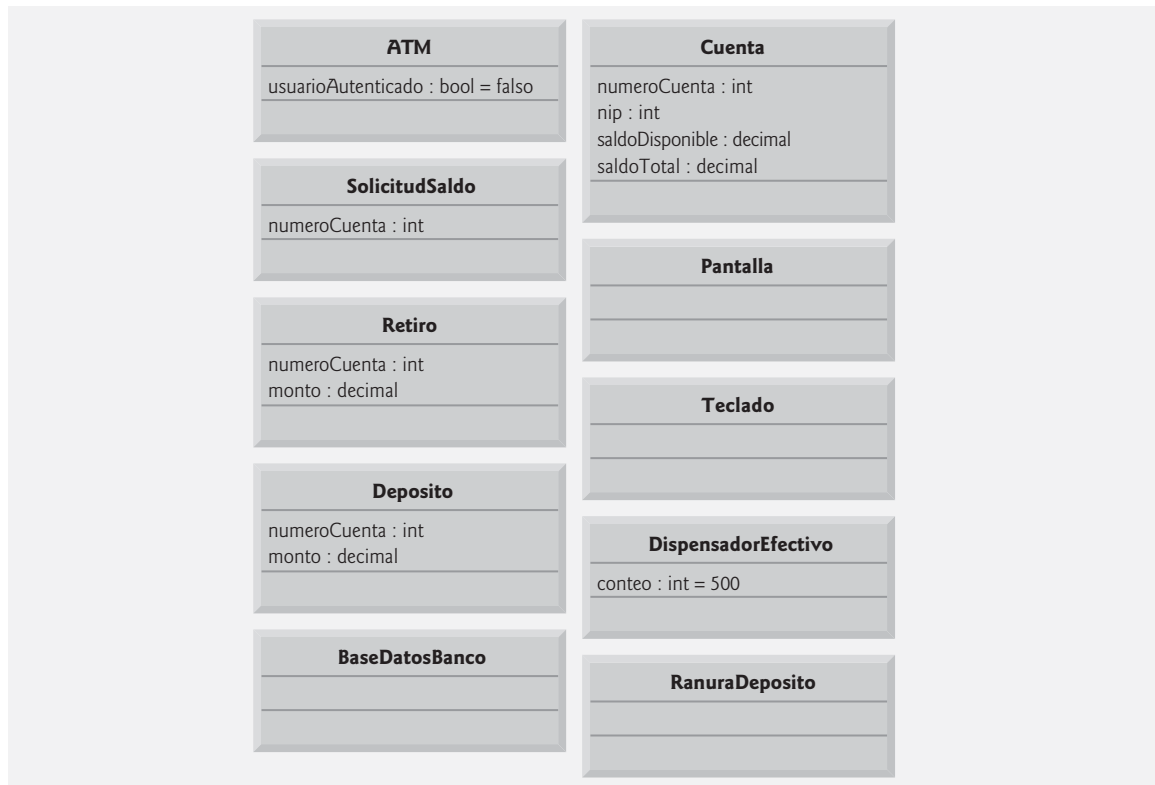


Figura 5.16 | Clases con atributos.

optado por modelar sólo los atributos de tipo simple en la figura 5.16; en breve hablaremos sobre el razonamiento detrás de esta decisión.

También podemos indicar un valor inicial para un atributo. El atributo `usuarioAutenticado` en la clase `ATM` tiene un valor inicial de `false`. Esto indica que al principio el sistema no considera que el usuario está autenticado. Si no se especifica un valor inicial para un atributo, sólo se muestran su nombre y tipo (separados por dos puntos). Por ejemplo, el atributo `numeroCuenta` de la clase `SolicitudSaldo` es un `int`. Aquí no mostramos un valor inicial, ya que el valor de este atributo es un número que todavía no conocemos. Este número se determinará en tiempo de ejecución, con base en el número de cuenta introducido por el usuario actual del ATM.

La figura 5.16 no contiene atributos para las clases `Pantalla`, `Teclado` y `RanuraDeposito`. Éstos son componentes importantes de nuestro sistema para los cuales nuestro proceso de diseño aún no ha revelado ningún atributo. No obstante, tal vez descubramos algunos en las fases restantes de diseño, o cuando implementemos estas clases en C#. Esto es perfectamente normal.



Observación de ingeniería de software 5.1

En las primeras fases del proceso de diseño, a menudo las clases carecen de atributos (y operaciones). Sin embargo, esas clases no deben eliminarse, ya que los atributos (y las operaciones) pueden hacerse evidentes en las fases posteriores de diseño e implementación.

Observe que la figura 5.16 tampoco incluye atributos para la clase `BaseDatosBanco`. Hemos optado por incluir sólo los atributos de tipo simple en la figura 5.16 (y en los diagramas de clases similares a lo largo del caso de estudio). Un atributo de tipo clase se modela con más claridad como una asociación (en particular, una composición) entre la clase con el atributo y la propia clase del atributo. Por ejemplo, el diagrama de clases de la figura 4.24 indica que la clase `BaseDatosBanco` participa en una relación de composición con cero o más objetos `Cuenta`. De esta composición podemos determinar que cuando implementemos el sistema ATM en C#,

tendremos que crear un atributo de la clase `BaseDatosBanco` para almacenar cero o más objetos `Cuenta`. De manera similar, asignaremos atributos a la clase `ATM` que correspondan a sus relaciones de composición con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`. Estos atributos basados en composiciones serían redundantes si los modeláramos en la figura 5.16, ya que las composiciones modeladas en la figura 4.24 ya transmiten el hecho de que la base de datos contiene información acerca de cero o más cuentas, y que un ATM está compuesto por una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos. Por lo general, los desarrolladores de software modelan estas relaciones de todo/parte como asociaciones de composición, en vez de modelarlas como atributos requeridos para implementar las relaciones.

El diagrama de clases de la figura 5.16 proporciona una base sólida para la estructura de nuestro modelo, pero no está completo. En la sección 6.9 identificaremos los estados y las actividades de los objetos en el modelo, y en la sección 7.15 identificaremos las operaciones que realizan los objetos. A medida que presentemos más acerca de UML y del diseño orientado a objetos, continuaremos reforzando la estructura de nuestro modelo.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

5.1 Por lo general identificamos los atributos de las clases en nuestro sistema mediante el análisis de _____ en el documento de requerimientos.

- a) los sustantivos y las frases nominales
- b) las palabras y frases descriptivas
- c) los verbos y las frases verbales
- d) Todo lo anterior.

5.2 ¿Cuál de los siguientes no es un atributo de un aeroplano?

- a) longitud
- b) envergadura
- c) volar
- d) número de asientos

5.3 Describa el significado de la siguiente declaración de un atributo de la clase `DispensadorEfectivo` en el diagrama de clases de la figura 5.16:

```
conteo : int = 500
```

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

5.1 b.

5.2 c. Volar es una operación o comportamiento de un aeroplano, no un atributo.

5.3 Esta declaración indica que el atributo `conteo` es un `int` con un valor inicial de 500; `conteo` lleva el registro del número de billetes disponibles en el `DispensadorEfectivo`, en cualquier momento dado.

5.13 Conclusión

Sólo se requieren tres tipos de estructuras de control (secuencia, selección y repetición) para desarrollar cualquier algoritmo para solucionar un problema. En este capítulo demostramos el uso de la instrucción de selección simple `if`, la instrucción de selección doble `if...else` y la instrucción de repetición `while`. Utilizamos el apilamiento de instrucciones de control para calcular el total y el promedio de un conjunto de calificaciones de estudiantes mediante la repetición controlada por un contador y controlada por un centinela, y utilizamos el anidamiento de instrucciones de control para analizar y tomar decisiones con base en un conjunto de resultados de un examen. Presentamos los operadores de asignación compuestos de C#, así como sus operadores de incremento y decremento. Por último, hablamos sobre los tipos simples. En el capítulo 6, Instrucciones de control: parte 2, continuaremos nuestra discusión acerca de las instrucciones de control, en donde presentaremos las instrucciones `for`, `do...while` y `switch`.

6

Instrucciones de control: parte 2

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Los fundamentos de la repetición controlada por un contador.
- Utilizar las instrucciones de repetición `for` y `do...while` para ejecutar instrucciones repetidas veces en una aplicación.
- Comprender el funcionamiento de la selección múltiple mediante el uso de la instrucción de selección `switch`.
- Cómo se utilizan las instrucciones de control de programa `break` y `continue` para alterar el flujo de control.
- Cómo utilizar los operadores lógicos para formar expresiones condicionales complejas en las instrucciones de control.

No todo lo que puede contarse cuenta, y no todo lo que cuenta puede contarse.

—Albert Einstein

¿Quién puede controlar su destino?

—William Shakespeare

La llave usada siempre brilla.

—Benjamin Franklin

Inteligencia... es la facultad de hacer que los objetos artificiales, en especial las herramientas, creen herramientas.

—Henri Bergson

Toda ventaja en el pasado se juzga a la luz de la cuestión final.

—Demóstenes

- 6.1 Introducción
- 6.2 Fundamentos de la repetición controlada por un contador
- 6.3 Instrucción de repetición `for`
- 6.4 Ejemplos acerca del uso de la instrucción `for`
- 6.5 Instrucción de repetición `do...while`
- 6.6 Instrucción de selección múltiple `switch`
- 6.7 Instrucciones `break` y `continue`
- 6.8 Operadores lógicos
- 6.9 (Opcional) Caso de estudio de ingeniería de software: identificación de los estados y actividades de los objetos en el sistema ATM
- 6.10 Conclusión

6.1 Introducción

En este capítulo demostraremos el resto (excepto una) de las instrucciones de control estructurado de C#: `for`, `do...while` y `switch`. A través de una serie de ejemplos cortos acerca del uso de `while` y `for`, exploraremos los fundamentos de la repetición controlada por un contador. Dedicaremos una parte de este capítulo (y del capítulo 8) a expandir la clase `LibroCalificaciones` que presentamos en los capítulos 4 y 5. En especial, crearemos una versión de la clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar el número de equivalentes de las calificaciones A, B, C, D y F en un conjunto de calificaciones numéricas introducidas por el usuario. Presentaremos las instrucciones de control de programa `break` y `continue`. Hablaremos sobre los operadores lógicos en C#, los cuales le permiten utilizar expresiones condicionales más complejas en las instrucciones de control.

6.2 Fundamentos de la repetición controlada por un contador

Esta sección utiliza la *instrucción de repetición*, presentada en el capítulo 5, para formalizar los elementos requeridos para realizar la repetición controlada por un contador. Este tipo de repetición requiere:

1. una *variable de control* (o contador de ciclo).
2. el *valor inicial* de la variable de control.
3. el *incremento* (o *decremento*) mediante el cual se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como cada *iteración del ciclo*).
4. la *condición de continuación de ciclo* que determina si debe o no continuar el ciclo.

Para ver estos elementos de la repetición controlada por un contador, considere la aplicación de la figura 6.1, que utiliza un ciclo para mostrar en pantalla los números del 1 al 10. Observe que la figura 6.1 sólo contiene un método llamado `Main`, el cual realiza todo el trabajo de la clase. En la mayoría de las aplicaciones de los capítulos 4 y 5 recomendamos el uso de dos archivos separados: uno que declara una clase reutilizable (por ejemplo, `Cuenta`) y otro que instancia uno o más objetos de esa clase (por ejemplo, `PruebaCuenta`) y demuestra su funcionalidad. Sin embargo, en algunas ocasiones es más apropiado crear sólo una clase cuyo método `Main` ilustre en forma concisa un concepto básico. A lo largo de este capítulo utilizaremos varios ejemplos de una sola clase como el de la figura 6.1, para demostrar la mecánica de varias de las instrucciones de control de C#.

En el método `Main` de la figura 6.1 (líneas 7-18), los elementos de la repetición controlada por un contador se definen en las líneas 9, 11 y 14. La línea 9 declara la variable de control (contador) como un `int`, reserva espacio para ella en memoria y establece su valor inicial en 1.

La línea 13 en la instrucción `while` muestra en pantalla el valor de la variable de control `contador` durante cada iteración del ciclo. La línea 14 incrementa la variable de control en 1 para cada iteración del ciclo. La condición de continuación de ciclo en el `while` (línea 11) evalúa si el valor de la variable de control es menor o igual a 10 (el valor final para el cual la condición es `true`). Observe que la aplicación ejecuta el cuerpo de este `while` incluso cuando la variable de control es 10. El ciclo termina cuando la variable de control excede a 10 (es decir, cuando el contador se convierte en 11).

```

1 // Fig. 6.1: ContadorWhile.cs
2 // Repetición controlada por un contador con la instrucción de repetición while.
3 using System;
4
5 public class ContadorWhile
6 {
7     public static void Main( string[] args )
8     {
9         int contador = 1; // declara e inicializa la variable de control
10
11         while ( contador <= 10 ) // condición de continuación de ciclo
12         {
13             Console.Write( "{0} ", contador );
14             contador++; // incrementa la variable de control
15         } // fin de while
16
17         Console.WriteLine(); // imprime en pantalla una nueva línea
18     } // fin de Main
19 } // fin de la clase ContadorWhile

```

1 2 3 4 5 6 7 8 9 10

Figura 6.1 | Repetición controlada por un contador con la instrucción de repetición `while`.



Error común de programación 6.1

Como los valores de punto flotante pueden ser aproximados, el control de ciclos con variables de punto flotante puede producir valores imprecisos del contador y pruebas que terminen en forma imprecisa.



Tip de prevención de errores 6.1

Controle los ciclos de conteo con enteros.



Buena práctica de programación 6.1

Coloque líneas en blanco por encima y por debajo de las instrucciones de control de repetición y selección, y aplique sangrías a los cuerpos de las instrucciones para mejorar la legibilidad.

La aplicación en la figura 6.1 puede hacerse más concisa si se inicializa `contador` en 0 en la línea 9 y se incrementa `contador` en la condición del `while` con el operador de incremento prefijo, como se muestra a continuación:

```

while ( ++contador <= 10 ) // condición de continuación de ciclo
    Console.Write( "{0} ", contador );

```

Este código ahorra una instrucción (y elimina la necesidad de tener llaves alrededor del cuerpo del ciclo), ya que la condición del `while` realiza el incremento antes de evaluar la condición (en la sección 5.10 vimos que la precedencia de `++` es mayor que la de `<=`). Codificar de esa manera condensada podría afectar la legibilidad del código, su depuración, modificación y mantenimiento.



Observación de ingeniería de software 6.1

“Mantener las cosas simples” es un buen consejo para la mayoría del código que usted escribirá.

6.3 Instrucción de repetición `for`

La sección 6.2 presentó los aspectos esenciales de la repetición controlada por un contador. La instrucción `while` puede utilizarse para implementar cualquier ciclo controlado por un contador. C# también cuenta con la

instrucción de repetición for, que especifica los elementos de la repetición controlada por un contador en una sola línea de código. La figura 6.2 reimplementa la aplicación de la figura 6.1, usando la instrucción **for**.

El método **Main** de la aplicación opera de la siguiente manera: cuando la instrucción **for** (líneas 11-12) empieza a ejecutar, la variable de control **contador** se declara e inicializa en 1 (en la sección 6.2 vimos que los primeros dos elementos de la repetición controlada por un contador son la variable de control y su valor inicial). Después, la aplicación verifica la condición de continuación de ciclo, **contador <= 10**, la cual se encuentra entre los dos signos de punto y coma requeridos. Como el valor inicial de **contador** es 1, al principio la condición es verdadera. Por lo tanto, la instrucción del cuerpo (línea 12) muestra en pantalla el valor de la variable de control **contador**, que es 1. Después de ejecutar el cuerpo del ciclo, la aplicación incrementa a **contador** en la expresión **contador++**, la cual aparece a la derecha del segundo signo de punto y coma. Luego, la prueba de continuación de ciclo se ejecuta de nuevo para determinar si la aplicación debe continuar con la siguiente iteración del ciclo. En este punto, el valor de la variable de control es 2, por lo que la condición sigue siendo verdadera (el valor final no se excede), y la aplicación ejecuta la instrucción del cuerpo otra vez (es decir, la siguiente iteración del ciclo). Este proceso continúa hasta que se muestran en pantalla los números del 1 al 10 y el valor de **contador** se vuelve 11, con lo cual falla la prueba de continuación de ciclo y termina la repetición (después de 10 repeticiones del cuerpo del ciclo en la línea 12). Después la aplicación ejecuta la primera instrucción después del **for**; en este caso, la línea 14.

Observe que la figura 6.2 utiliza (en la línea 11) la condición de continuación de ciclo **contador <= 10**. Si especificara por error **contador < 10** como la condición, el ciclo sólo iteraría nueve veces: un error lógico común, conocido como *error de desplazamiento por 1*.



Error común de programación 6.2

Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción de repetición produce un error de desplazamiento por uno.



Buena práctica de programación 6.2

Utilizar el valor final en la condición de una instrucción **while** o **for** con el operador relacional **<=** nos ayuda a evitar los errores de desplazamiento por uno. Para un ciclo que imprime los valores del 1 al 10, la condición de continuación de ciclo debe ser **contador <= 10**, en vez de **contador < 10** (lo cual produce un error de desplazamiento por uno) o **contador < 11** (que es correcto). Muchos programadores prefieren el llamado conteo con base cero, en el cual para contar 10 veces, **contador** se inicializaría a cero y la prueba de continuación de ciclo sería **contador < 10**.

```

1  // Fig. 6.2: ContadorFor.cs
2  // Repetición controlada por un contador con la instrucción de repetición for.
3  using System;
4
5  public class ContadorFor
6  {
7      public static void Main( string[] args )
8      {
9          // el encabezado de la instrucción for incluye la inicialización,
10         // la condición de continuación de ciclo y el incremento
11         for ( int contador = 1; contador <= 10; contador++ )
12             Console.Write( "{0} ", contador );
13
14         Console.WriteLine(); // imprime en pantalla una nueva línea
15     } // fin de Main
16 } // fin de la clase ContadorFor

```

1 2 3 4 5 6 7 8 9 10

Figura 6.2 | Repetición controlada por un contador, con la instrucción de repetición **for**.

variable de control se conoce como el *alcance* de la variable; y define en dónde puede utilizarse en una aplicación. Por ejemplo, una variable local sólo puede utilizarse en el método que declara a esa variable, y sólo a partir del punto de declaración, hasta el final del método. En el capítulo 7, Métodos: un análisis más detallado analizaremos el concepto de alcance.



Error común de programación 6.4

Si la variable de control de una instrucción `for` se declara en la sección de inicialización del encabezado del `for`, y se utiliza fuera del cuerpo del `for` se produce un error de compilación.

Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, C# asume que esta condición siempre será verdadera, con lo cual se crea un ciclo infinito. No puede omitir la expresión de *inicialización* si la aplicación inicializa la variable de control antes del ciclo; en este caso, el alcance de la variable no se limitará al ciclo. Puede omitir la expresión de *incremento* si la aplicación calcula el incremento mediante instrucciones dentro del cuerpo del ciclo, o si no se necesita un incremento. La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`. Por lo tanto, las expresiones

```
contador = contador + 1
contador += 1
++contador
contador++
```

son expresiones de incremento equivalentes en una instrucción `for`. Muchos programadores prefieren `contador++`, ya que es concisa y además un ciclo `for` evalúa su expresión de incremento después de la ejecución de su cuerpo; por lo cual, la forma de incremento postfijo parece más natural. En este caso, la variable que se incrementa no aparece en una expresión más grande, por lo que los operadores de incremento prefijo y postfijo tienen el mismo efecto.



Tip de rendimiento 6.1

Hay una ligera ventaja de rendimiento al utilizar el operador de incremento prefijo, pero si elige el operador de incremento postfijo debido a que parece ser más natural (como en el encabezado de un `for`), los compiladores con optimización generarán código MSIL que utilice la forma más eficiente de todas maneras.



Buena práctica de programación 6.3

En muchos casos, los operadores de incremento prefijo y postfijo se utilizan para sumar 1 a una variable en una instrucción por sí sola. En estos casos el efecto es idéntico, sólo que el operador de incremento prefijo tiene una ligera ventaja de rendimiento. Debido a que el compilador, por lo general, optimiza el código que usted escribe para ayudarlo a obtener el mejor rendimiento, puede usar cualquiera de los dos operadores (prefijo o postfijo) con el que se sienta más cómodo en estas situaciones.



Error común de programación 6.5

Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho del encabezado de un `for` convierte el cuerpo de ese `for` en una instrucción vacía. Por lo general esto es un error lógico.



Tip de prevención de errores 6.2

Los ciclos infinitos ocurren cuando la condición de continuación de ciclo en una instrucción de repetición nunca se vuelve falsa (false). Para evitar esta situación en un ciclo controlado por un contador, debe asegurarse que la variable de control se incremente (o decremente) durante cada iteración del ciclo. En un ciclo controlado por un centinela, asegúrese que el valor centinela se introduzca en algún momento dado.

Las porciones correspondientes a la inicialización, la condición de continuación de ciclo y el incremento de una instrucción `for` pueden contener expresiones aritméticas. Por ejemplo, suponga que $x = 2$ y $y = 10$; si x y y no se modifican en el cuerpo del ciclo, entonces la instrucción

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

es equivalente a la instrucción

```
for ( int j = 2; j <= 80; j += 5 )
```

El incremento de una instrucción `for` también puede ser negativo, en cuyo caso sería un decremento y el ciclo contaría en orden descendente.

Si al principio la condición de continuación de ciclo es `false`, la aplicación no ejecutará el cuerpo de la instrucción `for`, sino que la ejecución continuará con la instrucción después del `for`.

Con frecuencia, las aplicaciones muestran en pantalla el valor de la variable de control o lo utilizan en cálculos dentro del cuerpo del ciclo, pero este uso no es obligatorio. Por lo general, la variable de control se utiliza para controlar la repetición sin que se le mencione dentro del cuerpo del `for`.



Tip de prevención de errores 6.3

Aunque el valor de la variable de control puede cambiarse en el cuerpo de un ciclo `for`, evite hacerlo ya que esta práctica puede llevarlo a cometer errores sutiles.

El diagrama de actividad de UML de la instrucción `for` es similar al de la instrucción `while` (figura 5.4). La figura 6.4 muestra el diagrama de actividad de la instrucción `for` de la figura 6.2. El diagrama hace evidente que la inicialización ocurre sólo una vez antes de evaluar la condición de continuación de ciclo por primera vez, y que el incremento ocurre cada vez que se realiza una iteración y después de ejecutarse la instrucción del cuerpo.

6.4 Ejemplos acerca del uso de la instrucción for

Los siguientes ejemplos muestran técnicas para variar la variable de control en una instrucción `for`. En cada caso escribimos el encabezado apropiado para el `for`. Observe el cambio en el operador relacional para los ciclos en los que se decrementa la variable de control.

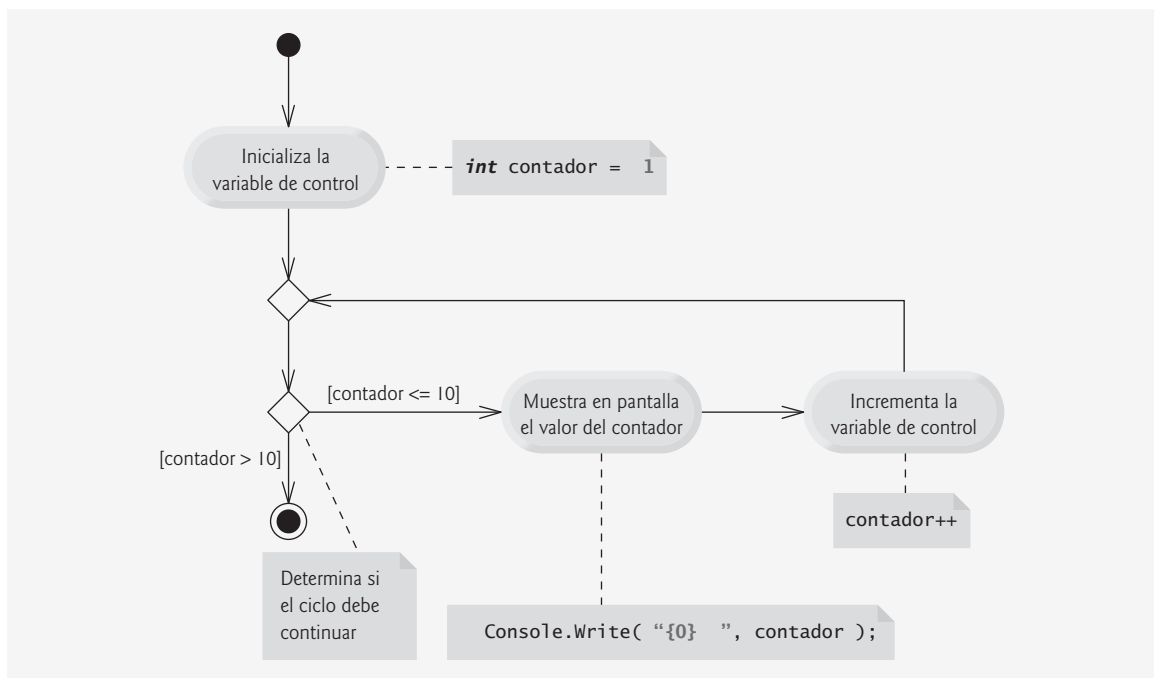


Figura 6.4 | Diagrama de actividad de UML para la instrucción `for` de la figura 6.2.

- a) Modificar la variable de control de 1 a 100, en incrementos de 1.

```
for ( int i = 1; i <= 100; i++ )
```

- b) Modificar la variable de control de 100 a 1, en decrementos de 1.

```
for ( int i = 100; i >= 1; i-- )
```

- c) Modificar la variable de control de 7 a 77, en incrementos de 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

- d) Modificar la variable de control de 20 a 2, en decrementos de 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

- e) Modificar la variable de control sobre la siguiente secuencia de valores: 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

- f) Modificar la variable de control sobre la siguiente secuencia de valores: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```



Error común de programación 6.6

Si no se utiliza el operador relacional apropiado en la condición de continuación de ciclo de un ciclo que cuente en orden descendente (es decir, que utilice `i <= 1` en vez de `i >= 1` en un ciclo que cuente hasta 1 en orden descendente), se produce un error lógico.

Aplicación: suma de los enteros pares del 2 al 20

Ahora consideraremos dos aplicaciones de ejemplo que demuestran los usos simples del `for`. La aplicación en la figura 6.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 20 y almacena el resultado en una variable `int` llamada `total`.

```
1 // Fig. 6.5: Suma.cs
2 // Suma de enteros con la instrucción for.
3 using System;
4
5 public class Suma
6 {
7     public static void Main( string[] args )
8     {
9         int total = 0; // inicializa el total
10
11         // suma los enteros pares del 2 al 20
12         for ( int numero = 2; numero <= 20; numero += 2 )
13             total += numero;
14
15         Console.WriteLine( "La suma es {0}", total ); // muestra los resultados
16     } // fin de Main
17 } // fin de la clase Suma
```

La suma es 110

Figura 6.5 | Suma de enteros con la instrucción `for`.

Las expresiones de *inicialización* e *incremento* pueden ser listas separadas por comas de expresiones que nos permitan utilizar varias expresiones de inicialización, o varias expresiones de incremento. Por ejemplo, el cuerpo de la instrucción `for` en las líneas 12-13 de la figura 6.5 podría mezclarse con la porción del incremento del encabezado `for` mediante el uso de una coma, como se muestra a continuación:

```
for ( int numero = 2; numero <= 20; total += numero, numero += 2 )
    ; // instrucción vacía
```



Buena práctica de programación 6.4

Limite el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.



Buena práctica de programación 6.5

Sólo coloque expresiones que involucren a las variables de control en las secciones de inicialización e incremento de una instrucción `for`. La manipulación de otras variables debe aparecer ya sea antes del ciclo (si se ejecutan sólo una vez, como las instrucciones de inicialización) o en el cuerpo del ciclo (si se ejecutan una vez por cada iteración del ciclo, como las instrucciones de incremento o decremento).

Aplicación: cálculos del interés compuesto

La siguiente aplicación utiliza la instrucción `for` para calcular el interés compuesto. Considere el siguiente problema:

Una persona invierte \$1,000 en una cuenta de ahorros que genera un 5% de interés anual compuesto. Suponiendo que todo el interés se deja para depósito, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Utilice la siguiente fórmula para determinar los montos:

$$a = p(1 + r)^n$$

en donde

p es el monto original invertido (es decir, el capital)

r es la tasa anual de interés (por ejemplo, utilice 0.05 para el 5%)

n es el número de años

a es el monto depositado al final del n -ésimo año.

Este problema implica el uso de un ciclo que realice el cálculo indicado para cada uno de los 10 años que permanecerá el dinero depositado. La solución es la aplicación que se muestra en la figura 6.6. Las líneas 9-11 en el método `Main` declaran las variables `decimal` llamadas `monto` y `capital`, y la variable `double` llamada `tasa`. Las líneas 10-11 también inicializan `capital` con 1000 (es decir, \$1000.00) y `tasa` con 0.05. C# asigna a las constantes de números reales como 0.05 el tipo `double`. De manera similar, C# asigna a las constantes de números enteros como 7 y 1000 el tipo `int`. Cuando `capital` se inicializa con 1000, el valor 1000 de tipo `int` se promueve al tipo `decimal` de manera implícita; no se requiere una conversión.

La línea 14 imprime en pantalla los encabezados para las dos columnas de resultado de la aplicación. La primera columna muestra el año y la segunda columna muestra el monto depositado al final de cada año. Observe que utilizamos el elemento de formato `{1, 20}` para mostrar en pantalla el objeto `string` "Monto depositado". El entero 20 después de la coma indica que el valor a imprimir debe mostrarse con una *anchura de campo* de 20; esto es, `WriteLine` debe mostrar el valor con al menos 20 posiciones de caracteres. Si el valor a imprimir tiene una anchura menor a 20 posiciones de caracteres (en este ejemplo son 17 caracteres), el valor se *justifica a la derecha* en el campo de manera predeterminada (en este caso, se colocan tres espacios en blanco antes del valor). Si el valor año a imprimir tuviera una anchura mayor a cuatro posiciones de caracteres, la anchura del campo se extendería a la derecha para dar cabida a todo el valor; esto desplazaría al campo `monto` a la derecha, con lo que se desacomodarían las columnas ordenadas de nuestros resultados tabulares. Para indicar que el resultado debe *justificarse a la izquierda*, sólo hay que usar una anchura de campo negativa.

La instrucción `for` (líneas 17-25) ejecuta su cuerpo 10 veces, con lo cual la variable de control `año` varía de 1 a 10, en incrementos de 1. Este ciclo termina cuando la variable de control `año` se vuelve 11 (observe que `año` representa a la n en el enunciado del problema).

```

1  // Fig. 6.6: Interes.cs
2  // Cálculos del interés compuesto con for.
3  using System;
4
5  public class Interes
6  {
7      public static void Main( string[] args )
8      {
9          decimal monto; // monto depositado al final de cada año
10         decimal capital = 1000; // monto inicial antes de los intereses
11         double tasa = 0.05; // tasa de interés
12
13         // muestra en pantalla los encabezados
14         Console.WriteLine( "{0}{1,20}", "Año", "Monto depositado" );
15
16         // calcula el monto depositado para cada uno de los 10 años
17         for ( int anio = 1; anio <= 10; anio++ )
18         {
19             // calcula el nuevo monto para el año especificado
20             monto = capital *
21                 ( ( decimal ) Math.Pow( 1.0 + tasa, anio ) );
22
23             // muestra el año y el monto
24             Console.WriteLine( "{0,4}{1,20:C}", anio, monto );
25         } // fin de for
26     } // fin de Main
27 } // fin de la clase Interes

```

Año	Monto depositado
1	\$1,050.00
2	\$1,102.50
3	\$1,157.63
4	\$1,215.51
5	\$1,276.28
6	\$1,340.10
7	\$1,407.10
8	\$1,477.46
9	\$1,551.33
10	\$1,628.89

Figura 6.6 | Cálculos del interés compuesto con for.

Las clases proporcionan métodos que realizan tareas comunes sobre los objetos. De hecho, la mayoría de los métodos a llamar debe pertenecer a un objeto específico. Por ejemplo, en la figura 4.2, llamamos al método `MostrarMensaje` del objeto `miLibroCalificaciones` para imprimir en pantalla un saludo. Muchas clases también cuentan con métodos que realizan tareas comunes y no necesitan pertenecer a un objeto para llamarlos. Dichos métodos se denominan **métodos static**. Por ejemplo, C# no incluye un operador de exponenciación, por lo que los diseñadores de la clase `Math` definieron el método `static` llamado `Pow` para elevar un valor a una potencia. Para llamar a un método `static` debe especificar el nombre de la clase, seguido del operador punto (`.`) y el nombre del método, como en

NombreClase.nombreMétodo (argumentos)

Observe que los métodos `Write` y `WriteLine` de `Console` son métodos `static`. En el capítulo 7 aprenderá a implementar métodos `static` en sus propias clases.

Utilizamos el método `static Pow` de la clase `Math` para realizar el cálculo del interés compuesto en la figura 6.6. `Math.Pow(x, y)` calcula el valor de x elevado a la y -ésima potencia. El método recibe dos argumentos `double`

y devuelve un valor `double`. Las líneas 20-21 realizan el cálculo $a = p(1 + r)^n$, en donde a es el monto, p es el capital, r es la tasa y n es el año. Observe que en este cálculo necesitamos multiplicar un valor `double` (principal) por un valor `double` (el valor de retorno de `Math.Pow`). C# no convierte implícitamente un tipo `double` en un tipo decimal, o viceversa, debido a la posible pérdida de información en cualquiera de las conversiones, por lo que la línea 21 contiene un operador de conversión (`decimal`) que convierte explícitamente el valor de retorno `double` de `Math.Pow` en un `decimal`.

Después de cada cálculo, la línea 24 imprime en pantalla el año y el monto depositado al final de ese año. El año se imprime en una anchura de campo de tres caracteres (según lo especificado por `{0, 4}`). El monto se imprime como un valor de moneda con el elemento de formato `{1, 20:C}`. El número 20 en el elemento de formato indica que el valor debe imprimirse justificado a la derecha, con una anchura de campo de 20 caracteres. El especificador de formato C especifica que el número deberá tener el formato de moneda.

Observe que declaramos las variables `monto` y `capital` de tipo `decimal` en vez de `double`. Recuerde que en la sección 4.10 introdujimos el tipo `decimal` para los cálculos monetarios. También utilizamos `decimal` en la figura 6.6 para este propósito. Tal vez tenga curiosidad acerca de por qué hacemos esto. Estamos tratando con partes fraccionales de dólares y, por ende, necesitamos un tipo que permita puntos decimales en sus valores. Por desgracia, los números de punto flotante de tipo `double` (o `float`) pueden provocar problemas en los cálculos monetarios. Dos montos, en dólares, tipo `double` almacenados en la máquina podrían ser 14.234 (que por lo general se redondea a 14.23 para fines de mostrarlo en pantalla) y 18.673 (que por lo general se redondea a 18.67 para fines de mostrarlo en pantalla). Al sumar estos montos, producen una suma interna de 32.907, que por lo general se redondea a 32.91 para fines de mostrarlo en pantalla. Por lo tanto, sus resultados podrían aparecer como

```

14.23
+ 18.67
-----
32.91

```

pero una persona que sume los números individuales, como se muestran, esperaría que la suma fuera de 32.90. ¡Ya ha sido advertido!



Buena práctica de programación 6.6

No utilice variables de tipo `double` (o `float`) para realizar cálculos monetarios precisos; use mejor el tipo `decimal`. La imprecisión de los números de punto flotante puede provocar errores que resulten en valores monetarios incorrectos.

Observe que el cuerpo de la instrucción `for` contiene el cálculo `1.0 + tasa`, el cual aparece como argumento para el método `Math.Pow`. De hecho, este cálculo produce el mismo resultado cada vez que se realiza una iteración en el ciclo, por lo que repetir el cálculo en todas las iteraciones del ciclo es un desperdicio.



Tip de rendimiento 6.2

En los ciclos, evite cálculos para los cuales el resultado nunca cambie; dichos cálculos por lo general deben colocarse antes del ciclo. [Nota: los compiladores con optimización comúnmente colocan dichos cálculos fuera de los ciclos en el código compilado.]

6.5 Instrucción de repetición `do...while`

La **instrucción de repetición `do...while`** es similar a la instrucción `while`. En el `while`, la aplicación evalúa la condición de continuación de ciclo al principio, antes de ejecutar el cuerpo del ciclo. Si la condición es falsa, el cuerpo nunca se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo; por lo tanto, éste siempre se ejecuta cuando menos una vez. Cuando la instrucción `do...while` termina, la ejecución continúa con la siguiente instrucción en secuencia. La figura 6.7 utiliza un `do...while` (líneas 11-15) para imprimir en pantalla los números del 1 al 10.

La línea 9 declara e inicializa la variable de control `contador`. Al momento de entrar a la instrucción `do...while`, la línea 13 imprime el valor de `contador` y la 14 incrementa esta variable. Después la aplicación evalúa la condición de continuación de ciclo en la parte inferior del ciclo (línea 15). Si la condición es verdadera, el ciclo

continúa a partir de la primera instrucción del cuerpo en el `do...while` (línea 13). Si la condición es falsa el ciclo termina, y la aplicación continúa con la siguiente instrucción después del ciclo.

La figura 6.8 contiene el diagrama de actividad de UML para la instrucción `do...while`. Este diagrama hace ver que la condición de continuación de ciclo no se evalúa sino hasta después de que el ciclo ejecuta su estado de acción, por lo menos una vez. Compare este diagrama de actividad con el de la instrucción `while` (figura 5.4). No es necesario utilizar llaves en la instrucción de repetición `do...while` si sólo hay una instrucción en el cuerpo. No obstante, la mayoría de los programadores incluye las llaves para evitar confusión entre las instrucciones `while` y `do...while`. Por ejemplo,

```
while ( condición )
```

es por lo general la primera línea de una instrucción `while`. Una instrucción `do...while` sin llaves alrededor de un cuerpo con una sola instrucción aparece así:

```
do
    instrucción
while ( condición );
```

lo cual puede ser confuso. El lector podría malinterpretar la última línea [`while( condición );`] como una instrucción `while` que contiene una instrucción vacía (el punto y coma por sí solo). Por ende, una instrucción `do...while` con una sola instrucción en su cuerpo se escribe generalmente así:

```
do
{
    instrucción
} while ( condición );
```



Tip de prevención de errores 6.4

Siempre incluya las llaves en una instrucción `do...while`, aun y cuando no sean necesarias. Esto ayuda a eliminar la ambigüedad entre las instrucciones `while` y `do...while` que contienen sólo una instrucción.

```
1 // Fig. 6.7: PruebaDoWhile.cs
2 // La instrucción de repetición do...while.
3 using System;
4
5 public class PruebaDoWhile
6 {
7     public static void Main( string[] args )
8     {
9         int contador = 1; // inicializa contador
10
11         do
12         {
13             Console.Write( "{0} ", contador );
14             contador++;
15         } while ( contador <= 10 ); // fin de do...while
16
17         Console.WriteLine(); // imprime en pantalla una nueva línea
18     } // fin de Main
19 } // fin de la clase PruebaDoWhile
```

1 2 3 4 5 6 7 8 9 10

Figura 6.7 | La instrucción de repetición `do...while`.

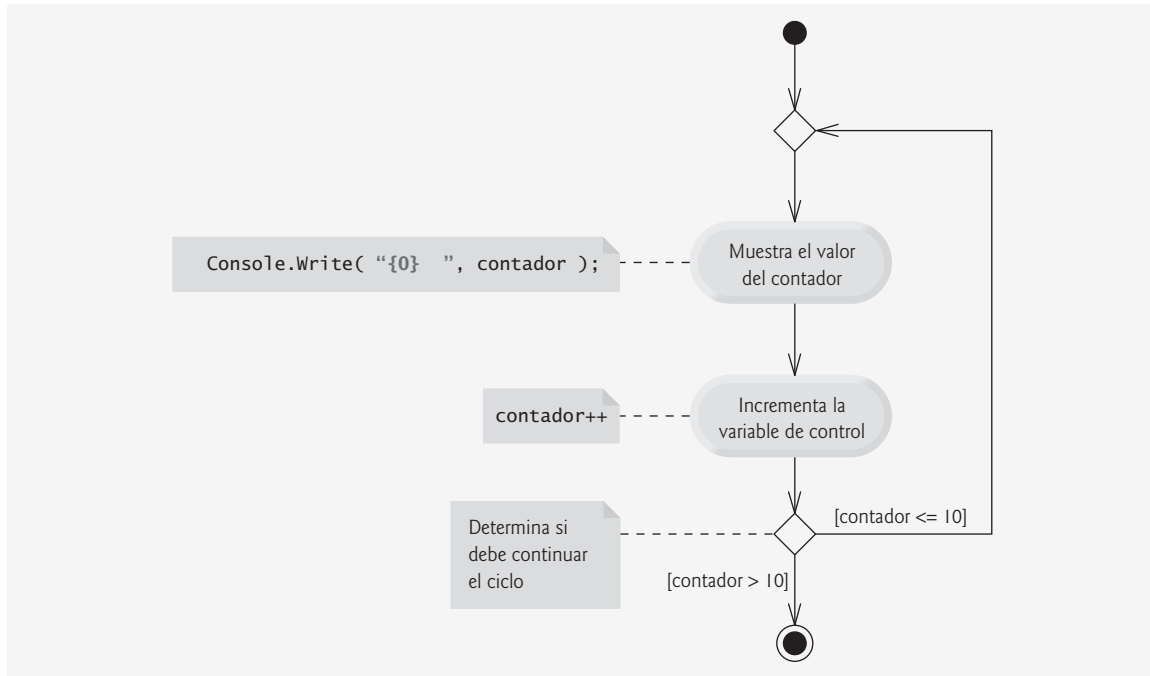


Figura 6.8 | Diagrama de actividad de UML de la instrucción de repetición `do...while`.

6.6 Instrucción de selección múltiple switch

En el capítulo 5 vimos la instrucción de selección simple `if` y la instrucción de selección doble `if...else`. C# cuenta con la instrucción *de selección múltiple* **switch** para realizar distintas acciones, con base en los posibles valores de una expresión. Cada acción se asocia con el valor de una *expresión constante integral* o con una *expresión constante de cadena* que la variable o expresión en la que se basa el `switch` puede asumir. Una expresión constante integral es cualquier expresión que implica constantes tipo carácter y enteras, que se evalúa como un valor entero (es decir, los valores de tipo `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` y `char`). Una expresión constante de cadena es cualquier expresión compuesta por literales de cadena, que siempre produce el mismo objeto `string`.

Clase LibroCalificaciones con la instrucción switch para contar las calificaciones A, B, C, D y F

La figura 6.9 contiene una versión mejorada de la clase `LibroCalificaciones` que presentamos en el capítulo 4, y desarrollamos un poco más en el capítulo 5. La versión de la clase que presentamos ahora no sólo calcula el promedio de un conjunto de calificaciones numéricas introducidas por el usuario, sino que utiliza una instrucción `switch` para determinar si cada calificación es el equivalente de A, B, C, D o F, y para incrementar el contador de la calificación apropiada. La clase también imprime en pantalla un resumen del número de estudiantes que recibieron cada calificación. La figura 6.10 muestra la entrada y la salida de ejemplo de la aplicación `PruebaLibroCalificaciones`, que utiliza la clase `LibroCalificaciones` para procesar un conjunto de calificaciones.

Al igual que las versiones anteriores de la clase, `LibroCalificaciones` (figura 6.9) declara la variable de instancia `nombreCurso` (línea 7), la propiedad `NombreCurso` (líneas 24-34) para acceder a `nombreCurso` y el método `MostrarMensaje` (líneas 37-42) para mostrar un mensaje de bienvenida al usuario. La clase también contiene un constructor (líneas 18-21) que inicializa el nombre del curso.

La clase `LibroCalificaciones` también declara las variables de instancia `total` (línea 8) y `contadorCalif` (línea 9), que llevan el registro de la suma de las calificaciones introducidas por el usuario y el número de calificaciones introducidas, respectivamente. Las líneas 10-14 declaran las variables `contador` para cada categoría de calificaciones. La clase `LibroCalificaciones` mantiene a `total`, `contadorCalif` y a los cinco contadores

de las letras de calificación como variables de instancia, de manera que estas variables puedan utilizarse o modificarse en cualquiera de los métodos de la clase. Observe que el constructor de la clase (líneas 18-21) establece sólo el nombre del curso; las siete variables de instancia restantes son de tipo `int` y se inicializan con 0, de manera predeterminada.

```

1  // Fig. 6.9: LibroCalificaciones.cs
2  // Clase LibroCalificaciones que utiliza la instrucción switch para contar las
   calificaciones A, B, C, D y F.
3  using System;
4
5  public class LibroCalificaciones
6  {
7      private string nombreCurso; // nombre del curso que representa este LibroCalificaciones
8      private int total; // suma de las calificaciones
9      private int contadorCalif; // número de calificaciones introducidas
10     private int contA; // cuenta de calificaciones A
11     private int contB; // cuenta de calificaciones B
12     private int contC; // cuenta de calificaciones C
13     private int contD; // cuenta de calificaciones D
14     private int contF; // cuenta de calificaciones F
15
16     // el constructor inicializa nombreCurso;
17     // las variables de instancia int se inicializan en 0 de manera predeterminada
18     public LibroCalificaciones( string nombre )
19     {
20         NombreCurso = nombre; // inicializa nombreCurso
21     } // fin del constructor
22
23     // propiedad que obtiene (get) y establecer (set) el nombre del curso
24     public string NombreCurso
25     {
26         get
27         {
28             return nombreCurso;
29         } // fin de get
30         set
31         {
32             nombreCurso = value;
33         } // fin de set
34     } // fin de la propiedad NombreCurso
35
36     // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
37     public void MostrarMensaje()
38     {
39         // NombreCurso obtiene el nombre del curso
40         Console.WriteLine( "Bienvenido al libro de calificaciones para\n{0}!\n",
41                             NombreCurso );
42     } // fin del método MostrarMensaje
43
44     // recibe como entrada un número arbitrario de calificaciones del usuario
45     public void IntroducirCalif()
46     {
47         int calificacion; // calificación introducida por el usuario
48         string entrada; // texto introducido por el usuario
49

```

Figura 6.9 | La clase `LibroCalificaciones`, que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F. (Parte I de 3).

```

50 Console.WriteLine( "{0}\n{1}",
51     "Escriba las calificaciones enteras en el rango de 0 a 100.",
52     "Escriba <Ctrl> z y oprima Intro para terminar la captura:" );
53
54 entrada = Console.ReadLine(); // lee la entrada del usuario
55
56 // itera hasta que el usuario escriba el indicador de fin de archivo (<Ctrl> z)
57 while ( entrada != null )
58 {
59     calificacion = Convert.ToInt32( entrada ); // lee la calificación del usuario
60     total += calificacion; // suma calificacion a total
61     contadorCalif++; // incrementa el número de calificaciones
62
63     // llama al método para incrementar el contador apropiado
64     IncrementarContadorLetraCalif( calificacion );
65
66     entrada = Console.ReadLine(); // lee entrada del usuario
67 } // fin de while
68 } // fin del método IntroducirCalif
69
70 // suma 1 al contador apropiado para la calificación especificada
71 private void IncrementarContadorLetraCalif( int calificacion )
72 {
73     // determina cuál calificación se introdujo
74     switch ( calificacion / 10 )
75     {
76         case 9: // la calificación estaba en los 90s
77         case 10: // la calificación fue 100
78             contA++; // incrementa contA
79             break; // necesario para salir de switch
80         case 8: // la calificación estaba entre 80 y 89
81             contB++; // incrementa contB
82             break; // sale del switch
83         case 7: // la calificación estaba entre 70 y 79
84             contC++; // incrementa contC
85             break; // sale del switch
86         case 6: // la calificación estaba entre 60 y 69
87             contD++; // incrementa contD
88             break; // sale del switch
89         default: // la calificación fue menor de 60
90             contF++; // incrementa contF
91             break; // sale del switch
92     } // fin de switch
93 } // fin del método IncrementarContadorLetraCalif
94
95 // muestra un reporte con base en las calificaciones introducidas por el usuario
96 public void MostrarReporteCalif()
97 {
98     Console.WriteLine( "\nReporte de calificaciones:" );
99
100     // si el usuario introdujo cuando menos una calificación...
101     if ( contadorCalif != 0 )
102     {
103         // calcula el promedio de todas las calificaciones introducidas
104         double promedio = ( double ) total / contadorCalif;
105
106         // imprime resumen de resultados

```

Figura 6.9 | La clase LibroCalificaciones, que utiliza una instrucción switch para contar las calificaciones A, B, C, D y F. (Parte 2 de 3).


```

107         Console.WriteLine( "El total de las {0} calificaciones introducidas es {1}",
108             contadorCalif, total );
109         Console.WriteLine( "El promedio de la clase es {0:F2}", promedio );
110         Console.WriteLine( "{0}A: {1}\nB: {2}\nC: {3}\nD: {4}\nF: {5}",
111             "Número de estudiantes que recibieron cada calificación:\n",
112             contA, // muestra el número de calificaciones A
113             contB, // muestra el número de calificaciones B
114             contC, // muestra el número de calificaciones C
115             contD, // muestra el número de calificaciones D
116             contF ); // muestra el número de calificaciones F
117     } // fin de if
118     else // no se introdujeron calificaciones, por lo que se imprime el mensaje apropiado
119         Console.WriteLine( "No se introdujeron calificaciones" );
120 } // fin del método MostrarReporteCalif
121 } // fin de la clase LibroCalificaciones

```

Figura 6.9 | La clase LibroCalificaciones, que utiliza una instrucción switch para contar las calificaciones A, B, C, D y F. (Parte 3 de 3).

La clase LibroCalificaciones contiene tres métodos adicionales: IntroducirCalif, IncrementarContadorLetraCalif y MostrarReporteCalif. El método IntroducirCalif (líneas 45-68) lee un número arbitrario de calificaciones enteras del usuario mediante el uso de la repetición controlada por un centinela, y actualiza las variables de instancia total y contadorCalif. El método IntroducirCalif llama al método IncrementarContadorLetraCalif (líneas 71-93) para actualizar el contador de letra de calificación apropiado para cada calificación introducida. La clase LibroCalificaciones también contiene el método MostrarReporteCalif (líneas 96-120), el cual imprime en pantalla un reporte que contiene el total de todas las calificaciones introducidas, el promedio de las mismas y el número de estudiantes que recibieron cada letra de calificación. Vamos a examinar estos métodos con más detalle.

Las líneas 47-48 en el método IntroducirCalif declaran las variables calificacion y entrada, las cuales almacenan la entrada del usuario, primero como un objeto string (en la variable entrada) y después lo convierten en un int para almacenarlo en la variable calificacion. Las líneas 50-52 piden al usuario que introduzca calificaciones enteras y que escriba <Ctrl> z y después oprima Intro para terminar la captura. La notación <Ctrl> z significa que hay que oprimir al mismo tiempo la tecla Ctrl y la tecla z cuando se escribe en una ventana de Símbolo del sistema. <Ctrl> z es la secuencia de teclas de Windows para escribir el *indicador de fin de archivo*. Ésta es una manera de informar a una aplicación que no hay más datos a introducir. (El indicador de fin de archivo es una combinación de pulsación de teclas independiente del sistema. En muchos sistemas diferentes de Windows, el indicador de fin de archivo se introduce escribiendo <Ctrl> d.) En el capítulo 18, Archivos y flujos, veremos cómo se utiliza el indicador de fin de archivo cuando una aplicación lee su entrada desde un archivo. [Nota: por lo general, Windows muestra los caracteres ^Z en una ventana de Símbolo del sistema cuando se escribe el indicador de fin de archivo, como se muestra en la salida de la figura 6.10.]

La línea 54 utiliza el método ReadLine para obtener la primera línea que introdujo el usuario y almacenarla en la variable entrada. La instrucción while (líneas 57-67) procesa esta entrada del usuario. La condición en la línea 57 comprueba si el valor de entrada es una referencia null. El método ReadLine de la clase Console sólo devolverá null si el usuario escribió un indicador de fin de archivo. Mientras que no se haya escrito el indicador de fin de archivo, entrada no contendrá una referencia null, y la condición pasará.

La línea 59 convierte el objeto cadena que contiene entrada en un tipo int. La línea 60 suma calificacion a total. La línea 61 incrementa contadorCalif. El método MostrarReporteCalif de la clase utiliza estas variables para calcular el promedio de las calificaciones. La línea 64 llama al método IncrementarContadorLetraCalif de la clase (declarado en las líneas 71-93) para incrementar el contador de letra de calificación apropiado, con base en la calificación numérica introducida.

El método IncrementarContadorLetraCalif contiene una instrucción switch (líneas 74-92) que determina cuál contador se debe incrementar. En este ejemplo, suponemos que el usuario introduce una calificación válida en el rango de 0 a 100. Una calificación en el rango de 90 a 100 representa la A, de 80 a 89 la B, de 70 a 79 la C, de 60 a 69 la D y de 0 a 59 la F. La instrucción switch consiste en un bloque que contiene una secuencia de

etiquetas case y una *etiqueta default* opcional. Estas etiquetas se utilizan en este ejemplo para determinar cuál contador a incrementar con base en la calificación.

Cuando el flujo de control llega al `switch`, la aplicación evalúa la expresión entre paréntesis (`calificacion / 10`) que va después de la palabra clave `switch`. A esto se le conoce como *expresión del switch*. La aplicación compara el valor de la expresión del `switch` con cada etiqueta `case`. La expresión del `switch` en la línea 74 realiza una división entera que trunca la parte fraccional del resultado. Así, cuando dividimos cualquier valor en el rango de 0 a 100 entre 10, el resultado siempre es un valor de 0 a 10. Utilizamos varios de estos valores en nuestras etiquetas `case`. Por ejemplo, si el usuario introduce el entero 85, la expresión del `switch` se evalúa como el valor `int 8`. El `switch` compara el 8 con cada `case`. Si hay una concordancia (`case 8`: en la línea 80), la aplicación ejecuta las instrucciones para ese `case`. Para el entero 8, la línea 81 incrementa a `contB`, ya que una calificación dentro del rango de 80 a 89 es B. La *instrucción break* (línea 82) hace que el control del programa continúe con la primera instrucción después del `switch`; en esta aplicación, llegamos al final del cuerpo del método `IncrementarContadorLetraCalif`, por lo que el control regresa a la línea 66 en el método `IntroducirCalif` (la primera línea después de la llamada a `IncrementarContadorLetraCalif`). Esta línea utiliza el método `ReadLine` para leer la siguiente línea introducida por el usuario y la asigna a la variable `entrada`. La línea 67 marca el final del cuerpo del ciclo `while` que recibe como entrada las calificaciones (líneas 57-67), por lo que el control fluye a la condición del `while` (línea 57) para determinar si el ciclo debe continuar ejecutándose, con base en el valor que se acaba de asignar a la variable `entrada`.

Las etiquetas `case` en nuestra instrucción `switch` evalúan explícitamente los valores 10, 9, 8, 7 y 6. Observe las etiquetas `case` en las líneas 76-77, que evalúan los valores 9 y 10 (las cuales representan la calificación A). Al listar las etiquetas `case` de consecutiva, sin instrucciones entre ellas, se permite a los `case` ejecutar el mismo conjunto de instrucciones; cuando la expresión del `switch` se evalúa como 9 o 10, se ejecutan las instrucciones en las líneas 78-79. La instrucción `switch` no cuenta con un mecanismo para evaluar rangos de valores, por lo que todos los valores a evaluar deben listarse en una etiqueta `case` separada. Observe que cada `case` puede tener varias instrucciones. La instrucción `switch` difiere de otras instrucciones de control en cuanto a que no requiere llaves alrededor de varias instrucciones en cada `case`.

En C, C++ y en muchos otros lenguajes de programación que utilizan la instrucción `switch`, no se requiere la instrucción `break` al final de un `case`. Sin instrucciones `break`, cada vez que ocurre una concordancia en el `switch`, se ejecutan las instrucciones para ese `case` y los `case` subsiguientes hasta encontrar una instrucción `break` o el final del `switch`. A menudo a esto se le conoce como que las etiquetas `case` “se pasarían” hacia las instrucciones en las etiquetas `case` subsiguientes. Por lo general esto produce errores lógicos cuando se olvida la instrucción `break`. Por esta razón, C# tiene una regla de “no pasar” las etiquetas `case` en un `switch`: una vez que se ejecutan las instrucciones en un `case`, es obligatorio incluir una instrucción que termine esa etiqueta `case`, como `break`, `return` o `throw` (en el capítulo 12 hablaremos sobre la instrucción `throw`).



Error común de programación 6.7

Olvidar una instrucción break cuando es necesaria en una instrucción switch es un error de sintaxis.

Si no ocurre una concordancia entre el valor de la expresión del `switch` y una etiqueta `case`, se ejecutan las instrucciones después de la etiqueta `default` (líneas 90-91). Utilizamos la etiqueta `default` en este ejemplo para procesar todos los valores de la expresión del `switch` que sean menores de 6; esto es, todas las calificaciones de reprobado. Si no ocurre una concordancia y la instrucción `switch` no contiene una etiqueta `default`, el control del programa simplemente continúa con la primera instrucción después de la instrucción `switch`.

Clase `PruebaLibroCalificaciones` para demostrar la clase `LibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 6.10) crea un objeto `LibroCalificaciones` (líneas 10-11). La línea 13 invoca el método `MostrarMensaje` del objeto para imprimir en pantalla un mensaje de bienvenida para el usuario. La línea 14 invoca el método `IntroducirCalif` del objeto para leer un conjunto de calificaciones del usuario y llevar el registro de la suma de todas las calificaciones introducidas y el número de calificaciones. Recuerde que el método `IntroducirCalif` también llama al método `IncrementarContadorLetraCalif` para llevar el registro del número de estudiantes que recibieron cada letra de calificación. La línea 15 invoca el método `MostrarReporteCalif` de la clase `LibroCalificaciones`, que imprime en pantalla un reporte con base en las calificaciones introducidas. La línea 101 de la clase `LibroCalificaciones` (figura 6.9) determina si el usuario introdujo

cuando menos una calificación; esto evita la división entre cero. De ser así, la línea 104 calcula el promedio de las calificaciones. Entonces, las líneas 107-116 imprimen en pantalla el total de todas las calificaciones, el promedio de la clase y el número de estudiantes que recibieron cada letra de calificación. Si no se introdujeron calificaciones, la línea 119 imprime en pantalla un mensaje apropiado. Los resultados en la figura 6.10 muestran un reporte de calificaciones de ejemplo, con base en 9 calificaciones.

Observe que la clase `PruebaLibroCalificaciones` (figura 6.10) no llama directamente al método `IncrementarContadorLetraCalif` de `LibroCalificaciones` (líneas 71-93 de la figura 6.9). Este método lo utiliza exclusivamente el método `IntroducirCalif` de la clase `LibroCalificaciones` para actualizar el contador de la calificación de letra apropiado, a medida que el usuario introduce cada nueva calificación. El método

```

1  // Fig. 6.10: PruebaLibroCalificaciones.cs
2  // Crea el objeto LibroCalificaciones, recibe como entrada las calificaciones y muestra
   un reporte.
3
4  public class PruebaLibroCalificaciones
5  {
6      public static void Main( string[] args )
7      {
8          // crea el objeto miLibroCalificaciones tipo LibroCalificaciones y
9          // pasa el nombre del curso al constructor
10         LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
11             "CS101 Introducción a la programación en C#" );
12
13         miLibroCalificaciones.MostrarMensaje(); // muestra mensaje de bienvenida
14         miLibroCalificaciones.IntroducirCalif(); // lee calificaciones del usuario
15         miLibroCalificaciones.MostrarReporteCalif(); // muestra un reporte basado en las
           calificaciones
16     } // fin de Main
17 } // fin de la clase PruebaLibroCalificaciones

```

Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C#!

Escriba las calificaciones enteras en el rango de 0 a 100.
Escriba <Ctrl> z y oprima Intro para terminar la captura:

```

99
92
45
100
57
63
76
14
92
^Z

```

Reporte de calificaciones:

El total de las 9 calificaciones introducidas es 638

El promedio de la clase es 70.89

Número de estudiantes que recibieron cada calificación:

```

A: 4
B: 0
C: 1
D: 1
F: 3

```

Figura 6.10 | Crear el objeto `LibroCalificaciones`, recibir como entrada las calificaciones y mostrar en pantalla el reporte de calificaciones.

IncrementarContadorLetraCalif existe únicamente para dar soporte a las operaciones de los demás métodos de la clase LibroCalificaciones, por lo que se declara como `private`. En el capítulo 4 vimos que los métodos que se declaran con el modificador de acceso `private` pueden llamarse sólo por otros métodos de la clase en la que están declarados los métodos `private`. Dichos métodos comúnmente se conocen como *métodos utilitarios* o *métodos ayudantes*, debido a que sólo pueden llamarse mediante otros métodos de esa clase y se utilizan para dar soporte a la operación de esos métodos.

Diagrama de actividad de UML de la instrucción switch

La figura 6.11 muestra el diagrama de actividad de UML para la instrucción `switch` general. Cada conjunto de instrucciones después de una etiqueta `case` debe terminar su ejecución en una instrucción `break` o `return` para terminar la instrucción `switch` después de procesar el `case`. Por lo general, se utilizan instrucciones `break`. La figura 6.11 enfatiza esto al incluir instrucciones `break` en el diagrama de actividad. El diagrama hace ver que la instrucción `break` al final de un `case` ocasiona que el control salga de la instrucción `switch` de inmediato.



Observación de ingeniería de software 6.2

Es conveniente que proporcione una etiqueta `default` en las instrucciones `switch`. Los casos que no se evalúen en forma explícita en un `switch` que carezca de una etiqueta `label` se ignoran. Al incluir una etiqueta `default` usted puede enfatizar en la necesidad de procesar las condiciones excepcionales.

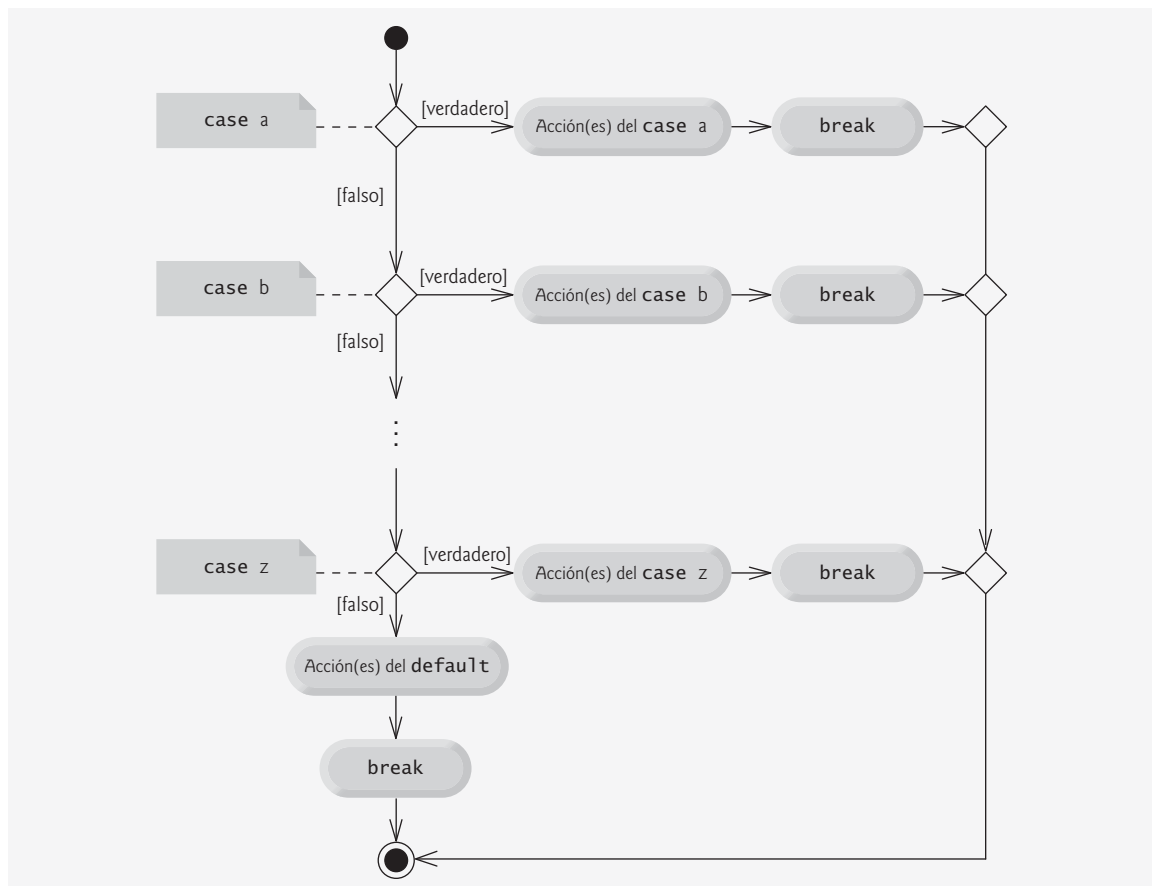


Figura 6.11 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.



Buena práctica de programación 6.7

Aunque cada `case` y la etiqueta `default` en una instrucción `switch` pueden ocurrir en cualquier orden, es conveniente colocar la etiqueta `default` al último para mejorar la legibilidad.

Cuando utilice la instrucción `switch`, recuerde que la expresión después de cada `case` sólo puede ser una expresión constante integral o una expresión constante de cadena; esto es, cualquier combinación de constantes que se evalúe como un valor constante de tipo integral o `string`. Una constante entera es tan solo un valor entero (por ejemplo, `-7`, `0` o `221`). Además, puede utilizar **constantes tipo carácter**: caracteres específicos entre comillas sencillas, como `'A'`, `'7'` o `'$'`, que representan los valores enteros de los caracteres. (En el apéndice D, Conjunto de caracteres ASCII, se muestran los valores enteros de los caracteres en el conjunto de caracteres ASCII, que es un subconjunto del conjunto de caracteres Unicode utilizado por C#.) Una constante `string` es una secuencia de caracteres entre comillas dobles, tal como `"¡Bienvenido a la programación en C#!"`.

La expresión en cada `case` también puede ser una **constante**: una variable que contiene un valor que no cambia durante toda la aplicación. Dicha variable se declara mediante la palabra clave `const` (la cual se describe en el capítulo 7, Métodos: un análisis más detallado). Por otro lado, C# tiene una característica conocida como enumeraciones, que también presentaremos en el capítulo 7. Las constantes de enumeración también pueden utilizarse en etiquetas `case`. En el capítulo 11, Polimorfismo, interfaces y sobrecarga de operadores, presentaremos una manera más elegante de implementar la lógica del `switch`; utilizaremos una técnica llamada polimorfismo para crear aplicaciones que a menudo son más legibles, fáciles de mantener y extender que las aplicaciones que utilizan lógica de `switch`.

6.7 Instrucciones `break` y `continue`

Además de las instrucciones de selección y repetición, C# cuenta con las instrucciones `break` y `continue` para alterar el flujo de control. En la sección anterior mostramos cómo puede utilizarse la instrucción `break` para terminar la ejecución de una instrucción `switch`. En esta sección veremos cómo utilizar `break` para terminar cualquier instrucción de repetición.

Instrucción `break`

Cuando la instrucción `break` se ejecuta en una instrucción `while`, `for`, `do...while`, `switch` o `foreach`, ocasiona la salida inmediata de esa instrucción. La ejecución continúa con la primera instrucción después de la instrucción de control. Los usos comunes de la instrucción `break` son para escapar anticipadamente de una instrucción de repetición, o para omitir el resto de una instrucción `switch` (como en la figura 6.9). La figura 6.12 demuestra el uso de una instrucción `break` para salir de un ciclo `for`.

Cuando la instrucción `if` anidada en la línea 13 dentro de la instrucción `for` (líneas 11-17) determina que conteo es 5, se ejecuta la instrucción `break` en la línea 14. Esto termina la instrucción `for` y la aplicación continúa a la línea 19 (inmediatamente después de la instrucción `for`), que muestra un mensaje indicando el valor de la variable de control cuando terminó el ciclo. El ciclo ejecuta su cuerpo por completo sólo cuatro veces en vez de 10, debido a la instrucción `break`.

Instrucción `continue`

Cuando la **instrucción `continue`** se ejecuta en una instrucción `while`, `for`, `do...while` o `foreach`, omite las instrucciones restantes en el cuerpo del ciclo y continúa con la siguiente iteración del ciclo. En las instrucciones `while` y `do...while`, la aplicación evalúa la prueba de continuación de ciclo justo después de que se ejecuta la instrucción `continue`. En una instrucción `for` se ejecuta la expresión de incremento y después la aplicación evalúa la prueba de continuación de ciclo.

La figura 6.13 utiliza la instrucción `continue` en un ciclo `for` para omitir la instrucción de la línea 14 cuando la instrucción `if` anidada (línea 11) determina que el valor de cuenta es 5. Cuando se ejecuta la instrucción `continue`, el control del programa continúa con el incremento de la variable de control en la instrucción `for` (línea 9).

En la sección 6.3 declaramos que la instrucción `while` puede utilizarse, en la mayoría de los casos, en lugar de `for`. La única excepción ocurre cuando la expresión de incremento en el `while` va después de una instrucción `continue`. En este caso, el incremento no se ejecuta antes de que la aplicación evalúe la condición de continuación de repetición, por lo que el `while` no se ejecuta de la misma manera que el `for`.

```

1 // Fig. 6.12: PruebaBreak.cs
2 // Instrucción break para salir de una instrucción for.
3 using System;
4
5 public class PruebaBreak
6 {
7     public static void Main( string[] args )
8     {
9         int cuenta; // variable de control; también se usa después de que termina el ciclo
10
11         for ( cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
12         {
13             if ( cuenta == 5 ) // si cuenta es 5,
14                 break; // termina el ciclo
15
16             Console.Write( "{0} ", cuenta );
17         } // fin de for
18
19         Console.WriteLine( "\nSalió del ciclo cuando cuenta = {0}", cuenta );
20     } // fin de Main
21 } // fin de la clase PruebaBreak

```

```

1 2 3 4
Salió del ciclo cuando cuenta = 5

```

Figura 6.12 | Instrucción break para salir de una instrucción for.



Observación de ingeniería de software 6.3

Algunos programadores sienten que las instrucciones `break` y `continue` violan la programación estructurada. Ya que pueden lograrse los mismos efectos con las técnicas de programación estructurada, estos programadores prefieren no utilizar instrucciones `break` o `continue`.

```

1 // Fig. 6.13: PruebaContinue.cs
2 // Instrucción continue para terminar una iteración de una instrucción for.
3 using System;
4
5 public class PruebaContinue
6 {
7     public static void Main( string[] args )
8     {
9         for ( int cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
10         {
11             if ( cuenta == 5 ) // si cuenta es 5,
12                 continue; // omite el código restante en el ciclo
13
14             Console.Write( "{0} ", cuenta );
15         } // fin de for
16
17         Console.WriteLine( "\nSe usó continue para omitir imprimir el 5" );
18     } // fin de Main
19 } // fin de la clase PruebaContinue

```

```

1 2 3 4 6 7 8 9 10
Se usó continue para omitir imprimir el 5

```

Figura 6.13 | Instrucción continue para terminar una iteración de una instrucción for.



Observación de ingeniería de software 6.4

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con mejor desempeño. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones excepto las que demanden el mayor rendimiento, aplique la siguiente regla empírica: primero, asegúrese de que su código sea simple y correcto; después hágalo rápido y pequeño, pero sólo si es necesario.

6.8 Operadores lógicos

Cada una de las instrucciones `if`, `if...else`, `while`, `do...while` y `for` requieren una condición para determinar cómo continuar con el flujo de control de una aplicación. Hasta ahora sólo hemos estudiado las *condiciones simples*, como cuenta `<= 10`, numero `!= valorCentinela` y `total > 1000`. Las condiciones simples se expresan en términos de los operadores relacionales `>`, `<`, `>=` y `<=`, y los operadores de igualdad `==` y `!=`. Cada expresión evalúa sólo una condición. Para evaluar condiciones múltiples en el proceso de tomar una decisión, ejecutamos estas pruebas en instrucciones separadas o en instrucciones `if` o `if...else` separadas. Algunas veces, las instrucciones de control requieren condiciones más complejas para determinar el flujo de control de una aplicación.

C# cuenta con los *operadores lógicos* para que usted pueda formar condiciones más complejas, al combinar las condiciones simples. Los operadores lógicos son `&&` (AND condicional), `||` (OR condicional), `&` (AND lógico booleano), `|` (OR inclusivo lógico booleano), `^` (OR exclusivo lógico booleano) y `!` (negación lógica).

Operador AND (&&) condicional

Suponga que deseamos asegurar en cierto punto de una aplicación que dos condiciones son verdaderas, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador `&&` (*AND condicional*) de la siguiente manera:

```
if ( genero == FEMENINO && edad >= 65 )
    mujeresMayores++;
```

Esta instrucción `if` contiene dos condiciones simples. La condición `genero == FEMENINO` compara la variable `genero` con la constante `FEMENINO`. Por ejemplo, esto podría evaluarse para determinar si una persona es mujer. La condición `edad >= 65` podría evaluarse para determinar si una persona es un ciudadano mayor. La instrucción `if` considera la condición combinada

```
genero == FEMENINO && edad >= 65
```

lo cual es verdadero si, y sólo si *ambas* condiciones son verdaderas. Si la condición combinada es verdadera, el cuerpo de la instrucción `if` incrementa a `mujeresMayores` en 1. Si una o ambas condiciones simples son falsas, la aplicación omite el incremento. Algunos programadores consideran que la condición combinada anterior es más legible si se agregan paréntesis redundantes, por ejemplo:

```
( genero == FEMENINO && edad >= 65 )
```

La tabla de la figura 6.14 sintetiza el uso del operador `&&`. Esta tabla muestra las cuatro combinaciones posibles de valores `false` y `true` para *expresión1* y *expresión2*. A dichas tablas se les conoce como *tablas de verdad*. C# evalúa todas las expresiones que incluyen operadores relacionales, de igualdad o lógicos como valores `bool`, los cuales pueden ser `true` o `false`.

Operador OR condicional (||)

Ahora suponga que deseamos asegurar que *cualquiera* o *ambas* condiciones sean verdaderas antes de elegir cierta ruta de ejecución. En este caso, utilizamos el operador `||` (*OR condicional*), como se muestra en el siguiente segmento de una aplicación:

```
if ( ( promedioSemestre >= 90 ) || ( examenFinal >= 90 ) )
    Console.WriteLine ( "La calificación del estudiante es A" );
```

Esta instrucción también contiene dos condiciones simples. La condición `promedioSemestre >= 90` se evalúa para determinar si el estudiante merece una A en el curso debido a que tuvo un sólido rendimiento a lo largo del semestre. La condición `examenFinal >= 90` se evalúa para determinar si el estudiante merece una A en el

expresión1	expresión2	expresión1 && expresión2
false	false	false
false	true	false
true	false	false
true	true	true

Figura 6.14 | Tabla de verdad del operador && (AND condicional).

curso debido a un desempeño sobresaliente en el examen final. Después, la instrucción `if` considera la condición combinada

```
( promedioSemestre >= 90 ) || ( examenFinal >= 90 )
```

y otorga una A al estudiante si una o ambas de las condiciones simples son verdaderas. La única vez que *no* se imprime el mensaje "La calificación del estudiante es A" es cuando ambas de las condiciones simples son falsas. La figura 6.15 es una tabla de verdad para el operador OR condicional (`||`). El operador `&&` tiene mayor precedencia que el operador `||`. Ambos operadores se asocian de izquierda a derecha.

Evaluación en corto circuito de condiciones complejas

Las partes de una expresión que contienen los operadores `&&` o `||` se evalúan sólo hasta que se sabe si la condición es verdadera o falsa. Por ende, la evaluación de la expresión

```
( genero == FEMENINO ) && ( edad >= 65 )
```

se detiene de inmediato si `genero` no es igual a `FEMENINO` (es decir, en este punto es evidente que toda la expresión es `false`) y continúa si `genero` es igual a `FEMENINO` (es decir, toda la expresión podría ser aún `true` si la condición `edad >= 65` es `true`). Esta característica de las expresiones AND y OR condicional se conoce como *evaluación en corto circuito*.



Error común de programación 6.8

En las expresiones que utilizan el operador `&&`, una condición (a la cual le llamamos *condición dependiente*) puede requerir que otra condición sea verdadera para que la evaluación de la condición dependiente tenga significado. En este caso, la condición dependiente debe colocarse después de la otra condición, o podría ocurrir un error. Por ejemplo, en la expresión `( i != 0 ) && ( 10 / i = 2 )`, la segunda condición debe aparecer después de la primera condición, o podría ocurrir un error de división entre cero.

Operadores AND lógico booleano (&) y OR lógico booleano (|)

Los operadores *AND lógico booleano* (`&`) y *OR inclusivo lógico booleano* (`|`) funcionan en forma idéntica a los operadores `&&` (AND condicional) y `||` (OR condicional), con una excepción: los operadores lógicos booleanos siempre evalúan ambos operandos (es decir, no realizan una evaluación en corto circuito). Por lo tanto, la expresión

```
( genero == 1 ) & ( edad >= 65 )
```

expresión1	expresión2	expresión1 expresión2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 6.15 | Tabla de verdad del operador `||` (OR condicional).

evalúa `edad >= 65`, sin importar que `genero` sea igual o no a 1. Esto es útil si el operando derecho del operador AND lógico booleano o del OR inclusivo lógico booleano tiene un *efecto secundario* requerido: la modificación del valor de una variable. Por ejemplo, la expresión

```
( cumpleaños == true ) | ( ++edad >= 65 )
```

garantiza que se evalúe la condición `++edad >= 65`. Por ende, la variable `edad` se incrementa en la expresión anterior, sin importar que la expresión total sea `true` o `false`.



Tip de prevención de errores 6.5

Por cuestión de claridad, evite las expresiones con efectos secundarios en las condiciones. Los efectos secundarios pueden tener una apariencia inteligente, pero pueden hacer que el código sea más difícil de entender y pueden producir errores sutiles.

OR exclusivo lógico booleano (^)

Una condición compleja que contiene el operador *OR exclusivo lógico booleano* (^) es `true` si y sólo si uno de sus operandos es `true` y el otro es `false`. Si ambos operandos son `true` o si ambos son `false`, toda la condición es `false`. La figura 6.16 es una tabla de verdad para el operador OR exclusivo lógico booleano (^). También se garantiza que este operador evaluará ambos operandos.

Operador lógico de negación (!)

El operador ! (*negación lógica*) le permite “invertir” el significado de una condición. A diferencia de los operadores lógicos &&, ||, &, | y ^, que son operadores binarios que combinan dos condiciones, el operador lógico de negación es un operador unario que sólo tiene una condición como operando. Este operador se coloca antes de una condición para elegir una ruta de ejecución si la condición original (sin el operador lógico de negación) es `false`, como en el siguiente segmento de código:

```
if ( ! ( calificacion == valorCentinela ) )
    Console.WriteLine( "La siguiente calificación es {0}", calificacion );
```

este segmento ejecuta la llamada a `WriteLine` sólo si `calificacion` no es igual a `valorCentinela`. Los paréntesis alrededor de la condición `calificacion == valorCentinela` son necesarios, ya que el operador lógico de negación tiene mayor precedencia que el operador de igualdad.

En la mayoría de los casos, puede evitar el uso de la negación lógica si expresa la condición en forma distinta, con un operador relacional o de igualdad apropiado. Por ejemplo, la instrucción anterior también puede escribirse de la siguiente manera:

```
if ( calificacion != valorCentinela )
    Console.WriteLine( "La siguiente calificación es {0}", calificacion );
```

Esta flexibilidad le puede ayudar a expresar una condición de una manera más conveniente. La figura 6.17 es una tabla de verdad para el operador lógico de negación.

Ejemplo de los operadores lógicos

La figura 6.18 demuestra el uso de los operadores lógicos y de los operadores lógicos booleanos; para ello produce sus tablas de verdad. La salida de esta aplicación muestra la expresión que se evalúo y el resultado `bool` de esa expresión. Las líneas 10-14 producen la tabla de verdad para el && (AND condicional). Las líneas 17-21 producen

expresión1	expresión2	expresión1 ^ expresión2
false	false	false
false	true	true
true	false	true
true	true	false

Figura 6.16 | Tabla de verdad del operador ^ (OR exclusivo lógico booleano).

expresión I	!expresión
false	true
true	false

Figura 6.17 | Tabla de verdad del operador ! (negación lógica).

```

1 // Fig. 6.18: OperadoresLogicos.cs
2 // Operadores lógicos.
3 using System;
4
5 public class OperadoresLogicos
6 {
7     public static void Main( string[] args )
8     {
9         // crea la tabla de verdad para el operador && (AND condicional)
10        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
11            "AND condicional (&&)", "false && false", ( false && false ),
12            "false && true", ( false && true ),
13            "true && false", ( true && false ),
14            "true && true", ( true && true ) );
15
16        // crea la tabla de verdad para el operador || (OR condicional)
17        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
18            "OR condicional (||)", "false || false", ( false || false ),
19            "false || true", ( false || true ),
20            "true || false", ( true || false ),
21            "true || true", ( true || true ) );
22
23        // crea la tabla de verdad para el operador & (AND lógico booleano)
24        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
25            "AND lógico booleano (&)", "false & false", ( false & false ),
26            "false & true", ( false & true ),
27            "true & false", ( true & false ),
28            "true & true", ( true & true ) );
29
30        // crea la tabla de verdad para el operador | (OR inclusivo lógico booleano)
31        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
32            "OR inclusivo lógico booleano (|)",
33            "false | false", ( false | false ),
34            "false | true", ( false | true ),
35            "true | false", ( true | false ),
36            "true | true", ( true | true ) );
37
38        // crea la tabla de verdad para el operador ^ (OR exclusivo lógico booleano)
39        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
40            "OR exclusivo lógico booleano (^)",
41            "false ^ false", ( false ^ false ),
42            "false ^ true", ( false ^ true ),
43            "true ^ false", ( true ^ false ),
44            "true ^ true", ( true ^ true ) );
45
46        // crea la tabla de verdad para el operador ! (negación lógica)
47        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}",
48            "Negación lógica (!)", "!false", ( !false ),
49            "!true", ( !true ) );

```

Figura 6.18 | Operadores lógicos. (Parte I de 2).

```

50     } // fin de Main
51 } // fin de la clase OperadoresLogicos

```

```

AND condicional (&&)
false && false: False
false && true: False
true && false: False
true && true: True

OR condicional (||)
false || false: False
false || true: True
true || false: True
true || true: True

AND lógico booleano (&)
false & false: False
false & true: False
true & false: False
true & true: True

OR inclusivo lógico booleano (|)
false | false: False
false | true: True
true | false: True
true | true: True

OR exclusivo lógico booleano (^)
false ^ false: False
false ^ true: True
true ^ false: True
true ^ true: False

Negación lógica (!)
!false: True
!true: False

```

Figura 6.18 | Operadores lógicos. (Parte 2 de 2).

la tabla de verdad para el `||` (OR condicional). Las líneas 24-28 producen la tabla de verdad para el `&` (AND lógico booleano). Las líneas 31-36 producen la tabla de verdad para el `|` (OR inclusivo lógico booleano). Las líneas 39-44 producen la tabla de verdad para el `^` (OR exclusivo lógico booleano). Las líneas 47-49 producen la tabla de verdad para el `!` (negación lógica).

La figura 6.19 muestra la precedencia y la asociatividad de los operadores de C# presentados hasta ahora. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia.

6.9 (Opcional) Caso de estudio de ingeniería de software: identificación de los estados y actividades de los objetos en el sistema ATM

En la sección 5.12 identificamos muchos de los atributos de las clases necesarios para implementar el sistema ATM, y los agregamos al diagrama de clases de la figura 5.16. En esta sección le mostraremos la forma en que estos atributos representan el estado de un objeto. Identificaremos algunos estados clave que pueden ocupar nuestros objetos y hablaremos acerca de cómo cambian de estado, en respuesta a los diversos eventos que ocurren en el sistema. También hablaremos sobre el flujo de trabajo, o *actividades*, que realizan los diversos objetos en el sistema ATM. En esta sección presentaremos los objetos de transacción `SolicitudSaldo` y `Retiro`.

Operadores	Asociatividad	Tipo
. new ++ (<i>postfijo</i>) -- (<i>postfijo</i>)	izquierda a derecha	mayor precedencia
++ -- + - ! (<i>tipo</i>)	derecha a izquierda	prefijo unario
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
&	izquierda a derecha	AND lógico booleano
^	izquierda a derecha	OR exclusivo lógico booleano
	izquierda a derecha	OR inclusivo lógico booleano
&&	izquierda a derecha	AND condicional
	izquierda a derecha	OR condicional
?:	derecha a izquierda	condicional
= += -= *= /= %=	izquierda a derecha	asignación

Figura 6.19 | Precedencia/asociatividad de los operadores descritos hasta ahora.

Diagramas de máquina de estado

Cada objeto en un sistema pasa a través de una serie de estados discretos. El estado de un objeto en un momento dado en el tiempo se indica mediante los valores de los atributos del objeto en ese momento. Los *diagramas de máquina de estado* modelan los estados clave de un objeto y muestran bajo qué circunstancias el objeto cambia de estado. A diferencia de los diagramas de clases que presentamos en las secciones anteriores del caso de estudio, que se enfocaban principalmente en la *estructura* del sistema, los diagramas de máquina de estado modelan parte del *comportamiento* del sistema.

La figura 6.20 es un diagrama de máquina de estado simple, el cual modela dos de los estados en un objeto de la clase ATM. UML representa a cada estado en un diagrama de máquina de estado como un *rectángulo redondeado* con el nombre del estado dentro de éste. Un *círculo relleno* con una punta de flecha designa el *estado inicial*. Recuerde que en el diagrama de clases de la figura 5.16 modelamos esta información de estado como el atributo `bool` de nombre `usuarioAutenticado`. Este atributo se inicializa en `false`, o estado “Usuario no autenticado”, de acuerdo con el diagrama de máquina de estado.

Las flechas indican las *transiciones* entre los estados. Un objeto puede pasar de un estado a otro, en respuesta a los diversos eventos que ocurren en el sistema. El nombre o la descripción del evento que ocasiona una transición se escribe cerca de la línea que corresponde a esa transición. Por ejemplo, el objeto ATM cambia del estado “Usuario no autenticado” al estado “Usuario autenticado”, una vez que la base de datos del banco autentica al usuario. En el documento de requerimientos vimos que para autenticar a un usuario, la base de datos compara el número de cuenta y el NIP introducidos por el usuario con los de la cuenta correspondiente en la base de datos. Si la base de datos indica que el usuario ha introducido un número de cuenta válido y el NIP correcto, el objeto ATM pasa al estado “Usuario autenticado” y cambia su atributo `usuarioAutenticado` al valor `true`. Cuando el usuario sale del sistema al seleccionar la opción “salir” del menú principal, el objeto ATM regresa al estado “Usuario no autenticado” para prepararse para el siguiente usuario del ATM.

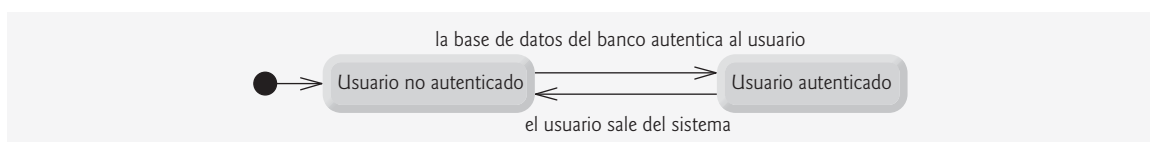


Figura 6.20 | Diagrama de máquina de estado para algunos de los estados del objeto ATM.



Observación de ingeniería de software 6.5

Por lo general, los diseñadores de software no crean diagramas de máquina de estado que muestren todos los posibles estados y transiciones de estados para todos los atributos; simplemente hay demasiados. Lo común es que los diagramas de máquina de estado muestren sólo los estados y las transiciones de estado más importantes o complejas.

Diagramas de actividad

Al igual que un diagrama de máquina de estado, un diagrama de actividad modela los aspectos del comportamiento de un sistema. A diferencia de un diagrama de máquina de estado, un diagrama de actividad modela el flujo de trabajo de un objeto (secuencia de eventos) durante la ejecución de una aplicación. Un diagrama de actividad modela las acciones a realizar y en qué orden las realizará el objeto. Si recuerda, utilizamos los diagramas de actividad de UML para ilustrar el flujo de control para las instrucciones de control que presentamos en los capítulos 5 y 6.

El diagrama de actividad de la figura 6.21 modela las acciones involucradas en la ejecución de una transacción `SolicitudSaldo`. Asumimos que ya se ha inicializado un objeto `SolicitudSaldo` y que ya se ha asignado un número de cuenta válido (el del usuario actual), por lo que el objeto sabe qué saldo extraer de la base de datos. El diagrama incluye las acciones que ocurren después de que el usuario selecciona la opción de solicitud de saldo del menú principal y antes de que el ATM devuelva al usuario al menú principal; un objeto `SolicitudSaldo` no realiza ni inicia estas acciones, por lo que no las modelamos aquí. El diagrama empieza extrayendo de la base de datos el saldo disponible de la cuenta del usuario. Después, `SolicitudSaldo` extrae el saldo total de la cuenta. Por último, la transacción muestra los saldos en la pantalla.

UML representa una acción en un diagrama de actividad como un estado de acción, el cual se modela mediante un rectángulo en el que sus lados izquierdo y derecho se sustituyen por arcos hacia afuera. Cada estado de acción contiene una expresión de acción (por ejemplo, “obtener de la base de datos el saldo disponible de la cuenta del usuario”); que especifica una acción a realizar. Una flecha conecta dos estados de acción, con lo cual indica el orden en que ocurren las acciones representadas por los estados de acción. El círculo relleno (en la parte superior de la figura 6.21) representa el estado inicial de la actividad: el inicio del flujo de trabajo antes de que el objeto realice las acciones modeladas. En este caso, la transacción primero ejecuta la expresión de acción “obtener de la base de datos el saldo disponible de la cuenta del usuario”. Después, la transacción extrae el saldo total. Por último, la transacción muestra ambos saldos en la pantalla. El círculo relleno encerrado en un círculo sin relleno (en la parte inferior de la figura 6.21) representa el estado final: el fin del flujo de trabajo una vez que el objeto realiza las acciones modeladas.

La figura 6.22 muestra un diagrama de actividad para una transacción tipo `Retiro`. Asumimos que ya se ha asignado un número de cuenta válido a un objeto `Retiro`. No modelaremos al usuario seleccionando la opción de retiro del menú principal ni al ATM devolviendo al usuario al menú principal, ya que estas acciones no las

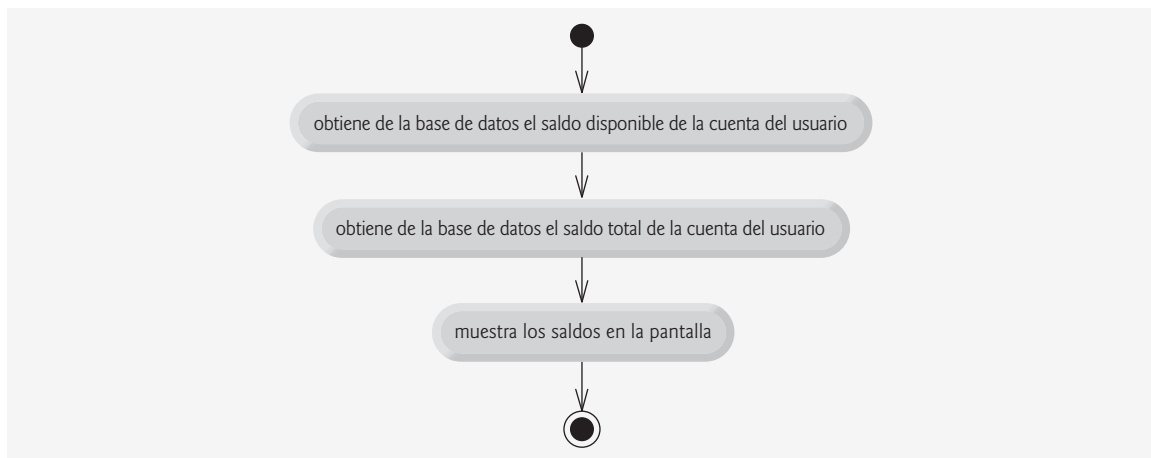


Figura 6.21 | Diagrama de actividad para una transacción tipo `SolicitudSaldo`.

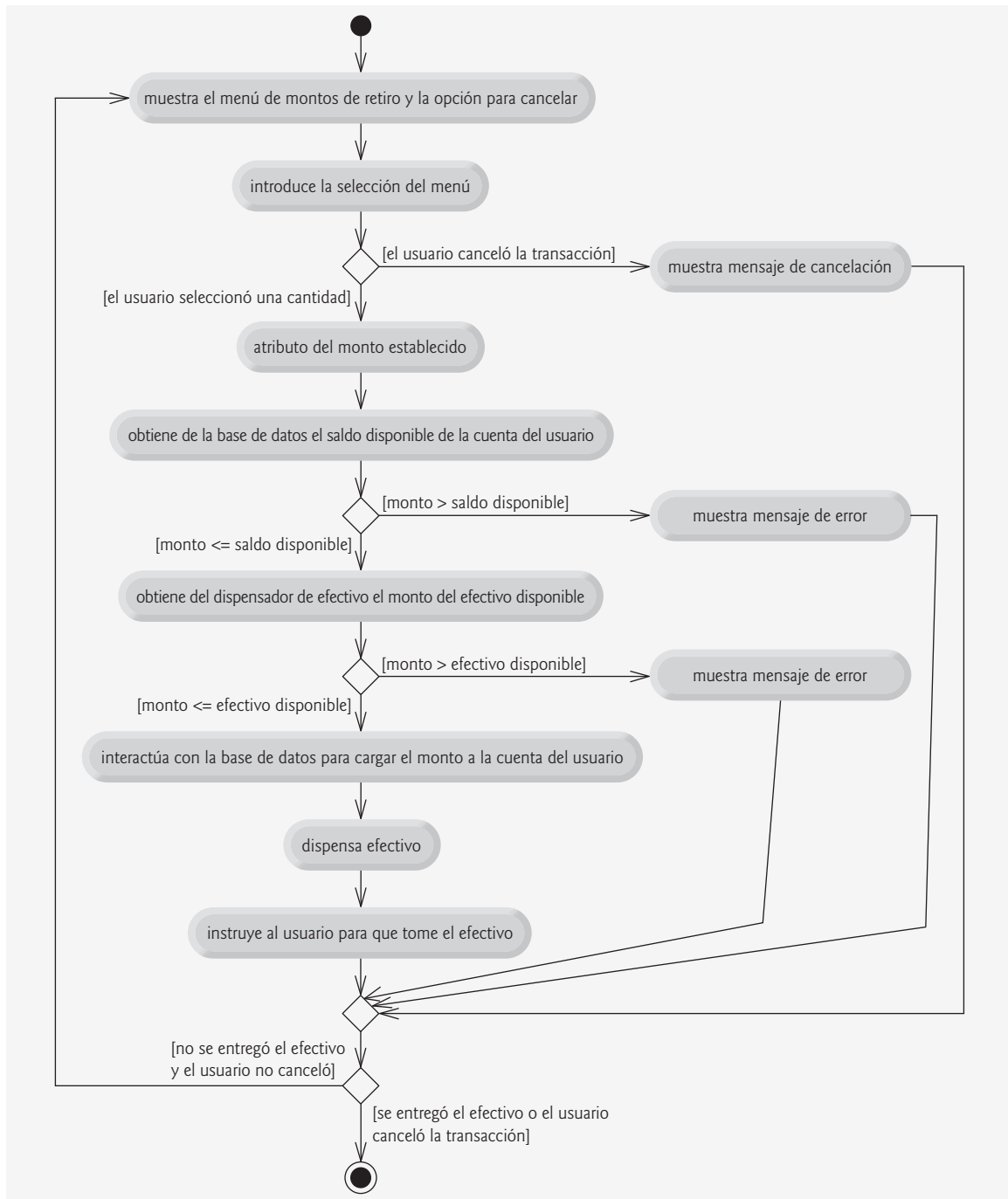


Figura 6.22 | Diagrama de actividad para una transacción tipo Retiro.

realiza un objeto Retiro. La transacción primero muestra un menú de montos estándar de retiro (figura 3.30) y una opción para cancelar la transacción. Después, la transacción recibe como entrada una selección del menú de parte del usuario. Ahora el flujo de actividad llega a un símbolo de decisión. Este punto determina la siguiente acción con base en las condiciones de guardia asociadas. Si el usuario cancela la transacción, el sistema muestra un mensaje apropiado. A continuación, el flujo de cancelación llega a un símbolo de fusión, en donde este flujo de

actividad se une con los otros posibles flujos de actividad de la transacción (que veremos en breve). Observe que una fusión puede tener cualquier número de flechas de transición entrantes, pero sólo una flecha de transición saliente. La decisión en la parte inferior del diagrama determina si la transacción debe repetirse desde el principio. Cuando el usuario cancela la transacción, la condición de guardia “se entregó el efectivo o el usuario canceló la transacción” es verdadera, por lo que el control pasa al estado final de la actividad.

Si el usuario selecciona un monto de retiro del menú, `monto` (un atributo de la clase `Retiro`, que se modeló originalmente en la figura 5.16) se establece al valor seleccionado por el usuario. A continuación, la transacción obtiene el saldo disponible de la cuenta del usuario (es decir, el atributo `saldoDisponible` del objeto `Cuenta` del usuario) de la base de datos. Después, el flujo de actividad llega a otra decisión. Si el monto de retiro requerido excede al saldo disponible del usuario, el sistema muestra un mensaje de error apropiado, en el cual informa al usuario sobre el problema. Enseguida, el control se fusiona con los demás flujos de actividad antes de llegar a la decisión que está en la parte inferior del diagrama. La condición de guardia “no se entregó el efectivo y el usuario no canceló” es verdadera, por lo que el flujo de actividad regresa a la parte superior del diagrama, y la transacción pide al usuario que introduzca un nuevo monto.

Si el monto de retiro solicitado es menor o igual al saldo disponible del usuario, la transacción evalúa si el dispensador de efectivo tiene suficiente efectivo para satisfacer la solicitud de retiro. Si éste no es el caso, la transacción muestra un mensaje de error apropiado y pasa hacia la fusión antes de llegar a la decisión final. Como el efectivo no se entregó, el flujo de actividad regresa al inicio del diagrama de actividad y la transacción pide al usuario que seleccione un nuevo monto. Si hay suficiente efectivo disponible, la transacción interactúa con la base de datos para cargar el monto retirado de la cuenta del usuario (es decir, restar el monto *tanto* del atributo `saldoDisponible` como del atributo `saldoTotal` del objeto `Cuenta` del usuario). Después la transacción entrega el monto deseado de efectivo e instruye al usuario para que lo tome.

Luego, el flujo principal de actividad se fusiona con los dos flujos de error y el flujo de cancelación. En este caso se entregó efectivo, por lo que el flujo de actividad llega al estado final.

Hemos llevado a cabo los primeros pasos para modelar el comportamiento del sistema ATM y hemos mostrado cómo los atributos de un objeto afectan sus actividades. En la sección 7.15 investigaremos las operaciones de nuestras clases para crear un modelo más completo del comportamiento del sistema.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 6.1** Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: los diagramas de máquina de estado modelan los aspectos estructurales de un sistema.
- 6.2** Un diagrama de actividad modela las (los) _____ que realiza un objeto y el orden en el que las (los) realiza.
- a) acciones
 - b) atributos
 - c) estados
 - d) transiciones de estado
- 6.3** Con base en el documento de requerimientos, cree un diagrama de actividad para una transacción de depósito.

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 6.1** Falso. Los diagramas de máquina de estado modelan algunos de los comportamientos del sistema.
- 6.2** a.
- 6.3** La figura 6.23 presenta un diagrama de actividad para una transacción de depósito. El diagrama modela las acciones que ocurren una vez que el usuario selecciona la opción de depósito del menú principal, y antes de que el ATM regrese al usuario al menú principal. Recuerde que una parte del proceso de recibir un monto de depósito de parte del usuario implica convertir un número entero de centavos a una cantidad en dólares. Recuerde también que para acreditar un monto de depósito a una cuenta sólo hay que incrementar el atributo `saldoTotal` del objeto `Cuenta` del usuario. El banco actualiza el atributo `saldoDisponible` del objeto `Cuenta` del usuario sólo después de confirmar el monto de efectivo en el sobre de depósito y después de verificar los cheques que haya incluido; esto ocurre en forma independiente del sistema ATM.

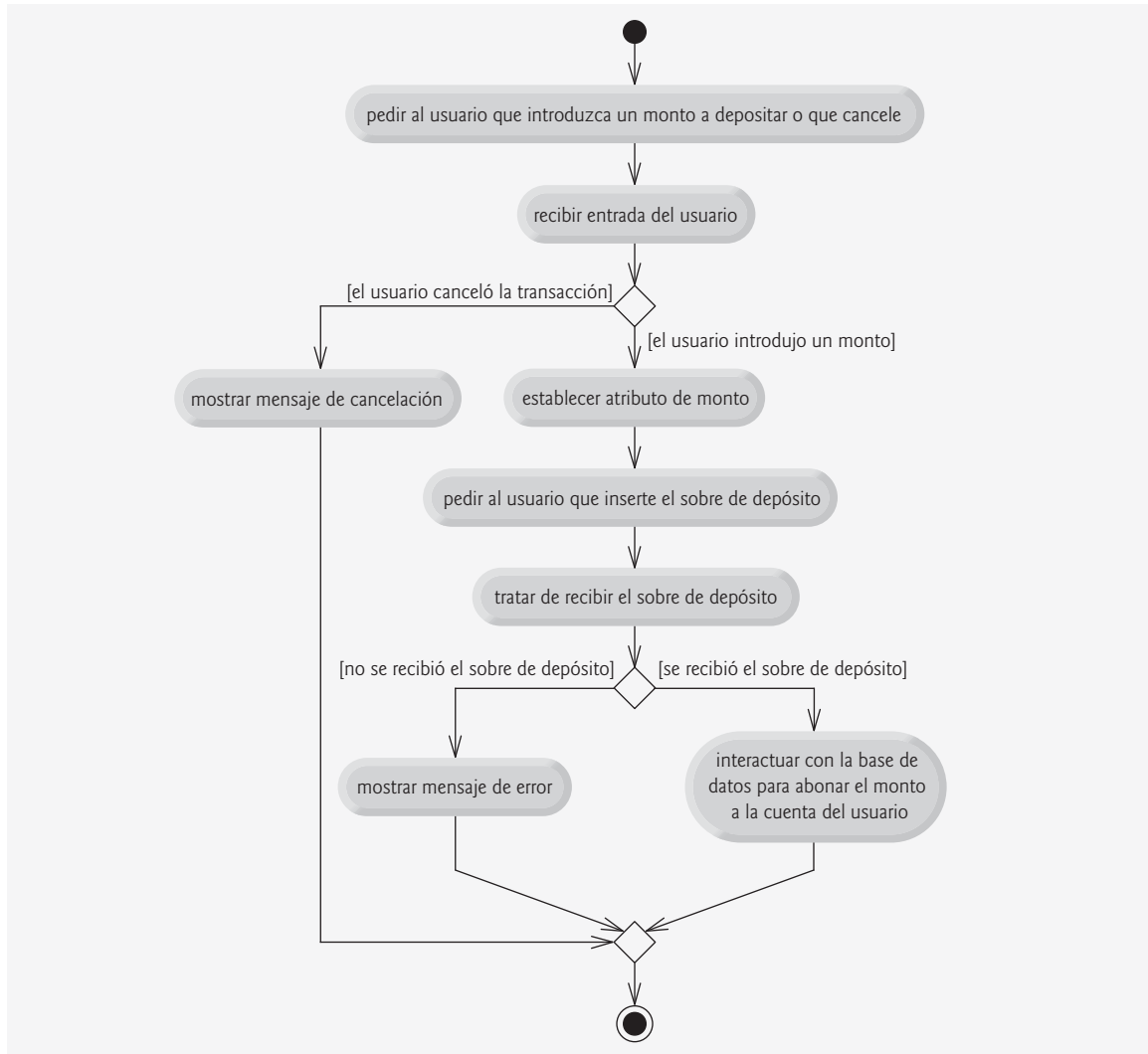


Figura 6.23 | Diagrama de actividad para una transacción de Depósito.

6.10 Conclusión

En el capítulo 5 vimos las instrucciones de control `if`, `if...else` y `while`. En este capítulo vimos las instrucciones de control `for`, `do...while` y `switch` (en el capítulo 8 hablaremos sobre la instrucción `foreach`). Usted aprendió que cualquier algoritmo puede desarrollarse mediante el uso de combinaciones de instrucciones de secuencia (es decir, instrucciones que se listan en el orden en el que deben ejecutarse), las tres instrucciones de selección (`if`, `if...else` y `switch`) y las cuatro instrucciones de repetición (`while`, `do...while`, `for` y `foreach`). También vio que las instrucciones `for` y `do...while` son tan sólo maneras más convenientes de expresar ciertos tipos de repetición. De manera similar, demostramos que la instrucción `switch` es una notación conveniente para la selección múltiple, en vez de utilizar instrucciones `if...else` anidadas. Hablamos sobre cómo puede usted combinar varias instrucciones de control mediante el apilamiento y el anidamiento. Le mostramos cómo usar las instrucciones `break` y `continue` para alterar el flujo de control en las instrucciones de repetición. También aprendió acerca de los operadores lógicos, los cuales le permiten utilizar expresiones condicionales más complejas en las instrucciones de control.

En el capítulo 4 presentamos los conceptos básicos de los objetos, las clases y los métodos. En los capítulos 5 y 6 proporcionamos una introducción detallada a las instrucciones de control que usted utilizará para especificar la lógica de la aplicación en métodos. En el capítulo 7 examinaremos los métodos con más detalle.

7

Métodos: un análisis más detallado

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Cómo se asocian los métodos y las variables `static` con toda una clase, en vez de asociarse con instancias específicas de la clase.
- Cómo se soporta el mecanismo de llamada/retorno de los métodos mediante la pila de llamadas a métodos y los registros de activación.
- Cómo utilizar la generación de números aleatorios para implementar aplicaciones para juegos.
- A comprender cómo se limita la visibilidad de las declaraciones a regiones específicas de las aplicaciones.
- Qué es la sobrecarga de métodos y cómo crear métodos sobrecargados.
- Qué son los métodos recursivos.
- Las diferencias entre pasar los argumentos de los métodos por valor y por referencia.

La forma siempre sigue a la función.

—Louis Henri Sullivan

E pluribus unum.

(Uno compuesto de muchos.)

—Virgilio

*Oh! Recuerdos del ayer, tratar
de regresar el tiempo.*

—William Shakespeare

Lláname Ismael.

—Herman Melville

*Cuando me digas eso,
¡sonríe!*

—Owen Wister

Respóndeme en una palabra.

—William Shakespeare

*Hay un punto en el cual
los métodos se devoran a sí mismos.*

—Frantz Fanon

*La vida sólo puede comprenderse
al revés; pero debe vivirse
hacia delante.*

—Søren Kierkegaard

- 7.1 Introducción
- 7.2 Empaquetamiento de código en C#
- 7.3 Métodos `static`, variables `static` y la clase `Math`
- 7.4 Declaración de métodos con múltiples parámetros
- 7.5 Notas acerca de cómo declarar y utilizar los métodos
- 7.6 La pila de llamadas a los métodos y los registros de activación
- 7.7 Promoción y conversión de argumentos
- 7.8 La Biblioteca de clases del .NET Framework
- 7.9 Caso de estudio: generación de números aleatorios
 - 7.9.1 Escalar y desplazar números aleatorios
 - 7.9.2 Repetitividad de números aleatorios para prueba y depuración
- 7.10 Caso de estudio: juego de probabilidad (introducción a las enumeraciones)
- 7.11 Alcance de las declaraciones
- 7.12 Sobrecarga de métodos
- 7.13 Recursividad
- 7.14 Paso de argumentos: diferencia entre paso por valor y paso por referencia
- 7.15 (Opcional) Caso de estudio de ingeniería de software: identificación de las operaciones de las clases en el sistema ATM
- 7.16 Conclusión

7.1 Introducción

En el capítulo 4 presentamos los métodos; y en este capítulo los estudiaremos detalladamente. Nos concentraremos en cómo declarar y utilizar métodos para facilitar el diseño, la implementación, la operación y el mantenimiento de aplicaciones extensas.

En breve verá que es posible que ciertos métodos, conocidos como métodos `static`, puedan llamarse sin necesidad de que exista un objeto de la clase a la que pertenecen. Aprenderá a declarar un método con más de un parámetro. También aprenderá acerca de cómo C# es capaz de llevar el rastro de cuál método ejecuta en un momento dado, cómo se pasan los argumentos de tipo por valor y de tipo por referencia a los métodos, cómo se mantienen las variables locales de los métodos en memoria y cómo sabe un método a dónde regresar una vez que termina su ejecución.

Hablaremos sobre las técnicas de *simulación* mediante la generación de números aleatorios y desarrollaremos una versión de un juego de dados conocido como “craps”, el cual utiliza la mayoría de las técnicas de programación que usted ha aprendido hasta este capítulo. Además, aprenderá a declarar valores que no pueden cambiar (es decir, constantes). También sabrá cómo escribir métodos que se llaman a sí mismos; a esto se le conoce como *recursividad*.

Muchas de las clases que utilizará o creará mientras desarrolla aplicaciones tendrán más de un método del mismo nombre. Esta técnica, conocida como *sobrecarga de operadores*, se utiliza para implementar métodos que realizan tareas similares, pero con distintos tipos o distintos números de argumentos.

7.2 Empaquetamiento de código en C#

Las tres formas comunes de empaquetar código son los métodos, las clases y los espacios de nombres. Las aplicaciones en C# se escriben mediante la combinación de nuevos métodos y clases que usted escribe, con los métodos y clases predefinidos que están disponibles en la *Biblioteca de clases del .NET Framework* (también conocida como *FCL*) y en varias bibliotecas más. A menudo, las clases relacionadas se agrupan en espacios de nombres y se compilan en bibliotecas de clases, de manera que puedan reutilizarse en otras aplicaciones. En el capítulo 9 aprenderá a crear sus propios espacios de nombres y bibliotecas de clases. La FCL cuenta con muchas clases predefinidas que contienen métodos para realizar cálculos matemáticos comunes, manipulaciones de cadenas, manipulaciones de

caracteres, operaciones de entrada/salida, operaciones de bases de datos, operaciones de red, procesamiento de archivos, comprobación de errores y muchas otras operaciones útiles.



Buena práctica de programación 7.1

Familiarícese con las clases y los métodos que proporciona la FCL (en inglés: msdn2.microsoft.com/en-us/library/ms229335; en español: msdn.microsoft.com/library/spa/default.asp?url=/library/SPA/cpref/html/cpref_start.asp). En la sección 7.8 presentaremos las generalidades acerca de varios espacios de nombres comunes.



Observación de ingeniería de software 7.1

No trate de “reinventar la rueda”. Siempre que pueda, reutilice las clases y métodos de la FCL. Esto reduce el tiempo de desarrollo de las aplicaciones y evita introducir errores de programación.



Observación de ingeniería de software 7.2

Para promover la reutilización del software, todo método debe limitarse a desempeñar una sola tarea bien definida, y el nombre del método debe expresar esa tarea con efectividad. Dichos métodos facilitan la escritura, depuración, mantenimiento y modificación de las aplicaciones.



Tip de prevención de errores 7.1

Un método pequeño que realiza una tarea es más fácil de probar y depurar que un método más grande que realiza muchas tareas.



Observación de ingeniería de software 7.3

Si no puede elegir un nombre conciso que exprese la tarea de un método, su método podría estar tratando de realizar demasiadas tareas. Por lo general es mejor dividir un método de este tipo en varios métodos más pequeños.

7.3 Métodos `static`, variables `static` y la clase `Math`

Aunque la mayoría de los métodos se ejecuta sobre objetos específicos, en respuesta a las llamadas a estos métodos, éste no siempre es el caso. Algunas veces, un método realiza una tarea que no depende del contenido de ningún objeto. Dicho método se aplica a la clase en la que está declarado como un todo, y se le denomina método `static`. Es común que una clase contenga un grupo de métodos `static` para realizar tareas comunes. Por ejemplo, si recuerda, en la figura 6.6 utilizamos el método `static` `Pow` de la clase `Math` para elevar un valor a una potencia. Para declarar un método como `static` coloque la palabra clave `static` antes del tipo de valor de retorno en la declaración del método. Para llamar a cualquier método `static`, hay que especificar el nombre de la clase en la que está declarado, seguido del operador punto (`.`) y el nombre del método, por ejemplo:

```
NombreClase.nombreMétodo( argumentos )
```

Aquí utilizaremos varios métodos de la clase `Math` para presentar el concepto de los métodos `static`. La clase `Math` (del espacio de nombres `System`) cuenta con una colección de métodos que le permiten realizar cálculos matemáticos comunes. Por ejemplo, puede calcular la raíz cuadrada de `900.0` con la llamada al método `static`

```
Math.Sqrt( 900.0 )
```

La expresión anterior se evalúa como `30.0`. El método `Sqrt` recibe un argumento de tipo `double` y devuelve un resultado del mismo tipo. Para imprimir en la ventana de consola el valor de la anterior llamada al método, podría escribir la instrucción

```
Console.WriteLine( Math.Sqrt( 900.0 ) );
```

En esta instrucción, el valor que devuelve `Sqrt` se convierte en el argumento para el método `WriteLine`. Observe que: no creamos un objeto `Math` antes de llamar al método `Sqrt` y que *todos* los métodos de `Math` son `static`; por lo tanto, para llamar a cada uno de ellos se antepone el nombre de la clase `Math` y el operador punto (`.`) al nombre

del método. De manera similar, el método `WriteLine` de `Console` es un método `static` de la clase `Console`, por lo que para invocarlo anteponeamos a su nombre al de la clase `Console` y el operador punto (`.`)

Los argumentos para los métodos pueden ser constantes, variables o expresiones. Si `c = 13`, `d = 3.0` y `f = 4.0`, entonces la instrucción

```
Console.WriteLine( Math.Sqrt( c + d * f ) );
```

calcula e imprime la raíz cuadrada de $13.0 + 3.0 * 4.0 = 25.0$; a saber, 5.0 . La figura 7.1 sintetiza varios métodos de la clase `Math`. En la figura, `x` y `y` son de tipo `double`.

Constantes `PI` y `E` de la clase `Math`

La clase `Math` también declara dos variables `static`, que representan a las constantes matemáticas de uso común: **`Math.PI`** y **`Math.E`**. La constante `Math.PI` (3.14159265358979323846) es la proporción de la circunferencia de un círculo en relación con su diámetro. La constante `Math.E` (2.7182818284590452354) es el valor base para los logaritmos naturales (se calcula mediante el método `static Log` de `Math`). Estas variables se declaran en la clase `Math` con los modificadores `public` y `const`. Al hacerlas `public` se permite a otros programadores utilizarlas en sus propias clases. Cualquier variable declarada con la palabra clave **`const`** es una constante; su valor no puede cambiarse una vez declarada. Tanto `PI` como `E` se declaran `const`, ya que sus valores nunca cambian. Además, cualquier constante es implícitamente `static` (por lo que es un error de sintaxis declarar una constante con la palabra clave `static` en forma explícita). Al hacer estas constantes `static` se permite acceder a ellas a través del

Método	Descripción	Ejemplo
<code>Abs(x)</code>	valor absoluto de x	<code>Abs(23.7)</code> es 23.7 <code>Abs(0.0)</code> es 0.0 <code>Abs(-23.7)</code> es 23.7
<code>Ceiling(x)</code>	redondea x al entero más pequeño no menor de x	<code>Ceiling(9.2)</code> es 10.0 <code>Ceiling(-9.8)</code> es -9.0
<code>Cos(x)</code>	coseno trigonométrico de x (x está en radianes)	<code>Cos(0.0)</code> es 1.0
<code>Exp(x)</code>	método exponencial e^x	<code>Exp(1.0)</code> es 2.71828 <code>Exp(2.0)</code> es 7.38906
<code>Floor(x)</code>	redondea x al entero más grande no mayor de x	<code>Floor(9.2)</code> es 9.0 <code>Floor(-9.8)</code> es -10.0
<code>Log(x)</code>	logaritmo natural de x (base e)	<code>Log(Math.E)</code> es 1.0 <code>Log(Math.E * Math.E)</code> es 2.0
<code>Max(x, y)</code>	el valor mayor de x y y	<code>Max(2.3, 12.7)</code> es 12.7 <code>Max(-2.3, -12.7)</code> es -2.3
<code>Min(x, y)</code>	el valor menor de x y y	<code>Min(2.3, 12.7)</code> es 2.3 <code>Min(-2.3, -12.7)</code> es -12.7
<code>Pow(x, y)</code>	x elevada a la potencia de y (es decir, x^y)	<code>Pow(2.0, 7.0)</code> es 128.0 <code>Pow(9.0, 0.5)</code> es 3.0
<code>Sin(x)</code>	seno trigonométrico de x (x está en radianes)	<code>Sin(0.0)</code> es 0.0
<code>Sqrt(x)</code>	raíz cuadrada de x	<code>Sqrt(900.0)</code> es 30.0
<code>Tan(x)</code>	tangente trigonométrica de x (x está en radianes)	<code>Tan(0.0)</code> es 0.0

Figura 7.1 | Métodos de la clase `Math`.

nombre de clase `Math` y del operador punto (`.`), justo igual que los métodos de la clase `Math`. En la sección 4.5 vimos que cuando cada objeto de una clase mantiene su propia copia de un atributo, la variable que representa a ese atributo también se conoce como variable de instancia; cada objeto (instancia) de la clase tiene una instancia separada de esa variable en memoria. Hay variables para las cuales cada objeto de una clase *no* tiene una instancia separada de esa variable. Éste es el caso con las variables `static`. Cuando se crean objetos de una clase que contiene variables `static`, todos los objetos de esa clase comparten una copia de dichas variables. En conjunto, las variables `static` y las variables de instancia representan los *campos* de una clase. En la sección 9.10 aprenderá más acerca de las variables `static`.

¿Por qué el método `Main` se declara como `static`?

¿Por qué `Main` debe declararse `static`? Durante el inicio de una aplicación, cuando no se han creado objetos de la clase, es necesario hacer una llamada al método `Main` para iniciar la ejecución del programa. A este método se le conoce también como el *punto de entrada* de la aplicación. Al declarar a `Main` como `static` se permite que el entorno de ejecución invoque a `Main` sin necesidad de crear una instancia de la clase. Por lo general, el método `Main` se declara con el siguiente encabezado:

```
public static void Main( string args[] )
```

Cuando usted ejecuta su aplicación desde la línea de comandos, escribe el nombre de la aplicación, por ejemplo:

```
NombreAplicación argumento1 argumento2 ...
```

En el comando anterior, *argumento1* y *argumento2* son los *argumentos de línea de comandos* para la aplicación, los cuales especifican una lista de objetos `string` (separados por espacios) que pasará el entorno de ejecución al método `Main` de su aplicación. Dichos argumentos podrían utilizarse para especificar opciones (por ejemplo, un nombre de archivo) para ejecutar la aplicación. Como aprenderá en el capítulo 8, Arreglos, su aplicación puede acceder a esos argumentos de línea de comandos y utilizarlos para personalizar la aplicación.

Comentarios adicionales acerca del método `Main`

El encabezado de un método `Main` no necesita aparecer exactamente como lo hemos mostrado. Las aplicaciones que no reciben argumentos de línea de comandos pueden omitir el parámetro `string[] args`. La palabra clave `public` también puede omitirse. Además, puede declarar a `Main` con el tipo de valor de retorno `int` (en vez de `void`) para permitir que `Main` devuelva un código de error con la instrucción `return`. Un método `Main` declarado con cualquiera de estos encabezados puede utilizarse como el punto de entrada de la aplicación; pero sólo se puede declarar un método `Main` en cada clase.

En los capítulos anteriores, la mayoría de las aplicaciones tenía una clase que contenía sólo el método `Main` y algunos ejemplos tenían una segunda clase que `Main` utilizaba para crear y manipular objetos. En realidad, cualquier clase puede contener un método `Main`. De hecho, cada uno de nuestros ejemplos con dos clases podría haberse implementado como una clase. Por ejemplo, en la aplicación en las figuras 6.9 y 6.10, el método `Main` (líneas 6-16 de la figura 6.10) se pudo haber tomado como estaba y colocarse en la clase `LibroCalificaciones` (figura 6.9). Los resultados de la aplicación serían idénticos a los de la versión con dos clases. Puede colocar un método `Main` en cada clase que declare. Algunos programadores aprovechan esto para crear una pequeña aplicación de prueba en cada clase que declaran. No obstante, si declara más de un método `Main` entre las clases de su proyecto, tendrá que indicar al IDE cuál desea que sea el punto de entrada de la aplicación. Para hacer esto haga clic en el menú **Proyecto > Propiedades de [NombreDelProyecto]...** (en donde **[NombreDelProyecto]** es el nombre de su proyecto) y, desde el cuadro de lista **Objeto inicial**, seleccione la clase que contenga el método `Main` que debe ser el punto de entrada.

7.4 Declaración de métodos con múltiples parámetros

En los capítulos del 4 al 6 presentamos clases que contenían métodos simples, los cuales tenían cuando mucho un parámetro. A menudo, los métodos requieren más de una pieza de información para realizar sus tareas. Ahora veremos cómo puede escribir sus propios métodos con múltiples parámetros.

La aplicación en las figuras 7.2 y 7.3 utiliza un método, definido por el usuario, llamado `Maximo` para determinar y devolver el mayor de tres valores `double` introducidos por el usuario. Cuando la aplicación empieza a ejecutarse, el método `Main` de la clase `PruebaBuscadorMaximo` (líneas 6-10 de la figura 7.3) crea un objeto de la clase `BuscadorMaximo` (línea 8) y llama al método `DeterminarMaximo` del objeto (línea 9) para producir el resultado de la aplicación. En la clase `BuscadorMaximo` (figura 7.2), las líneas 11-15 del método `DeterminarMaximo` piden al usuario que introduzca tres valores `double` y los lea del usuario (mediante el teclado). La línea 18 llama al método `Maximo` (declarado en las líneas 25-38) para determinar el mayor de los tres valores `double` que se pasan como argumentos al método. Cuando el método `Maximo` devuelve el resultado a la línea 18, la aplicación asigna el valor de retorno de `Maximo` a la variable local `resultado`. Después, la línea 21 imprime en pantalla el valor máximo. Al final de esta sección hablaremos sobre el uso del operador `+` en la línea 21.

Considere la declaración del método `Maximo` (líneas 25-38). La línea 25 indica que el método devuelve un valor `double`, que el nombre del método es `Maximo` y que el método requiere tres parámetros `double` (`x`, `y` y `z`) para realizar su tarea. Cuando un método tiene más de un parámetro, éstos se especifican como una lista separada por comas. Cuando se hace la llamada a `Maximo` en la línea 18 de la figura 7.2, el parámetro `x` se inicializa con el valor del argumento `numero1`, el parámetro `y` se inicializa con el valor del argumento `numero2` y el parámetro `z`

```

1  // Fig. 7.2: BuscadorMaximo.cs
2  // Método Maximo definido por el usuario.
3  using System;
4
5  public class BuscadorMaximo
6  {
7      // obtiene tres valores de punto flotante y determina el valor máximo
8      public void DeterminarMaximo()
9      {
10         // pide y recibe como entrada tres valores de punto flotante
11         Console.WriteLine( "Escriba tres valores de punto flotante,\n"
12             + " oprima 'Intro' después de cada uno: " );
13         double numero1 = Convert.ToDouble( Console.ReadLine() );
14         double numero2 = Convert.ToDouble( Console.ReadLine() );
15         double numero3 = Convert.ToDouble( Console.ReadLine() );
16
17         // determina el valor máximo
18         double resultado = Maximo( numero1, numero2, numero3 );
19
20         // muestra el valor máximo
21         Console.WriteLine( "Maximo es: " + resultado );
22     } // fin del método DeterminarMaximo
23
24     // devuelve el máximo de sus tres parámetros double
25     public double Maximo( double x, double y, double z )
26     {
27         double valorMaximo = x; // asume que x es el mayor para empezar
28
29         // determina si y es mayor que valorMaximo
30         if ( y > valorMaximo )
31             valorMaximo = y;
32
33         // determina si z es mayor que valorMaximo
34         if ( z > valorMaximo )
35             valorMaximo = z;
36
37         return valorMaximo;
38     } // fin del método Maximo
39 } // fin de la clase BuscadorMaximo

```

Figura 7.2 | Método `Maximo` definido por el usuario.


```

1 // Fig. 7.3: PruebaBuscadorMaximo.cs
2 // Aplicación para evaluar la clase BuscadorMaximo.
3 public class PruebaBuscadorMaximo
4 {
5     // punto inicial de la aplicación
6     public static void Main( string[] args )
7     {
8         BuscadorMaximo buscadorMaximo = new BuscadorMaximo();
9         buscadorMaximo.DeterminarMaximo();
10    } // fin de Main
11 } // fin de la clase PruebaBuscadorMaximo

```

Escriba tres valores de punto flotante,
oprima 'Intro' después de cada uno:

3.33
2.22
1.11
Maximo es: 3.33

Escriba tres valores de punto flotante,
oprima 'Intro' después de cada uno:

2.22
3.33
1.11
Maximo es: 3.33

Escriba tres valores de punto flotante,
oprima 'Intro' después de cada uno:

1.11
2.22
867.5309
Maximo es: 867.5309

Figura 7.3 | Aplicación para probar la clase BuscadorMaximo.

se inicializa con el valor del argumento `numero3`. Debe haber un argumento en la llamada al método para cada parámetro (algunas veces conocido como *parámetro formal*) en la declaración del método. Además, cada argumento debe ser consistente con el tipo del parámetro correspondiente. Por ejemplo, un parámetro de tipo `double` puede recibir valores como 7.35 (un `double`), 22 (un `int`) o -0.03456 (un `double`), pero no objetos `string` como "ho1a". En la sección 7.7 veremos los tipos de argumentos que pueden proporcionarse en la llamada a un método para cada parámetro de un tipo simple.

Para determinar el valor máximo, comenzamos con la suposición de que el parámetro `x` contiene el valor más grande, por lo que la línea 27 (figura 7.2) declara la variable local `valorMaximo` y la inicializa con el valor del parámetro `x`. Desde luego que es posible que el parámetro `y` o `z` contenga el valor más grande, por lo que debemos comparar cada uno de estos valores con `valorMaximo`. La instrucción `if` en las líneas 30-31 determina si `y` es mayor que `valorMaximo`. De ser así, la línea 31 asigna `y` a `valorMaximo`. La instrucción `if` en las líneas 34-35 determina si `z` es mayor que `valorMaximo`. De ser así, la línea 35 asigna `z` a `valorMaximo`. En este punto, el mayor de los tres valores reside en `valorMaximo`, por lo que la línea 37 devuelve ese valor a la línea 18. Cuando el control del programa regresa al punto en la aplicación en donde se llamó al método `Maximo`, los parámetros `x`, `y` y `z` de `Maximo` ya no están accesibles en la memoria. Observe que los métodos pueden devolver a lo máximo un valor; el valor devuelto puede ser una referencia a un objeto que contenga muchos valores.

Observe que `resultado` es una variable local en el método `DeterminarMaximo`, ya que se declara en el bloque que representa el cuerpo del método. Las variables deben declararse como campos de una clase (es decir, ya sea como variables de instancia o como variables `static` de la clase) sólo si se requiere su uso en más de un método de la clase, o si la aplicación debe almacenar sus valores entre las llamadas a los métodos de la clase.



Error común de programación 7.1

Declarar parámetros del mismo tipo para un método, como `float x, y` en vez de `float x, float y` es un error de sintaxis; se requiere un tipo para cada parámetro en la lista de parámetros.



Observación de ingeniería de software 7.4

Un método que tiene muchos parámetros puede estar realizando demasiadas tareas. Considere dividir el método en métodos más pequeños que realizan las tareas separadas. Como lineamiento, trate de ajustar el encabezado del método en una línea, si es posible.

Implementación del método `Maximo` mediante la reutilización del método `Math.Max`

En la figura 7.1 vimos que la clase `Math` tiene un método `Max`, que puede determinar el mayor de dos valores. Todo el cuerpo de nuestro método para encontrar el valor máximo también podría implementarse mediante llamadas anidadas a `Math.Max`, como se muestra a continuación:

```
return Math.Max( x, Math.Max( y, z ) );
```

La llamada de más a la izquierda a `Math.Max` especifica los argumentos `x` y `Math.Max( y, z )`. Antes de poder llamar a cualquier método, todos sus argumentos deben evaluarse para determinar sus valores. Si un argumento es una llamada a un método, es necesario realizar esta llamada para determinar su valor de retorno. Por lo tanto, en la instrucción anterior `Math.Max( y, z )` se evalúa primero para determinar el máximo entre `y` y `z`. Después, el resultado se pasa como el segundo argumento para la otra llamada a `Math.Max`, que devuelve el mayor de sus dos argumentos. El uso de `Math.Max` de esta forma es un buen ejemplo de la reutilización de software; buscamos el mayor de los tres valores reutilizando `Math.Max`, el cual busca el mayor de dos valores. Observe lo conciso de este código, en comparación con las líneas 27-37 de la figura 7.2.

Ensamblado de cadenas mediante la concatenación

C# permite crear objetos `string` mediante el ensamblado de objetos `string` más pequeños para formar objetos `string` más grandes, mediante el uso del operador `+` (o del operador de asignación compuesto `+=`). A esto se le conoce como *concatenación de objetos `string`*. Cuando ambos operandos del operador `+` son objetos `string`, el operador `+` crea un nuevo objeto `string` en el cual se coloca una copia de los caracteres del operando derecho al final de una copia de los caracteres en el operando izquierdo. Por ejemplo, la expresión `"hola" + "a todos"` crea el objeto `string` `"hola a todos"` sin perturbar los objetos `string` originales.

En la línea 21 de la figura 7.2, la expresión `"Maximo es: " + resultado` utiliza el operador `+` con operandos de tipo `string` y `double`. Cada valor de un tipo simple en C# tiene una representación `string`. Cuando uno de los operandos del operador `+` es un objeto `string`, el otro se convierte implícitamente en un objeto `string` y después se concatenan los dos. En la línea 21, el valor `double` se convierte implícitamente en su representación `string` y se coloca al final del objeto `string` `"Maximo es: "`. Si hay ceros a la derecha en un valor `double`, éstos se descartan cuando el número se convierte en objeto `string`. Por ende, el número 9.3500 se representa como 9.35 en el objeto `string` resultante.

Para los valores de tipos simples que se utilizan en la concatenación de objetos `string`, los valores se convierten en objetos `string`. Si un valor `boolean` se concatena con un objeto `string`, el valor `bool` se convierte en el objeto `string` `"True"` o `"False"` (observe que cada uno empieza con mayúscula). Todos los objetos tienen un método `ToString` que devuelve una representación `string` del objeto. Cuando se concatena un objeto con un `string`, se hace una llamada implícita al método `ToString` de ese objeto para obtener la representación `string`.

La línea 21 de la figura 7.2 también podría haberse escrito utilizando el formato de `string`, como se muestra a continuación:

```
Console.WriteLine( "Maximo es: {0}", resultado );
```

Al igual que con la concatenación de objetos `string`, al utilizar un elemento de formato para sustituir un objeto en un `string` se hace una llamada implícita al método `ToString` del objeto para obtener su representación `string`. En el capítulo 8, Arreglos, aprenderá más acerca del método `ToString`.

Cuando se escribe una literal `string` extensa en el código fuente de una aplicación, se puede dividir en varios objetos `string` más pequeños para colocarlos en varias líneas y mejorar la legibilidad. Los objetos `string` pueden reensamblarse mediante el uso de la concatenación o el formato de cadenas. En el capítulo 16, Cadenas, caracteres y expresiones regulares, hablaremos sobre los detalles de los objetos `string`.



Error común de programación 7.2

Es un error de sintaxis dividir una literal `string` en varias líneas en una aplicación. Si una literal `string` no cabe en una línea, divídala en objetos `string` más pequeños y utilice la concatenación para formar la literal `string` deseada.



Error común de programación 7.3

Confundir el operador `+` que se utiliza para la concatenación de cadenas con el operador `+` que se utiliza para la suma puede producir resultados extraños. El operador `+` es asociativo a la izquierda. Por ejemplo, si la variable entera y tiene el valor 5, la expresión `"y + 2 = " + y + 2` produce la cadena `"y + 2 = 52"`, no `"y + 2 = 7"`, ya que el primer valor de y (5) se concatena con la cadena `"y + 2 ="` y después el valor 2 se concatena con la nueva cadena `"y + 2 = 5"` más larga. La expresión `"y + 2 =" + (y + 2)` produce el resultado deseado `"y + 2 = 7"`.

7.5 Notas acerca de cómo declarar y utilizar los métodos

Hemos visto tres formas de llamar a un método:

1. Utilizando el nombre de un método por sí mismo para llamar a un método de la misma clase, como `Maximo( numero1, numero2, numero3 )` en la línea 18 de la figura 7.2.
2. Utilizando una variable que contiene una referencia a un objeto, seguida del operador punto (`.`) y del nombre del método para llamar a un método no `static` del objeto al que se hace referencia, como la llamada al método en la línea 9 de la figura 7.3, `buscadorMaximo.DeterminarMaximo()`, con lo cual se llama a un método de la clase `BuscadorMaximo` desde el método `Main` de `PruebaBuscadorMaximo`.
3. Utilizando el nombre de la clase y el operador punto (`.`) para llamar a un método `static` de una clase, como `Math.Sqrt( 900.0 )` en la sección 7.3.

Observe que un método `static` sólo puede llamar directamente a otros métodos `static` de la misma clase (es decir, usando el nombre del método por sí mismo) y sólo puede manipular directamente variables `static` en la misma clase. Para acceder a los miembros no `static` de la clase, un método `static` debe hacer referencia a un objeto de esa clase. Recuerde que los métodos `static` se relacionan con una clase como un todo, mientras que los métodos no `static` se asocian con una instancia específica (objeto) de la clase y pueden manipular las variables de instancia de ese objeto. Pueden existir muchos objetos de una clase al mismo tiempo, cada uno con sus propias copias de las variables de instancia. Suponga que un método `static` invoca a un método no `static` en forma directa. ¿Cómo sabría el método qué variables de instancia manipular de cuál objeto? ¿Qué ocurriría si no existieran objetos de la clase en el momento en el que se invocara el método no `static`? Por consecuencia, C# no permite que un método `static` acceda directamente a los miembros no `static` de la misma clase.

Existen tres formas de regresar el control a la instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método, o cuando se ejecuta la instrucción

`return;`

Si el método devuelve un resultado, la instrucción

`return expresión;`

evalúa la *expresión* y después devuelve el resultado (y el control) al método que hizo la llamada.



Error común de programación 7.4

Declarar un método fuera del cuerpo de la declaración de una clase, o dentro del cuerpo de otro método es un error de sintaxis.



Error común de programación 7.5

Omitir el tipo de valor de retorno en la declaración de un método es un error de sintaxis.



Error común de programación 7.6

Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de la declaración de un método es un error de sintaxis.



Error común de programación 7.7

Redeclarar el parámetro de un método como una variable local en el cuerpo de ese método es un error de compilación.



Error común de programación 7.8

Olvidar devolver el valor de un método que debe devolver un valor es un error de compilación. Si se especifica un tipo de valor de retorno distinto de void, el método debe contener una instrucción return que devuelva un valor consistente con el tipo de valor de retorno del método. Devolver un valor de un método cuyo tipo de valor de retorno se haya declarado como void es un error de compilación.

7.6 La pila de llamadas a los métodos y los registros de activación

Para comprender la forma en que C# realiza las llamadas a los métodos, primero necesitamos considerar una estructura de datos (es decir, una colección de elementos de datos relacionados) conocida como *pila*. Puede considerarse una pila como una analogía de una pila de platos. Cuando se coloca un plato en la pila, por lo general se coloca en la parte superior (lo que se conoce como *meter* el plato en la pila). De manera similar, cuando se extrae un plato de la pila, siempre se extrae de la parte superior (lo que se conoce como *sacar* el plato de la pila). Las pilas se denominan *estructuras de datos “último en entrar, primero en salir” (LIFO)*; el último elemento que se mete (inserta) en la pila es el primero que se saca (se extrae) de ella.

Cuando una aplicación llama a un método, éste debe saber cómo regresar al que lo llamó, por lo que la dirección de retorno del método que hizo la llamada se mete en la *pila de ejecución del programa* (también conocida como *pila de llamadas a los métodos*). Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila, en el orden “último en entrar, primero en salir”, para que cada método pueda regresar al que lo llamó.

La pila de ejecución del programa también contiene la memoria para las variables locales que se utilizan en cada invocación de un método durante la ejecución de una aplicación. Estos datos, que se almacenan como una porción de la pila de ejecución del programa, se conocen como el *registro de activación* o *marco de pila* de la llamada a un método. Cuando se hace la llamada a un método, el registro de activación para la llamada a ese método se mete en la pila de ejecución del programa. Cuando el método regresa al que lo llamó, el registro de activación para esa llamada al método se saca de la pila y esas variables locales ya no son conocidas para la aplicación. Si una variable local que almacena una referencia a un objeto es la única variable en la aplicación con una referencia a ese objeto, cuando se saca de la pila el registro de activación que contiene a esa variable local, la aplicación ya no puede acceder a ese objeto, el cual se eliminará de memoria en algún momento dado durante la “recolección de basura”. En la sección 9.9 hablaremos sobre la recolección de basura.

Desde luego que la cantidad de memoria en una computadora es finita, por lo que sólo puede utilizarse cierta cantidad de memoria para almacenar los registros de activación en la pila de ejecución del programa. Si ocurren más llamadas a métodos de las que se puedan almacenar los registros de activación en la pila de ejecución del programa, se produce un error conocido como *desbordamiento de pila*.

7.7 Promoción y conversión de argumentos

Otra característica importante de las llamadas a los métodos es la *promoción de argumentos*: convertir en forma implícita el valor de un argumento al tipo que el método espera recibir en su correspondiente parámetro. Por ejemplo, una aplicación puede llamar al método Sqrt de Math con un argumento entero, aun y cuando el método espera recibir un argumento double (pero no viceversa, como pronto veremos). La instrucción

```
Console.WriteLine( Math.Sqrt( 4 ) );
```

evalúa `Math.Sqrt( 4 )` correctamente e imprime el valor 2.0. La lista de parámetros de la declaración del método hace que C# convierta el valor `int` 4 en el valor `double` 4.0 antes de pasar ese valor a `Sqrt`. Tratar de realizar estas conversiones puede ocasionar errores de compilación, si no se satisfacen las *reglas de promoción* de C#. Las reglas de promoción especifican qué conversiones son permitidas; esto es, qué conversiones pueden realizarse sin perder datos. En el ejemplo anterior de `Sqrt`, un `int` se convierte en `double` sin modificar su valor. No obstante, la conversión de un `double` a un `int` trunca la parte fraccional del valor `double`; por lo tanto, se pierde parte del valor. Además, las variables `double` pueden almacenar valores mucho mayores (y mucho menores) que las variables `int`, por lo que asignar un `double` a un `int` puede provocar una pérdida de información cuando el valor `double` no cabe en el valor `int`. La conversión de tipos de enteros largos a tipos de enteros pequeños (por ejemplo, de `long` a `int`) puede también producir valores modificados.

Las reglas de promoción se aplican a las expresiones que contienen valores de dos o más tipos simples, y a los valores de tipos simples que se pasan como argumentos para los métodos. Cada valor se promueve al tipo apropiado en la expresión. (En realidad, la expresión utiliza una copia temporal de cada valor; los tipos de los valores originales permanecen sin cambios.) La figura 7.4 lista los tipos simples en orden alfabético, y los tipos a los cuales se puede promover cada uno de ellos. Observe que los valores de todos los tipos simples también pueden convertirse implícitamente al tipo `object`. En el capítulo 24, Estructuras de datos, demostraremos dichas conversiones implícitas.

De manera predeterminada, C# no nos permite convertir implícitamente valores entre los tipos simples, si el tipo de destino no puede representar el valor del tipo original (por ejemplo, el valor `int` 2000000 no puede representarse como un `short`, y cualquier número de punto flotante con dígitos después de su punto decimal no pueden representarse en un tipo entero como `long`, `int` o `short`). Por lo tanto, para evitar un error de compilación en casos en los que la información puede perderse debido a una conversión implícita entre los tipos simples, el compilador requiere que utilicemos un operador de conversión (el cual presentamos en la sección 5.7) para forzar explícitamente la conversión. Eso nos permite “tomar el control” del compilador. En esencia decimos, “Sé que esta conversión podría ocasionar pérdida de información, pero para mis fines aquí, eso está bien”. Suponga que crea un método `Square` que calcula el cuadrado de un entero y por ende requiere un argumento `int`. Para llamar a `Square` con un argumento `double` llamado `valorDouble`, escribiría la llamada al método como `Square( (int) valorDouble )`. La llamada a este método convierte explícitamente el valor de `valorDouble` a un entero, para usarlo en el método `Square`. Por ende, si el valor de `valorDouble` es 4.5, el método recibe el valor 4 y devuelve 16, no 20.25 (lo que, por desgracia, resulta en la pérdida de información).

Tipo	Tipo de conversiones
<i>bool</i>	no hay conversiones implícitas posibles a otros tipos simples
<i>byte</i>	<i>ushort, short, uint, int, ulong, long, decimal, float o double</i>
<i>char</i>	<i>ushort, int, uint, long, ulong, decimal, float o double</i>
<i>decimal</i>	no hay conversiones implícitas posibles a otros tipos simples
<i>double</i>	no hay conversiones implícitas posibles a otros tipos simples
<i>float</i>	<i>double</i>
<i>int</i>	<i>long, decimal, float o double</i>
<i>long</i>	<i>decimal, float o double</i>
<i>sbyte</i>	<i>short, int, long, decimal, float o double</i>
<i>short</i>	<i>int, long, decimal, float o double</i>
<i>uint</i>	<i>ulong, long, decimal, float o double</i>
<i>ulong</i>	<i>decimal, float o double</i>
<i>ushort</i>	<i>uint, int, ulong, long, decimal, floatx o double</i>

Figura 7.4 | Conversiones implícitas entre tipos simples.



Error común de programación 7.9

Convertir un valor de tipo simple a un valor de otro tipo simple puede modificar ese valor si no se permite la promoción. Por ejemplo, convertir un valor de punto flotante a un valor integral puede introducir errores de truncamiento (pérdida de la parte fraccional) en el resultado.

7.8 La Biblioteca de clases del .NET Framework

Muchas clases predefinidas se agrupan en categorías de clases relacionadas, llamadas espacios de nombres. En conjunto, estos espacios de nombres se conocen como la Biblioteca de clases del .NET Framework, o FCL.

A lo largo de este libro, las directivas `using` nos permiten utilizar las clases de biblioteca de la FCL sin necesidad de especificar sus nombres completamente calificados. Por ejemplo, la declaración

```
using System;
```

permite a una aplicación utilizar los nombres de las clases del espacio de nombres `System` sin tener que calificar completamente sus nombres. Esto nos permite utilizar el *nombre de clase descalificado* `Console`, en vez del nombre de clase completamente calificado `System.Console`, en el código. Una gran ventaja de C# es el extenso número de clases en los espacios de nombres de la FCL. En la figura 7.5 se describen algunos espacios de nombres clave de la FCL, que representan sólo una pequeña porción de las clases reutilizables en la FCL. En su proceso de aprendizaje de C#, le recomendamos invertir tiempo en explorar los espacios de nombres y las clases en la documentación .NET (*en inglés*: msdn2.microsoft.com/en-us/library/ms229335; *en español*: msdn.microsoft.com/library/spa/default.aspx?url=/library/SPA/cpref/html/cpref_start.asp).

El conjunto de espacios de nombres disponibles en la FCL es bastante extenso. Además de los espacios de nombres que se sintetizan en la figura 7.5, la FCL contiene espacios de nombres para gráficos complejos, interfaces gráficas de usuario avanzadas, impresión, redes avanzadas, seguridad, procesamiento de bases de datos, multimedia, accesibilidad (para personas con discapacidad) y muchas otras capacidades. El URL anterior de la documentación .NET proporciona las generalidades acerca de los espacios de nombres de la Biblioteca de clases del .NET Framework.

Espacio de nombres	Descripción
<code>System.Windows.Forms</code>	Contiene las clases requeridas para crear y manipular GUIs. (En el capítulo 13, Conceptos de interfaz gráfica de usuario, parte 1, y en el capítulo 14, Conceptos de interfaz gráfica de usuario, parte 2, hablaremos sobre varias de las clases en este espacio de nombres.)
<code>System.IO</code>	Contiene clases que permiten las operaciones de entrada/salida de datos en los programas. (En el capítulo 18, Archivos y flujos, aprenderá más acerca de este espacio de nombres.)
<code>System.Data</code>	Contiene clases que permiten a los programas acceder a las bases de datos (es decir, colecciones organizadas de datos) y manipularlas. (En el capítulo 20, Bases de datos, SQL y ADO.NET, aprenderá más acerca de este espacio de nombres.)
<code>System.Web</code>	Contiene clases que se utilizan para crear y mantener aplicaciones Web, a las cuales se accede a través de Internet. (En el capítulo 21, ASP.NET 2.0, formularios Web Forms y controles Web, aprenderá más acerca de este espacio de nombres.)
<code>System.Xml</code>	Contiene clases para crear y manipular datos de XML. Los datos pueden leerse o escribirse en archivos de XML. (En el capítulo 19, Lenguaje de marcado extensible (XML), aprenderá más acerca de este espacio de nombres.)

Figura 7.5 | Espacios de nombres de la FCL (un subconjunto). (Parte 1 de 2).

Espacio de nombres	Descripción
System.Collections System.Collections.Generic	Contiene clases que definen estructuras de datos para mantener colecciones de datos. (En el capítulo 26, Colecciones, aprenderá más acerca de este espacio de nombres.)
System.Net	Contiene clases que permiten que los programas se comuniquen a través de redes de computadoras como Internet. (En el capítulo 23, Redes: Sockets basados en flujos y datagramas, aprenderá más acerca de este espacio de nombres.)
System.Text	Contiene clases e interfaces que permiten a los programas manipular caracteres y cadenas. (En el capítulo 16, Cadenas, caracteres y expresiones regulares, aprenderá más acerca de este espacio de nombres.)
System.Threading	Contiene clases que permiten a los programas realizar varias tareas al mismo tiempo. (En el capítulo 15, Subprocesamiento múltiple, aprenderá más acerca de este espacio de nombres.)
System.Drawing	Contiene clases que permiten a los programas realizar el procesamiento básico de gráficos, tal como mostrar figuras y arcos. (En el capítulo 17, Gráficos y multimedia, aprenderá más acerca de este espacio de nombres.)

Figura 7.5 | Espacios de nombres de la FCL (un subconjunto). (Parte 2 de 2).

En la Referencia de la Biblioteca de clases del .NET Framework podrá encontrar información adicional acerca de los métodos de las clases predefinidas en C#. Cuando visite este sitio, verá un listado alfabético de todos los espacios de nombres en la FCL. Localice el espacio de nombres y haga clic en su vínculo para poder ver un listado alfabético de todas sus clases, con una breve descripción de cada una de ellas. Haga clic en el vínculo de una clase para ver una descripción más completa de la misma. Haga clic en el vínculo **Métodos (Methods)** de la columna izquierda para ver un listado de los métodos de la clase.



Buena práctica de programación 7.2

La documentación en línea del .NET Framework es fácil de explorar y proporciona muchos detalles acerca de cada clase. A medida que conozca cada una de las clases en este libro, es conveniente que revise la clase en la documentación en línea para obtener información adicional.

7.9 Caso de estudio: generación de números aleatorios

En esta sección y en la siguiente, desarrollaremos una aplicación de juego muy bien estructurada, con varios métodos. La aplicación utiliza la mayoría de las instrucciones de control presentadas hasta ahora en el libro; además introduce varios conceptos nuevos de programación.

Hay algo en el aire de un casino que vigoriza a las personas; desde los grandes apostadores en las lujosas mesas de dados de caoba y fieltro, hasta las ranuras que expulsan monedas en los “bandidos de un solo brazo”. Es el *elemento de probabilidad*, la posibilidad de que la suerte convierta unos cuantos pesos en una montaña de riqueza. El elemento de probabilidad puede introducirse en una aplicación mediante un objeto de la clase **Random** (del espacio de nombres **System**). Los objetos de la clase **Random** pueden producir valores **byte**, **int** y **double** aleatorios. En los siguientes ejemplos utilizaremos objetos de la clase **Random** para producir números aleatorios.

Se puede crear un nuevo objeto generador de números aleatorios de la siguiente manera:

```
Random numerosAleatorios = new Random();
```

Así, el objeto generador de números aleatorios puede utilizarse para generar valores **byte**, **int** y **double** aleatorios; aquí sólo veremos valores **int** aleatorios. Para obtener más información sobre la clase **Random**, visite el sitio en inglés msdn2.microsoft.com/en-us/library/ts6se2ek, y el sitio en español msdn.microsoft.com/library/spa/default.asp?url=/library/SPA/cpref/html/cpref_start.asp.

Considere la siguiente instrucción:

```
int valorAleatorio = numerosAleatorios.Next();
```

El método `Next` de la clase `Random` genera un valor `int` aleatorio en el rango de 0 a +2,147,483,646, inclusive. Si en verdad el método `Next` produce valores al azar, entonces todos los valores en ese rango deberían tener la misma oportunidad (o probabilidad) de ser elegidos cada vez que se hace una llamada al método `Next`. En realidad, los valores devueltos por `Next` son *números pseudoaleatorios* (una secuencia de valores producidos por un cálculo matemático complejo. El cálculo utiliza la hora actual del día (que, por supuesto, cambia en forma constante) para *sembrar* el generador de números aleatorios, de tal forma que cada ejecución de una aplicación produzca una secuencia distinta de valores aleatorios.

Con frecuencia, el rango de valores que produce directamente el método `Next` difiere del rango de valores requeridos en una aplicación específica de C#. Por ejemplo, una aplicación que simule el lanzamiento de una moneda podría requerir sólo 0 para “águila” y 1 para “sol”. Una aplicación que simule el tiro de un dado de seis lados podría requerir enteros aleatorios en el rango de 1 a 6. Un videojuego que adivine en forma aleatoria la siguiente nave espacial (de cuatro posibilidades distintas) que volará a lo largo del horizonte podría requerir enteros aleatorios en el rango de 1 a 4. Para casos como éstos, la clase `Random` cuenta con otras versiones del método `Next`. Una de ellas recibe un argumento `int` y devuelve un valor de 0 hasta, pero sin incluir, el valor del argumento. Por ejemplo, podría utilizar la instrucción

```
int valorAleatorio = numerosAleatorios.Next( 6 );
```

que devuelve 0, 1, 2, 3, 4 o 5. El argumento 6 (llamado el *factor de escala*) representa el número de valores únicos que debe producir `Next` (en este caso, seis: 0, 1, 2, 3, 4 y 5). A esta manipulación se le conoce como *escalar* el rango de valores producidos por el método `Next` de `Random`.

Suponga que deseamos simular un dado de seis caras que tiene los números del 1 al 6 en sus caras, no del 0 al 5. No basta con sólo escalar el rango de valores, sino que tenemos que *desplazar* el rango de números producidos. Para ello podríamos agregar un *valor de desplazamiento* (en este caso, 1) al resultado del método `Next`, como en

```
cara = 1 + numerosAleatorios.Next( 6 );
```

El valor de desplazamiento (1) especifica el primer valor en el conjunto deseado de enteros aleatorios. La instrucción anterior asigna a `cara` un entero aleatorio en el rango del 1 al 6.

La tercera alternativa del método `Next` proporciona una manera más intuitiva de expresar tanto el desplazamiento como la acción de escalar. Este método recibe dos argumentos `int` y devuelve un valor desde el valor del primer argumento hasta, pero sin incluir, el valor del segundo argumento. Podríamos utilizar este método para escribir una instrucción equivalente a nuestra instrucción anterior, como en

```
cara = numerosAleatorios.Next( 1, 7 );
```

Tirar un dado de seis caras

Para demostrar el uso de los números aleatorios, desarrollaremos una aplicación que simula 20 tiros de un dado de seis caras y muestra el valor de cada tiro. La figura 7.6 muestra dos ejecuciones de ejemplo, que confirman que los resultados del cálculo anterior son enteros en el rango del 1 al 6 y que cada ejecución de la aplicación puede producir una secuencia distinta de números aleatorios. La directiva `using` en la línea 3 nos permite utilizar la clase `Random` sin necesidad de calificar completamente su nombre. La línea 9 crea el objeto `numerosAleatorios` de la clase `Random` para producir valores aleatorios. La línea 16 se ejecuta 20 veces en un ciclo para tirar el dado. La instrucción `if` (líneas 21-22) en el ciclo inicia una nueva línea de salida después de cada cinco números, para poder presentar los resultados en varias líneas.

Tirar un dado de seis caras 6000 veces

Para mostrar que los números producidos por `Next` ocurren con una probabilidad aproximadamente igual, simularemos 6000 tiros de un dado (figura 7.7). Cada entero del 1 al 6 debe aparecer aproximadamente 1000 veces.

Como muestran los dos resultados de ejemplo, los valores producidos por el método `Next` permiten a la aplicación simular en forma realista el tiro de un dado de seis caras. La aplicación utiliza instrucciones de control

anidadas (el switch está anidado dentro del for) para determinar el número de veces que se tiró cada una de las caras del dado. La instrucción for (líneas 21-47) itera 6000 veces. Durante cada iteración, la línea 23 produce un valor aleatorio del 1 al 6. Este valor de cara se utiliza después como la expresión del switch (línea 26) en la

```

1  // Fig. 7.6: EnterosAleatorios.cs
2  // Desplazar y escalar enteros aleatorios.
3  using System;
4
5  public class EnterosAleatorios
6  {
7      public static void Main( string[] args )
8      {
9          Random numerosAleatorios = new Random(); // generador de números aleatorios
10         int cara; // almacena cada entero aleatorio que se genera
11
12         // itera 20 veces
13         for ( int contador = 1; contador <= 20; contador++ )
14         {
15             // selecciona un entero aleatorio del 1 al 6
16             cara = numerosAleatorios.Next( 1, 7 );
17
18             Console.Write( "{0} ", cara ); // muestra el valor generado
19
20             // si el contador es divisible entre 5, inicia una nueva línea de salida
21             if ( contador % 5 == 0 )
22                 Console.WriteLine();
23         } // fin de for
24     } // fin de Main
25 } // fin de la clase EnterosAleatorios

```

```

1 5 5 1 3
3 1 4 3 3
3 1 1 6 4
2 4 3 6 4

```

```

6 2 1 5 4
6 2 2 1 4
4 1 4 3 4
4 4 4 3 5

```

Figura 7.6 | Desplazar y escalar enteros aleatorios.

```

1  // Fig. 7.7: TirarDado.cs
2  // Tirar un dado de seis caras 6000 veces.
3  using System;
4
5  public class TirarDado
6  {
7      public static void Main( string[] args )
8      {
9          Random numerosAleatorios = new Random(); // generador de números aleatorios
10
11         int frecuencia1 = 0; // se tira cuenta de 1
12         int frecuencia2 = 0; // se tira cuenta de 2
13         int frecuencia3 = 0; // se tira cuenta de 3

```

Figura 7.7 | Tirar un dado de seis caras 6000 veces. (Parte I de 2).


```

14     int frecuencia4 = 0; // se tira cuenta de 4
15     int frecuencia5 = 0; // se tira cuenta de 5
16     int frecuencia6 = 0; // se tira cuenta de 6
17
18     int cara; // almacena el último valor que se tiró
19
20     // sintetiza los resultados de tirar el dado 6000 veces
21     for ( int tiro = 1; tiro <= 6000; tiro++ )
22     {
23         cara = numerosAleatorios.Next( 1, 7 ); // número del 1 al 6
24
25         // determina el valor del tiro del 1 al 6 e incrementa el contador apropiado
26         switch ( cara )
27         {
28             case 1:
29                 frecuencia1++; // incrementa el contador de 1
30                 break;
31             case 2:
32                 frecuencia2++; // incrementa el contador de 2
33                 break;
34             case 3:
35                 frecuencia3++; // incrementa el contador de 3
36                 break;
37             case 4:
38                 frecuencia4++; // incrementa el contador de 4
39                 break;
40             case 5:
41                 frecuencia5++; // incrementa el contador de 5
42                 break;
43             case 6:
44                 frecuencia6++; // incrementa el contador de 6
45                 break;
46         } // fin de switch
47     } // fin de for
48
49     Console.WriteLine( "Cara\tFrecuencia" ); // encabezados de salida
50     Console.WriteLine( "1\t{0}\n2\t{1}\n3\t{2}\n4\t{3}\n5\t{4}\n6\t{5}",
51         frecuencia1, frecuencia2, frecuencia3, frecuencia4,
52         frecuencia5, frecuencia6 );
53 } // fin de Main
54 } // fin de la clase TirarDado

```

Cara	Frecuencia
1	1022
2	1007
3	971
4	946
5	1042
6	1012

Cara	Frecuencia
1	1017
2	1000
3	1015
4	1004
5	969
6	995

Figura 7.7 | Tirar un dado de seis caras 6000 veces. (Parte 2 de 2).

instrucción `switch` (líneas 26-46). Con base en el valor de `cara`, la instrucción `switch` incrementa una de las seis variables contador durante cada iteración del ciclo. (En el capítulo 8, Arreglos, mostraremos una manera elegante de sustituir toda la instrucción `switch` completa en esta aplicación con una sola instrucción.) Observe que la instrucción `switch` no tiene etiqueta `default`, ya que tenemos una etiqueta `case` para cada uno de los posibles valores del dado que puede producir la expresión en la línea 23. Ejecute la aplicación varias veces y observe los resultados. Podrá ver que cada vez que ejecuta esta aplicación, produce distintos resultados.

7.9.1 Escalar y desplazar números aleatorios

Anteriormente demostramos la instrucción

```
cara = numerosAleatorios.Next( 1, 7 );
```

que simula el tiro de un dado de seis caras. Esta instrucción siempre asigna a la variable `cara` un entero en el rango $1 \leq \text{cara} < 7$. La amplitud de este rango (es decir, el número de enteros consecutivos en el rango) es 6, y el número inicial en el rango es 1. Si hacemos referencia a la instrucción anterior, podemos ver que la amplitud del rango se determina en base a la diferencia entre los dos enteros que se pasan al método `Next` de `Random` y que el número inicial del rango es el valor del primer argumento. Podemos generalizar este resultado de la siguiente manera:

```
numero = numerosAleatorios.Next( valorDesplazamiento, valorDesplazamiento + factorEscala );
```

en donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos y *factorEscala* especifica cuántos números hay en el rango.

También es posible elegir enteros al azar, a partir de conjuntos de valores distintos a los rangos de enteros consecutivos. Para este fin, es más sencillo utilizar la versión del método `Next` que sólo recibe un argumento. Por ejemplo, para obtener un valor aleatorio de la secuencia 2, 5, 8, 11 y 14, podríamos utilizar la siguiente instrucción:

```
numero = 2 + 3 * numerosAleatorios.Next( 5 );
```

En este caso, `numerosAleatorios.Next( 5 )` produce valores en el rango de 0 a 4. Cada valor producido se multiplica por 3 para producir un número en la secuencia 0, 3, 6, 9 y 12. Después sumamos 2 a ese valor para desplazar el rango de valores y obtener un valor de la secuencia 2, 5, 8, 11 y 14. Podemos generalizar este resultado así:

```
numero = valorDesplazamiento +  
diferenciaEntreValores * numerosAleatorios.Next( factorEscala );
```

en donde *valorDesplazamiento* especifica el primer número en el rango deseado de valores, *diferenciaEntreValores* representa la diferencia entre números consecutivos en la secuencia y *factorEscala* especifica cuántos números hay en el rango.

7.9.2 Repetitividad de números aleatorios para prueba y depuración

Como mencionamos antes en la sección 7.9, los métodos de la clase `Random` en realidad generan números pseudoaleatorios con base en cálculos matemáticos complejos. Si se llama repetidas veces cualquiera de los métodos de `Random`, se produce una secuencia de números que parecen ser aleatorios. El cálculo que producen los números pseudoaleatorios utiliza la hora del día como *valor de semilla* para cambiar el punto inicial de la secuencia. Cada nuevo objeto `Random` se siembra a sí mismo con un valor basado en el reloj del sistema computacional al momento en que se crea el objeto, con lo cual se permite que cada ejecución de la aplicación produzca una secuencia distinta de números aleatorios.

Al depurar una aplicación, en ocasiones es útil repetir la misma secuencia exacta de números pseudoaleatorios durante cada ejecución de la aplicación. Esta repetitividad nos permite probar que la aplicación esté funcionando para una secuencia específica de números aleatorios, antes de evaluar la aplicación con distintas secuencias de números aleatorios. Cuando la repetitividad es importante, podemos crear un objeto `Random` de la siguiente manera:

```
Random numerosAleatorios = new Random( valorSemilla );
```

El argumento `valorSemilla` (de tipo `int`) siembra el cálculo del número aleatorio. Si se utiliza siempre el mismo valor para `valorSemilla`, el objeto `Random` produce la misma secuencia de números aleatorios.



Tip de prevención de errores 7.2

Mientras una aplicación esté en desarrollo, cree el objeto Random con un valor de semilla específico para producir una secuencia repetible de números aleatorios cada vez que se ejecute la aplicación. Si se produce un error lógico, corrija el error y evalúe la aplicación otra vez con el mismo valor de semilla; esto le permitirá reconstruir la misma secuencia de números aleatorios que produjeron el error. Una vez que se hayan eliminado los errores lógicos, cree el objeto Random sin utilizar un valor de semilla, para que el objeto Random genere una nueva secuencia de números aleatorios cada vez que se ejecute la aplicación.

7.10 Caso de estudio: juego de probabilidad (introducción a las enumeraciones)

Un juego de azar popular es el juego de dados conocido como “craps”, el cual se juega en casinos y callejones por todo el mundo. Las reglas del juego son simples:

Un jugador tira dos dados. Cada dado tiene seis caras, las cuales contienen uno, dos, tres, cuatro, cinco y seis puntos negros, respectivamente. Una vez que los dados dejan de moverse, se calcula la suma de los puntos negros en las dos caras superiores. Si la suma es 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado “craps”), el jugador pierde (es decir, la “casa” gana). Si la suma es 4, 5, 6, 8, 9 o 10 en el primer tiro, esta suma se convierte en el “punto” del jugador. Para ganar, el jugador debe seguir tirando los dados hasta que salga otra vez “su punto” (es decir, que tire ese mismo valor de punto). El jugador pierde si tira un 7 antes de llegar a su punto.

La aplicación en las figuras 7.8 y 7.9 simula el juego craps, utilizando varios métodos para definir la lógica del juego. En el método `Main` de la clase `PruebaCraps` (figura 7.9), la línea 7 crea un objeto de la clase `Craps` (figura 7.8) y la línea 8 llama a su método `Jugar` para iniciar el juego. El método `Jugar` (figura 7.8, líneas 24-70) llama al método `TirarDado` (figura 7.8, líneas 73-85) según sea necesario para tirar los dos dados y calcular su suma. Los cuatro resultados de ejemplo en la figura 7.9 muestran que se ganó en el primer tiro, se perdió en el primer tiro, se ganó en un tiro subsecuente y se perdió en un tiro subsecuente, en forma respectiva.

```

1  // Fig. 7.8: Craps.cs
2  // La clase Craps simula el juego de dados llamado craps.
3  using System;
4
5  public class Craps
6  {
7      // crea el generador de números aleatorios para usarlo en el método TirarDados
8      private Random numerosAleatorios = new Random();
9
10     // enumeración con constantes que representan el estado del juego
11     private enum Estado { CONTINUA, GANO, PERDIO }
12
13     // enumeración con constantes que representan tiros comunes del dado
14     private enum NombresDados
15     {
16         DOS_UNOS = 2,
17         TRES = 3,
18         SIETE = 7,
19         ONCE = 11,
20         DOCE = 12
21     }
22
23     // ejecuta un juego de craps
24     public void Jugar()
25     {
26         // estadoJuego puede contener CONTINUA, GANO o PERDIO
27         Estado estadoJuego = Estado.CONTINUA;

```

Figura 7.8 | La clase `Craps` simula el juego de dados llamado “craps”. (Parte I de 2).

```

28     int miPunto = 0; // punto si no gana ni pierde en el primer tiro
29
30     int sumaDeDados = TirarDados(); // primer tiro de los dados
31
32     // determina el estado del juego y el punto con base en el primer tiro
33     switch ( ( NombresDados ) sumaDeDados )
34     {
35         case NombresDados.SIETE: // gana con 7 en el primer tiro
36         case NombresDados.ONCE: // gana con 11 en el primer tiro
37             estadoJuego = Estado.GANO;
38             break;
39         case NombresDados.DOS_UNOS: // pierde con 2 en el primer tiro
40         case NombresDados.TRES: // pierde con 3 en el primer tiro
41         case NombresDados.DOCE: // pierde con 12 en el primer tiro
42             estadoJuego = Estado.PERDIO;
43             break;
44         default: // no ganó ni perdió, entonces hay que recordar el punto
45             estadoJuego = Estado.CONTINUA; // el juego no ha terminado
46             miPunto = sumaDeDados; // recuerda el punto
47             Console.WriteLine( "El punto es {0}", miPunto );
48             break;
49     } // fin de switch
50
51     // mientras el juego no termine
52     while ( estadoJuego == Estado.CONTINUA ) // no GANO ni PERDIO el juego
53     {
54         sumaDeDados = TirarDados(); // tira los dados otra vez
55
56         // determina el estado del juego
57         if ( sumaDeDados == miPunto ) // gana haciendo punto
58             estadoJuego = Estado.GANO;
59         else
60             // pierde si tira 7 antes del punto
61             if ( sumaDeDados == ( int ) NombresDados.SIETE )
62                 estadoJuego = Estado.PERDIO;
63     } // fin de while
64
65     // muestra mensaje de ganó o perdió
66     if ( estadoJuego == Estado.GANO )
67         Console.WriteLine( "El jugador gana" );
68     else
69         Console.WriteLine( "El jugador pierde" );
70 } // fin del método Jugar
71
72 // tira los dados, calcula la suma y muestra los resultados
73 public int TirarDados()
74 {
75     // elige valores aleatorios para los dados
76     int dado1 = numerosAleatorios.Next( 1, 7 ); // tiro del primer dado
77     int dado2 = numerosAleatorios.Next( 1, 7 ); // tiro del segundo dado
78
79     int suma = dado1 + dado2; // suma de los valores de cada dado
80
81     // muestra los resultados de este tiro
82     Console.WriteLine( "El jugador tiró {0} + {1} = {2}",
83         dado1, dado2, suma );
84     return suma; // devuelve la suma de los dados
85 } // fin del método TirarDados
86 } // fin de la clase Craps

```

Figura 7.8 | La clase Craps simula el juego de dados llamado “craps”. (Parte 2 de 2).

```

1  // Fig. 7.9: PruebaCraps.cs
2  // Aplicación para probar la clase Craps.
3  public class PruebaCraps
4  {
5      public static void Main( string[] args )
6      {
7          Craps juego = new Craps();
8          juego.Jugar(); // jugar un juego de craps
9      } // fin de Main
10 } // fin de la clase PruebaCraps

```

El jugador tiró 2 + 5 = 7
El jugador gana

El jugador tiró 2 + 1 = 3
El jugador pierde

El jugador tiró 4 + 6 = 10
El punto es 10
El jugador tiró 1 + 3 = 4
El jugador tiró 1 + 3 = 4
El jugador tiró 2 + 3 = 5
El jugador tiró 4 + 4 = 8
El jugador tiró 6 + 6 = 12
El jugador tiró 4 + 4 = 8
El jugador tiró 4 + 5 = 9
El jugador tiró 2 + 6 = 8
El jugador tiró 6 + 6 = 12
El jugador tiró 6 + 4 = 10
El jugador gana

El jugador tiró 2 + 4 = 6
El punto es 6
El jugador tiró 3 + 1 = 4
El jugador tiró 5 + 5 = 10
El jugador tiró 6 + 1 = 7
El jugador pierde

Figura 7.9 | Aplicación para probar la clase Craps.

Hablaremos sobre la declaración de la clase Craps en la figura 7.8. En las reglas del juego, el jugador debe tirar dos dados en el primer tiro y debe hacer lo mismo en todos los tiros subsiguientes. Declaramos el método `TirarDados` (líneas 73-85) para tirar el dado y calcular e imprimir su suma. El método `TirarDados` se declara una vez, pero se llama desde dos lugares (líneas 30 y 54) en el método `Play`, el cual contiene la lógica para un juego completo de craps. El método `TirarDados` no tiene argumentos, por lo tanto, su lista de parámetros está vacía. Cada vez que se llama, `TirarDados` devuelve la suma de los dados, por lo que se indica el tipo de valor de retorno en el encabezado del método (línea 73). Aunque las líneas 76 y 77 se ven iguales (excepto por el nombre de los dados), no necesariamente producen el mismo resultado. Cada una de estas instrucciones produce un valor aleatorio en el rango de 1 a 6. Observe que `numerosAleatorios` (se utiliza en las líneas 76 y 77) no se declara en el método, sino que se declara como una variable `private` de la clase y se inicializa en la línea 8. Esto nos permite crear un objeto `Random` que se reutiliza en cada llamada a `TirarDados`.

El juego es razonablemente complejo. El jugador puede ganar o perder en el primer tiro, o puede ganar o perder en cualquier tiro subsiguiente. El método `Jugar` (líneas 24-70) utiliza la variable local `estadoJuego` (línea 27) para llevar el registro del estado del juego en general, la variable local `miPunto` (línea 28) para almacenar el

“punto” si el jugador no gana o pierde en el primer tiro, y la variable local `sumaDeDados` (línea 30) para mantener la suma de los dados para el tiro más reciente. Observe que `miPunto` se inicializa con 0 para asegurar que la aplicación se compile. Si no inicializa `miPunto`, el compilador genera un error ya que `miPunto` no recibe un valor en todas las ramificaciones de la instrucción `switch`; por ende, la aplicación podría tratar de utilizar `miPunto` antes de que se le asigne definitivamente un valor. En contraste, `estadoJuego` no requiere inicialización, ya que se le asigna un valor en todas las ramificaciones de la instrucción `switch`; por lo tanto, se garantiza que se inicialice antes de usarse. No obstante y como buena práctica de programación, la inicializamos de todas formas.

Observe que la variable local `estadoJuego` se declara como de un nuevo tipo llamado `Estado`, el cual declaramos en la línea 11. El tipo `Estado` se declara como un miembro `private` de la clase `Craps`, ya que sólo se utiliza en esa clase. `Estado` es un tipo definido por el usuario, conocido como *enumeración*, el cual declara un conjunto de constantes representadas por identificadores. Para introducir una enumeración se utiliza la palabra clave **enum** y un nombre para el tipo (en este caso, `Estado`). Al igual que con una clase, las llaves (`{` y `}`) delimitan el cuerpo de una declaración de `enum`. Dentro de las llaves hay una lista de *constantes de enumeración* separada por comas. Los nombres de las constantes `enum` deben ser únicos, pero sus valores subyacentes no necesitan serlo.



Buena práctica de programación 7.3

Use sólo letras mayúsculas en los nombres de las constantes. Esto hace que resalten en una aplicación y le recuerdan que las constantes de enumeración no son variables.

A las variables de tipo `Estado` se les debe asignar sólo una de las tres constantes declaradas en la enumeración. Cuando el jugador gana el juego, la aplicación asigna a la variable local `estadoJuego` el valor `Estado.GANO` (líneas 37 y 58). Cuando el jugador pierde el juego, la aplicación asigna a la variable local `estadoJuego` el valor `Estado.PERDIO` (líneas 42 y 63). En cualquier otro caso, la aplicación asigna a la variable local `estadoJuego` el valor `Estado.CONTINUA` (línea 45) para indicar que hay que tirar los dados otra vez.



Buena práctica de programación 7.4

El uso de constantes de enumeración (como `Estado.GANO`, `Estado.PERDIO` y `Estado.CONTINUA`) en vez de valores enteros constantes (como 0, 1 y 2) aumenta la legibilidad del código y simplifica su mantenimiento.

La línea 30 en el método `Jugar` llama a `TirarDados`, el cual elige dos valores aleatorios del 1 al 6, muestra el valor del primer dado, el del segundo y la suma de los dados, y devuelve esa suma. Después, el método `Jugar` entra a la instrucción `switch` en las líneas 33-49, en donde se utiliza el valor de `sumaDeDados` de la línea 30 para determinar si el jugador ganó o perdió el juego, o si debe continuar con otro tiro.

Las sumas de los dados que ocasionan que se gane o pierda el juego en el primer tiro se declaran en la enumeración `NombresDados`, en las líneas 14-21. Estos valores se utilizan en las etiquetas `case` de la instrucción `switch`. Los nombres de los identificadores utilizan los términos comunes en el casino para estas sumas. Observe que en la enumeración `NombresDados` se asigna un valor en forma explícita a cada nombre de identificador. Cuando se declara la enumeración, cada constante en la declaración `enum` contiene un valor constante subyacente de tipo `int`. Si no asigna un valor a un identificador en la declaración `enum`, el compilador lo hará por usted. Si la primera constante `enum` no está asignada, el compilador le asigna el valor 0. Si cualquier otra constante `enum` no está asignada, el compilador le asigna un valor igual a uno más que el valor de la constante `enum` anterior. Por ejemplo, en la enumeración `Estado`, el compilador asigna en forma implícita el valor 0 a `Estado.GANO`, el valor 1 a `Estado.CONTINUA` y el 2 a `Estado.PERDIO`.

También puede declarar un tipo subyacente de `enum` como `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` o `ulong`; para ello hay que escribir lo siguiente:

```
private enum MiEnum : nombreTipo { CONSTANTE1, CONSTANTE2, ... }
```

en donde *nombreTipo* representa uno de los tipos integrales simples.

Si necesita comparar un valor de tipo simple con el valor subyacente de una constante de enumeración, debe utilizar un operador de conversión para hacer que los dos tipos concuerden. En la instrucción `switch` de las líneas 33-49, utilizamos el operador de conversión para convertir el valor `int` en `sumaDados` al tipo `NombresDados` y lo comparamos con cada una de las constantes en `NombresDados`. Las líneas 35-36 determinan si el jugador ganó en

el primer tiro con SIETE (7) u ONCE (11). Las líneas 39-41 determinan si el jugador perdió en el primer tiro con DOS_UNOS (2), TRES (3) o DOCE (12). Después del primer tiro, si el juego no ha terminado, la etiqueta del caso `default` (líneas 44-48) almacena `sumaDeDados` en `miPunto` (línea 46) y muestra el punto (línea 47).

Si aún estamos tratando de “hacer nuestro punto” (es decir, el juego continúa de un tiro anterior), se ejecuta el ciclo de las líneas 52-63. En la línea 54 se tira el dado otra vez. Si `sumaDeDados` concuerda con `miPunto` en la línea 57, la línea 58 asigna a `estadoJuego` el valor `Estado.GANO` y el ciclo termina, ya que el juego está terminado. En la línea 61 utilizamos el operador de conversión `(int)` para obtener el valor subyacente de `NombresDados.SIETE`, de manera que podamos compararlo con `sumaDeDados`. Si `sumaDeDados` es igual a `SIETE` (7), la línea 62 asigna el valor `Estado.PERDIO` a `sumaDeDados` y el ciclo termina, ya que se acabó el juego. Cuando termina el juego, las líneas 66-69 muestran un mensaje en el que se indica si el jugador ganó o perdió y la aplicación termina.

Observe el uso de varios mecanismos de control del programa que hemos visto antes. La clase `Craps` utiliza dos métodos: `Jugar` (que se llama desde `PruebaCraps.Main`) y `TirarDados` (que se llama dos veces desde `Jugar`), y las instrucciones de control `switch`, `while`, `if...else` e `if anidado`. Observe también el uso de múltiples etiquetas `case` en la instrucción `switch` para ejecutar las mismas instrucciones para las sumas de SIETE y ONCE (líneas 35-36), y para las sumas de DOS_UNOS, TRES y DOCE (líneas 39-41).

7.11 Alcance de las declaraciones

Ya hemos visto declaraciones de varias entidades de C# como las clases, los métodos, las propiedades, las variables y los parámetros. Las declaraciones introducen nombres que pueden utilizarse para hacer referencia a dichas entidades de C#. El *alcance* de una declaración es la porción de la aplicación que puede hacer referencia a la entidad declarada, a través de su nombre descalificado. Dicha entidad se dice que está “dentro del alcance” para esa porción de la aplicación. En esta sección introduciremos varias cuestiones importantes relacionadas con el alcance. Para obtener más información sobre el alcance, consulte la *sección 3.7, Alcances*, de la *Especificación del lenguaje C#*.

Las reglas básicas de alcance son las siguientes:

1. El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece la declaración.
2. El alcance de la declaración de una variable local es a partir del punto en el cual aparece la declaración, hasta el final del bloque que contiene esa declaración.
3. El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for` y las demás expresiones en el encabezado.
4. El alcance de un método, propiedad o campo de una clase es todo el cuerpo de la clase. Esto permite a los métodos y propiedades `static` de la clase utilizar cualquiera de los campos, métodos y propiedades de la clase, sin importar el orden en el que estén declarados. De manera similar, los métodos y propiedades `static` pueden utilizar cualquiera de los miembros `static` de la clase.

Cualquier bloque puede contener declaraciones de variables. Si una variable local o parámetro en un método tiene el mismo nombre que un campo, el campo se oculta hasta que el bloque termina su ejecución. En el capítulo 9 veremos cómo acceder a los campos ocultos.



Tip de prevención de errores 7.3

Use nombres distintos para los campos y las variables locales, para ayudar a evitar los errores lógicos sutiles que se producen cuando se hace la llamada a un método y una variable local de ese método oculta un campo con el mismo nombre en la clase.

La aplicación en las figuras 7.10 y 7.11 demuestra las cuestiones de alcance con los campos y las variables locales. Cuando la aplicación empieza a ejecutarse, el método `Main` de la clase `PruebaAlcance` (figura 7.11, líneas 6-10) crea un objeto de la clase `Alcance` (línea 8) y llama al método `Iniciar` del objeto (línea 9) para producir el resultado de la aplicación (el cual se muestra en la figura 7.11).

En la clase `Alcance` (figura 7.11), la línea 8 declara e inicializa la variable de instancia `x` con 1. Esta variable de instancia está oculta en cualquier bloque (o método) que declare a la variable local de nombre `x`. El método `Iniciar` (líneas 12-31) declara la variable local `x` (línea 14) y la inicializa con 5. El valor de esta variable local se imprime en pantalla, para mostrar que la variable de instancia `x` (cuyo valor es 1) está oculta en el método `Iniciar`.

La aplicación declara otros dos métodos: `UsarVariableLocal` (líneas 34-43) y `UsarVariableInstancia` (líneas 46-53); cada uno de ellos no tiene argumentos y no devuelve resultados. El método `Iniciar` llama a cada método dos veces (líneas 19-28). El método `UsarVariableLocal` declara la variable local `x` (línea 36). Cuando se llama

```

1  // Fig. 7.10: Alcance.cs
2  // La clase Alcance demuestra los alcances de las variables de instancia y locales.
3  using System;
4
5  public class Alcance
6  {
7      // variable de instancia que es accesible para todos los métodos de esta clase
8      private int x = 1;
9
10     // el método Iniciar crea e inicializa la variable local x
11     // y llama a los métodos UsarVariableLocal y UsarVariableInstancia
12     public void Iniciar()
13     {
14         int x = 5; // la variable local x del método oculta a la variable de instancia x
15
16         Console.WriteLine("x local en el método Iniciar es {0}", x );
17
18         // UsarVariableLocal tiene su propia x local
19         UsarVariableLocal();
20
21         // UsarVariableInstancia utiliza la variable de instancia x de la clase Alcance
22         UsarVariableInstancia();
23
24         // UsarVariableLocal reinicializa su propia x local
25         UsarVariableLocal();
26
27         // la variable de instancia x de la clase Alcance retiene su valor
28         UsarVariableInstancia();
29
30         Console.WriteLine( "\nx local en el método Iniciar es {0}", x );
31     } // fin del método Iniciar
32
33     // crea e inicializa la variable local x durante cada llamada
34     public void UsarVariableLocal()
35     {
36         int x = 25; // se inicializa cada vez que se hace una llamada a UsarVariableLocal
37
38         Console.WriteLine(
39             "\nx local al entrar al método UsarVariableLocal es {0}", x );
40         x++; // modifica la variable local x de este método
41         Console.WriteLine(
42             "x local antes de salir del método UsarVariableLocal es {0}", x );
43     } // fin del método UsarVariableLocal
44
45     // modifica la variable de instancia x de la clase Alcance durante cada llamada
46     public void UsarVariableInstancia()
47     {
48         Console.WriteLine( "\nvariable de instancia x al entrar al {0} es {1}",
49             "método UsarVariableInstancia", x );
50         x *= 10; // modifica la variable de instancia x de la clase Alcance
51         Console.WriteLine( "variable de instancia x antes de salir del {0} es {1}",
52             "método UsarVariableInstancia", x );
53     } // fin del método UsarVariableInstancia
54 } // fin de la clase Alcance

```

Figura 7.10 | La clase `Alcance` demuestra los alcances de las variables de instancia y locales.


```

1 // Fig. 7.11: PruebaAlcance.cs
2 // Aplicación para probar la clase Alcance.
3 public class PruebaAlcance
4 {
5     // punto inicial de la aplicación
6     public static void Main( string[] args )
7     {
8         Alcance pruebaAlcance = new Alcance();
9         pruebaAlcance.Iniciar();
10    } // fin de Main
11 } // fin de la clase PruebaAlcance

```

x local en el método Iniciar es 5

x local al entrar al método UsarVariableLocal es 25

x local antes de salir del método UsarVariableLocal es 26

variable de instancia x al entrar al método UsarVariableInstancia es 1

variable de instancia x antes de salir del método UsarVariableInstancia es 10

x local al entrar al método UsarVariableLocal es 25

x local antes de salir del método UsarVariableLocal es 26

variable de instancia x al entrar al método UsarVariableInstancia es 10

variable de instancia x antes de salir del método UsarVariableInstancia es 100

x local en el método Iniciar es 5

Figura 7.11 | Aplicación para probar la clase Alcance.

por primera vez a `UsarVariableLocal` (línea 19), crea una variable local `x` y la inicializa con 25 (línea 36), muestra en pantalla el valor de `x` (líneas 38-39), incrementa `x` (línea 40) y muestra nuevamente su valor en pantalla (líneas 41-42). Cuando se llama a `UsarVariableLocal` por segunda vez (línea 25), vuelve a crear la variable local `x` y la reinicializa con 25, por lo que la salida de cada llamada a `UsarVariableLocal` es idéntica.

El método `UsarVariableInstancia` no declara variables locales. Por lo tanto, cuando hace referencia a `x`, se utiliza la variable de instancia `x` (línea 8) de la clase. Cuando el método `UsarVariableInstancia` se llama por primera vez (línea 22), muestra en pantalla el valor (1) de la variable de instancia `x` (líneas 48-49), multiplica la variable de instancia `x` por 10 (línea 50) y muestra nuevamente en pantalla el valor (10) de la variable de instancia `x` (líneas 51-52) antes de regresar. La siguiente vez que se llama al método `UsarVariableInstancia` (línea 28), la variable de instancia `x` tiene el valor modificado de 10, por lo que el método muestra en pantalla un 10 y después un 100. Por último, en el método `Iniciar` la aplicación muestra en pantalla el valor de la variable local `x` otra vez (línea 30), para mostrar que ninguna de las llamadas a los métodos modificó la variable local `x`, ya que todos los métodos hicieron referencia a las variables llamadas `x` en otros alcances.

7.12 Sobrecarga de métodos

Pueden declararse métodos con el mismo nombre en la misma clase, siempre y cuando tengan distintos conjuntos de parámetros (determinados en base al número, tipos y orden de los parámetros). A esto se le conoce como *sobrecarga de métodos*. Cuando se hace una llamada a un *método sobrecargado*, el compilador de C# selecciona el método apropiado mediante un análisis del número, tipos y orden de los argumentos en la llamada. Por lo general, la sobrecarga de métodos se utiliza para crear varios métodos con el mismo nombre que realicen la misma tarea o tareas similares, pero con distintos tipos o distintos números de argumentos. Por ejemplo, los métodos `Min` y `Max` de `Math` (sintetizados en la sección 7.3) se sobrecargan con 11 versiones. Estos métodos buscan el mínimo y el máximo, respectivamente, de dos valores de cada uno de los 11 tipos numéricos simples. Nuestro siguiente ejemplo muestra cómo declarar e invocar a métodos sobrecargados. En el capítulo 9 veremos ejemplos de constructores sobrecargados.

Declaración de métodos sobrecargados

En la clase `SobrecargaMetodos` (figura 7.12) incluimos dos versiones sobrecargadas de un método llamado `Cuadrado`: una que calcula el cuadrado de un `int` (y devuelve un `int`) y otra que calcula el cuadrado de un `double` (y devuelve un `double`). Aunque estos métodos tienen el mismo nombre, además de listas de parámetros y cuerpos similares, podemos considerarlos simplemente como métodos *diferentes*. Puede ser útil si consideramos los nombres de los métodos como “Cuadrado de `int`” y “Cuadrado de `double`”, en forma respectiva. Cuando la aplicación empieza a ejecutarse, el método `Main` de la clase `PruebaSobrecargaMetodos` (figura 7.13, líneas 5-9) crea un objeto de la clase `SobrecargaMetodos` (línea 7) y llama al método `ProbarMetodosSobrecargados` del objeto (línea 8) para producir la salida de la aplicación (figura 7.13).

En la figura 7.12, la línea 10 invoca al método `Cuadrado` con el argumento 7. Los valores enteros literales se tratan como de tipo `int`, por lo que la llamada al método en la línea 10 invoca a la versión de `Cuadrado` de las

```

1  // Fig. 7.12: SobrecargaMetodos.cs
2  // Declaraciones de métodos sobrecargados.
3  using System;
4
5  public class SobrecargaMetodos
6  {
7      // prueba los métodos Cuadrado sobrecargados
8      public void ProbarMetodosSobrecargados()
9      {
10         Console.WriteLine( "El cuadrado del integer 7 es {0}", Cuadrado( 7 ) );
11         Console.WriteLine( "El cuadrado del double 7.5 es {0}", Cuadrado( 7.5 ) );
12     } // fin del método ProbarMetodosSobrecargados
13
14     // Método Cuadrado con argumento int
15     public int Cuadrado( int valorInt )
16     {
17         Console.WriteLine( "Se llamó a Cuadrado con argumento int: {0}",
18             valorInt );
19         return valorInt * valorInt;
20     } // fin del método Cuadrado con argumento int
21
22     // Método Cuadrado con argumento double
23     public double Cuadrado( double valorDouble )
24     {
25         Console.WriteLine( "Se llamó a Cuadrado con argumento double: {0}",
26             valorDouble );
27         return valorDouble * valorDouble;
28     } // fin del método Cuadrado con argumento double
29 } // fin de la clase SobrecargaMetodos

```

Figura 7.12 | Declaraciones de métodos sobrecargados.

```

1  // Fig. 7.13: PruebaSobrecargaMetodos.cs
2  // Aplicación para probar la clase SobrecargaMetodos.
3  public class PruebaSobrecargaMetodos
4  {
5      public static void Main( string[] args )
6      {
7          SobrecargaMetodos sobrecargaMetodos = new SobrecargaMetodos();
8          sobrecargaMetodos.ProbarMetodosSobrecargados();
9      } // fin de Main
10 } // fin de la clase PruebaSobrecargaMetodos

```

Figura 7.13 | Aplicación para probar la clase `SobrecargaMetodos`. (Parte 1 de 2).

```
Se llamó a Cuadrado con argumento int: 7
El cuadrado del integer 7 es 49
Se llamó a Cuadrado con argumento double: 7.5
El cuadrado del double 7.5 es 56.25
```

Figura 7.13 | Aplicación para probar la clase `SobrecargaMetodos`. (Parte 2 de 2).

líneas 15-20, la cual especifica un parámetro `int`. De manera similar, la línea 11 invoca al método `Cuadrado` con el argumento 7.5. Los valores de números reales literales se tratan como de tipo `double`, por lo que la llamada al método en la línea 11 invoca a la versión de `Cuadrado` de las líneas 23-28, la cual especifica un parámetro `double`. Cada método imprime en pantalla primero una línea de texto, para mostrar que se llamó al método apropiado en cada caso.

Observe que los métodos sobrecargados en la figura 7.12 realizan el mismo cálculo, pero con dos tipos distintos. La nueva característica genérica de C# proporciona un mecanismo para escribir un solo “método genérico” que puede realizar las mismas tareas que todo un conjunto de métodos sobrecargados. En el capítulo 25 veremos los métodos genéricos.

Cómo se diferencian los métodos sobrecargados entre sí

El compilador diferencia los métodos sobrecargados en base a su *firma*: una combinación del nombre del método y del número, tipos y orden de sus parámetros. La firma también incluye la forma en que se pasan esos parámetros, los cuales pueden modificarse mediante las palabras `ref` y `out` que veremos en la sección 7.14. Si el compilador sólo se fijara en los nombres de los métodos durante la compilación, el código de la figura 7.12 sería ambiguo; el compilador no sabría cómo distinguir entre los dos métodos `Cuadrado` (líneas 15-20 y 23-28). De manera interna, el compilador utiliza las firmas para determinar si los métodos en una clase son únicos.

Por ejemplo, en la figura 7.12 el compilador utiliza las firmas de los métodos para diferenciar entre el método “`Cuadrado de int`” (el método `Cuadrado` que especifica un parámetro `int`) y el método “`Cuadrado de double`” (el método `Cuadrado` que especifica un parámetro `double`). Si la declaración de `Metodo1` empieza así:

```
void Metodo1( int a, float b )
```

entonces ese método tendrá una firma distinta a la del método que se declara de la siguiente manera:

```
void Metodo1( float a, int b )
```

El orden de los tipos de los parámetros es importante; el compilador considera que los dos encabezados anteriores de `Metodo` son distintos.

Tipo de valor de retorno de los métodos sobrecargados

Cuando hablamos sobre los nombres lógicos de los métodos que utiliza el compilador, no mencionamos los tipos de valor de retorno de los métodos. Esto se debe a que las *llamadas* a los métodos no pueden diferenciarse en base al tipo de valor de retorno. La aplicación de la figura 7.14 ilustra los errores que genera el compilador cuando dos métodos tienen la misma firma, pero distintos tipos de valores de retorno. Los métodos sobrecargados pueden tener tipos de valor de retorno iguales o distintos, si los métodos tienen distintas listas de parámetros. Además, los métodos sobrecargados no necesitan tener el mismo número de parámetros.



Error común de programación 7.10

Declarar métodos sobrecargados con listas de parámetros idénticas es un error de compilación, sin importar que los tipos de los valores de retorno sean distintos.

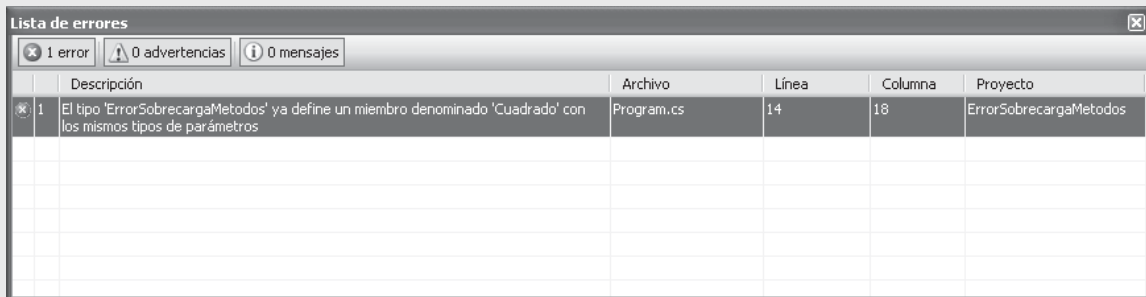
7.13 Recursividad

Las aplicaciones que hemos visto hasta ahora se estructuran de manera general como métodos que llaman a otro método en una forma disciplinada y jerárquica. No obstante, para algunos problemas es útil que un método se llame a sí mismo. Un *método recursivo* se llama a sí mismo, ya sea en forma directa o indirecta a través de otro método.

```

1 // Fig. 7.14: ErrorSobrecargaMetodos.cs
2 // Los métodos sobrecargados con firmas idénticas provocan errores de
3 // compilación, aun si los tipos de los valores de retorno son distintos.
4 public class ErrorSobrecargaMetodos
5 {
6     // declaración del método Cuadrado con argumento int
7     public int Cuadrado( int x )
8     {
9         return x * x;
10    }
11
12    // la segunda declaración del método Cuadrado con argumento int provoca un error
13    // de compilación, aun y cuando los tipos de los valores de retorno son distintos
14    public double Cuadrado( int y )
15    {
16        return y * y;
17    }
18 } // fin de la clase ErrorSobrecargaMetodos

```



	Descripción	Archivo	Línea	Columna	Proyecto
1	El tipo 'ErrorSobrecargaMetodos' ya define un miembro denominado 'Cuadrado' con los mismos tipos de parámetros	Program.cs	14	18	ErrorSobrecargaMetodos

Figura 7.14 | Los métodos sobrecargados con firmas idénticas pueden provocar errores de compilación, aun si sus tipos de valor de retorno son distintos.

Primero consideraremos la recursividad en forma conceptual. Después analizaremos una aplicación que contiene un método recursivo. Los métodos para solucionar problemas recursivos tienen un número de elementos en común. Cuando se hace la llamada a un método recursivo para resolver un problema, este método en realidad es capaz de resolver sólo el (los) caso(s) más simple(s), o *caso(s) base*. Si se hace una llamada al método con un caso base, el método devuelve un resultado. Si se hace una llamada al método con un problema más complejo, éste divide el problema en dos piezas conceptuales: una que el método sabe cómo resolver y la otra que no sabe cómo. Para que la recursividad sea factible, la pieza que no se puede resolver debe ser similar al problema original, pero debe ser una versión un poco más simple o pequeña de éste. Como este nuevo problema se ve igual que el problema original, el método llama a una copia nueva de sí mismo para trabajar sobre el problema más pequeño; a esto se le conoce como *llamada recursiva* y o *paso de recursividad*. Por lo general, el paso de recursividad incluye una instrucción `return`, ya que su resultado se combinará con la porción del problema que el método supo cómo resolver, para formar un resultado que se pasará de vuelta al método original que hizo la llamada.

El paso de recursividad se ejecuta mientras la llamada original al método sigue estando activa (es decir, mientras no haya finalizado la ejecución). El paso de recursividad puede originar muchas más llamadas recursivas, a medida que el método divida cada nuevo subproblema en dos piezas conceptuales. Para que la recursividad pueda terminar en un momento dado, cada vez que el método se llama a sí mismo con una versión un poco más simple del problema original, la secuencia de problemas cada vez más pequeños debe convergir en el caso base. En ese punto, el método reconoce el caso base y devuelve un resultado a la copia anterior del método. Después se produce una secuencia de instrucciones `return` hasta que la llamada original al método devuelve el resultado al método que hizo la llamada. Este proceso suena complejo en comparación con la solución convencional de problemas que hemos realizado hasta ahora.

Cálculos recursivos del factorial

Como ejemplo del funcionamiento de los conceptos de recursividad, escribiremos una aplicación recursiva para realizar un cálculo matemático popular. Considere el factorial de un entero n no negativo, que se escribe como $n!$ (y se pronuncia como “factorial de n ”), y que es el producto:

$$n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

$1!$ es igual a 1 y $0!$ se define también como 1. Por ejemplo, $5!$ es el producto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

El factorial de un entero llamado `numero` mayor o igual a 0 puede calcularse en forma iterativa (sin recursividad) mediante el uso de la siguiente instrucción `for`:

```
factorial = 1;
for ( int contador = numero; contador >= 1; contador-- )
    factorial *= contador;
```

Para llegar a una declaración recursiva del método del factorial, observemos la siguiente relación:

$$n! = n \cdot (n-1)!$$

Por ejemplo, es evidente que $5!$ es igual a $5 \cdot 4!$, como se muestra mediante las siguientes ecuaciones:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

La evaluación de $5!$ procedería como se muestra en la figura 7.15. La figura 7.15(a) muestra cómo procede la sucesión de llamadas recursivas hasta que se evalúa $1!$ como 1, con lo cual termina la recursividad. La figura 7.15(b) muestra los valores devueltos de cada llamada recursiva al método que hizo la llamada, hasta que el valor se calcula y se devuelve.

La figura 7.16 utiliza la recursividad para calcular e imprimir los factoriales de los enteros del 0 al 10. El método recursivo `Factorial` (líneas 16-24) primero evalúa para determinar si una condición de terminación (línea 19) es `true`. Si `numero` es menor o igual a 1 (el caso base), `Factorial` devuelve 1, no se requiere más recursividad y el método regresa. Si `numero` es mayor que 1, la línea 23 expresa el problema como el producto de `numero` y una llamada recursiva a `Factorial`, en la que se evalúa el factorial de `numero - 1`, lo cual es un problema un poco más simple que el cálculo original de `Factorial( numero )`.

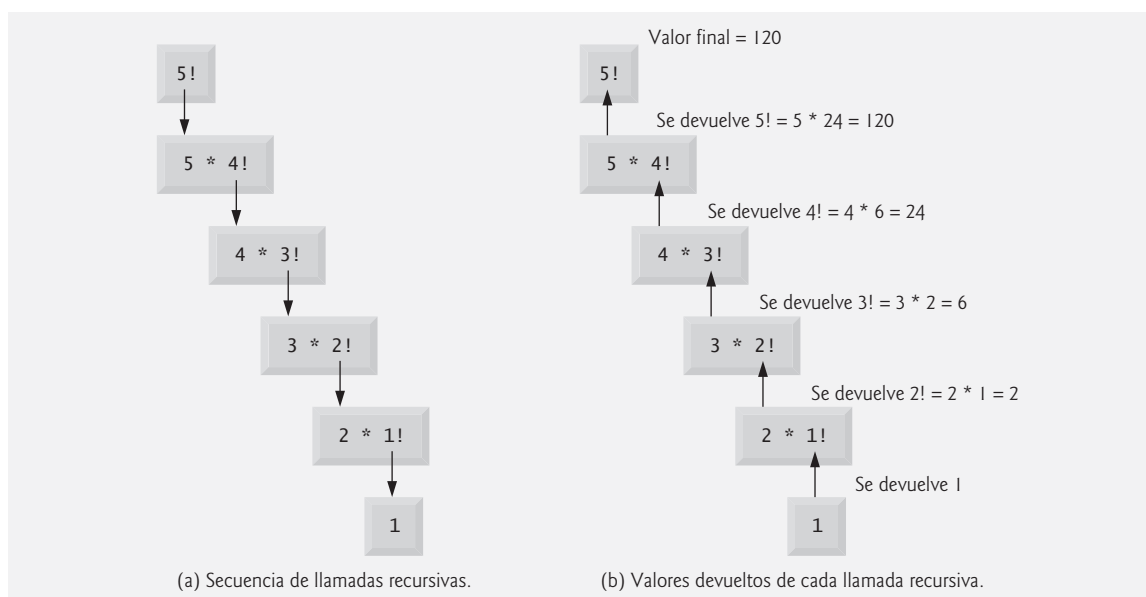


Figura 7.15 | Evaluación recursiva de $5!$

```

1 // Fig. 7.16: PruebaFactorial.cs
2 // Método recursivo Factorial.
3 using System;
4
5 public class PruebaFactorial
6 {
7     public static void Main( string[] args )
8     {
9         // calcula los factoriales del 0 al 10
10        for ( long contador = 0; contador <= 10; contador++ )
11            Console.WriteLine( "{0}! = {1}",
12                               contador, Factorial( contador ) );
13    } // fin del método Main
14
15    // declaración recursiva del método Factorial
16    public static long Factorial( long numero )
17    {
18        // caso base
19        if ( numero <= 1 )
20            return 1;
21        // paso de recursividad
22        else
23            return numero * Factorial( numero - 1 );
24    } // fin del método Factorial
25 } // fin de la clase PruebaFactorial

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

Figura 7.16 | Método recursivo Factorial.

El método `Factorial` (líneas 16-24) recibe un parámetro de tipo `long` y devuelve un resultado del mismo tipo. Como podemos ver en la figura 7.16, los valores de los factoriales se vuelven extensos con mucha rapidez. Seleccionamos el tipo `long` (que puede representar enteros relativamente extensos) de manera que la aplicación pudiera calcular factoriales mayores de 20!. Por desgracia, el método `Factorial` produce valores extensos con tanta rapidez que los valores de los factoriales exceden rápidamente incluso hasta al valor máximo que puede almacenarse en una variable `long`. Debido a las restricciones en los tipos integrales, podría ser necesario utilizar en última instancia variables tipo `float`, `double` y `decimal` para calcular factoriales de números más grandes. Esta situación apunta a una debilidad en muchos lenguajes de programación: los lenguajes no pueden extenderse fácilmente para manejar los requerimientos únicos de varias aplicaciones. Como usted sabe, C# es un lenguaje extensible que le permite crear un tipo que soporte enteros arbitrariamente extensos, si así lo desea. Por ejemplo, podría crear una clase `EnteroEnorme` que permitiera a una aplicación calcular los factoriales de números arbitrariamente extensos.



Error común de programación 7.11

Si se omite el caso base o el paso de recursividad se escribe en forma incorrecta, de tal forma que no converja en el caso base, se producirá una recursividad infinita, con lo cual la memoria se agotará en un momento dado. Este error es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva).

7.14 Paso de argumentos: diferencia entre paso por valor y paso por referencia

Dos maneras de pasar argumentos a las funciones en muchos lenguajes de programación son: *paso por valor* y *paso por referencia*. Cuando se pasa un argumento por valor (la opción predeterminada en C#), se crea una copia del valor del argumento y se pasa a la función que se llamó. Los cambios en la copia no afectan al valor de la variable original en el método que hizo la llamada. Esto evita los efectos secundarios accidentales que tanto entorpecen el desarrollo de sistemas de software correctos y confiables. Cada argumento que se ha pasado en los programas de este capítulo hasta ahora, se ha pasado por valor. Cuando se pasa un argumento por referencia, el método que hace la llamada proporciona al método llamado la habilidad de acceder y modificar la variable original del método que llama.



Tip de rendimiento 7.1

El paso por referencia es bueno por cuestiones de rendimiento, ya que puede eliminar la sobrecarga implicada en el paso por valor, en relación con el proceso de copiar grandes cantidades de datos.



Observación de ingeniería de software 7.5

El paso por referencia puede debilitar la seguridad, ya que la función llamada puede corromper los datos del que hace la llamada.

Para pasar un objeto por referencia en un método, sólo hay que proporcionar como argumento para la llamada al método la variable que hace referencia al objeto. Después, en el cuerpo del método se hace referencia al objeto, utilizando el nombre del parámetro. Este parámetro hace referencia al objeto original en memoria, por lo que el método llamado puede acceder directamente al objeto original.

Anteriormente hablamos sobre la diferencia entre los tipos por valor y los tipos por referencia. Una de las principales diferencias es que las variables de tipo por valor almacenan valores, por lo que si se especifica una variable de tipo por valor en la llamada a un método, se pasa una copia del valor de la variable al método. Las variables de tipo por referencia almacenan referencias a objetos, por lo que si se especifica una variable de tipo por referencia como argumento, se pasa al método una copia de la referencia actual al objeto. Aun y cuando la referencia en sí se pasa por valor, el método de todas formas puede utilizar la referencia que recibe para modificar el objeto original en memoria. De manera similar, cuando se devuelve información de un método a través de una instrucción `return`, el método devuelve una copia del valor almacenado en una variable de tipo por valor, o una copia de la referencia almacenada en una variable de tipo por referencia. Cuando se devuelve una referencia, el método que se llamó puede utilizar esa referencia para interactuar con el objeto referenciado. Así que, en efecto, los objetos siempre se pasan por referencia.

¿Qué sucede si usted desea pasar una variable por referencia, para que el método que se llamó pueda modificar el valor de esa variable? Para ello, C# cuenta con las palabras clave **ref** y **out**. Si aplica la palabra clave **ref** a la declaración de un parámetro, puede pasar una variable a un método por referencia; el método que se llamó podrá modificar la variable original en el método que hizo la llamada. La palabra clave **ref** se utiliza para variables que ya se han inicializado en el método que hace la llamada. Por lo general, cuando la llamada a un método contiene una variable no inicializada como argumento, el compilador genera un error. Si se antepone la palabra clave **out** al nombre de un parámetro se crea un *parámetro de salida*. Esto indica al compilador que el argumento se va a pasar al método que se llamó por referencia, y que el método que se llamó asignará un valor a la variable original en el método que hizo la llamada. Si el método no asigna un valor al parámetro de salida en todas las rutas posibles de ejecución, el compilador genera un error. Esto también evita que el compilador genere un mensaje de error para una variable no inicializada que se pase como argumento para un método. Un método sólo puede devolver un valor al método que lo llamó a través de una instrucción `return`, pero puede devolver muchos valores si especifica múltiples parámetros de salida.

También se puede pasar por referencia una variable de tipo por referencia, lo que permite modificar la variable de tipo por referencia que se pasó, de manera que haga referencia a un nuevo objeto. La técnica de pasar una referencia por referencia es engañosa pero poderosa, y la analizaremos en la sección 8.8.

La aplicación en las figuras 7.17 y 7.18 utiliza las palabras clave `ref` y `out` para manipular valores enteros. La clase `ParametrosPorReferenciaYDeSalida` (figura 7.17) contiene tres métodos que calculan el cuadrado de un entero. El método `CuadradoRef` (líneas 37-40) multiplica su parámetro `x` por sí mismo y asigna el nuevo valor a `x`. El parámetro `x` de `CuadradoRef` se declara como `ref int`, lo que indica que el argumento que se pase a este método debe ser un entero que se pase por referencia. Como el argumento se pasa por referencia, la asignación en la línea 39 modifica el valor del argumento original en el método que hace la llamada.

El método `CuadradoSal` (líneas 44-48) asigna a su parámetro el valor 6 (línea 46) y después calcula el cuadrado de ese valor. El parámetro de `CuadradoSal` se declara como `out int`, lo que indica que el argumento que se pase a este método debe ser un entero que se pase por referencia, y que el argumento no necesita inicializarse de antemano.

El método `Cuadrado` (líneas 52-55) multiplica su parámetro `x` por sí mismo y asigna el nuevo valor a `x`. Cuando se hace la llamada a este método, se pasa una copia del argumento al parámetro `x`. Por ende, aun y cuando el parámetro `x` se modifica en el método, no se modifica el valor original en el método que hace la llamada.

```

1  // Fig. 7.17: ParametrosPorReferenciaYDeSalida.cs
2  // Parámetros por referencia, de salida y por valor.
3  using System;
4
5  class ParametrosPorReferenciaYDeSalida
6  {
7      // llama a métodos con parámetros por referencia, de salida y por valor
8      public void DemostrarParametrosPorReferenciaYDeSalida()
9      {
10         int y = 5; // inicializa y con 5
11         int z; // declara z, pero no la inicializa
12
13         // muestra los valores originales de y y z
14         Console.WriteLine( "Valor original de y: {0}", y );
15         Console.WriteLine( "Valor original de z: sin inicializar\n" );
16
17         // paso de y y z por referencia
18         CuadradoRef( ref y ); // debe usar la palabra clave ref
19         CuadradoSal( out z ); // debe usar la palabra clave out
20
21         // muestra los valores de y y z después de ser modificados por
22         // los métodos CuadradoRef y CuadradoSal, respectivamente
23         Console.WriteLine( "Valor de y después de CuadradoRef: {0}", y );
24         Console.WriteLine( "Valor de z después de CuadradoSal: {0}\n", z );
25
26         // paso de y y z por valor
27         Cuadrado( y );
28         Cuadrado( z );
29
30         // muestra los valores de y y z después de pasarlos al método Cuadrado
31         // para demostrar que los argumentos pasados por valor no se modifican
32         Console.WriteLine( "Valor de y después de Cuadrado: {0}", y );
33         Console.WriteLine( "Valor de z después de Cuadrado: {0}", z );
34     } // fin del método DemostrarParametrosPorReferenciaYDeSalida
35
36     // usa el parámetro por referencia x para modificar la variable del que hace la llamada
37     void CuadradoRef( ref int x )
38     {
39         x = x * x; // calcula el cuadrado del valor de la variable del que hace la llamada
40     } // fin del método CuadradoRef
41

```

Figura 7.17 | Parámetros por referencia, de salida y por valor. (Parte 1 de 2).


```

42 // usa el parámetro de salida x para asignar un valor
43 // a una variable sin inicializar
44 void CuadradoSal( out int x )
45 {
46     x = 6; // asigna un valor a la variable del que hace la llamada
47     x = x * x; // calcula el cuadrado del valor de la variable del que hace la llamada
48 } // fin del método CuadradoSal
49
50 // el parámetro x recibe una copia del valor que se pasa como argumento,
51 // por lo que este método no puede modificar la variable del que hace la llamada
52 void Cuadrado( int x )
53 {
54     x = x * x;
55 } // fin del método Cuadrado
56 } // fin de la clase ParametrosPorReferenciaYDeSalida

```

Figura 7.17 | Parámetros por referencia, de salida y por valor. (Parte 2 de 2).

```

1 // Fig. 7.18: PruebaParametrosPorReferenciaYDeSalida.cs
2 // Aplicación para probar la clase ParametrosPorReferenciaYDeSalida.
3 class PruebaParametrosPorReferenciaYDeSalida
4 {
5     static void Main( string[] args )
6     {
7         ParametrosPorReferenciaYDeSalida prueba =
8             new ParametrosPorReferenciaYDeSalida();
9         prueba.DemostrarParametrosPorReferenciaYDeSalida();
10    } // fin de Main
11 } // fin de la clase PruebaParametrosPorReferenciaYDeSalida

```

```

Valor original de y: 5
Valor original de z: sin inicializar

Valor de y después de CuadradoRef: 25
Valor de z después de CuadradoSal: 36

Valor de y después de Cuadrado: 25
Valor de z después de Cuadrado: 36

```

Figura 7.18 | Aplicación para probar la clase ParametrosPorReferenciaYDeSalida.

El método `DemostrarParametrosPorReferenciaYDeSalida` (líneas 8-34) invoca a los métodos `CuadradoRef`, `CuadradoSal` y `Cuadrado`. Este método empieza por inicializar la variable `y` con 5 y declara, pero no inicializa, la variable `z`. Las líneas 18-19 llaman a los métodos `CuadradoRef` y `CuadradoSal`. Observe que cuando se pasa una variable a un método con un parámetro por referencia, es necesario anteponer al argumento la misma palabra clave (`ref` o `out`) que se utilizó para declarar el parámetro por referencia. Las líneas 23-24 muestran los valores de `y` y `z` después de las llamadas a `CuadradoRef` y `CuadradoSal`. Observe que `y` se cambió a 25 y que a `z` se le asignó 36.

Las líneas 27-28 llaman al método `Cuadrado` con `y` y `z` como argumentos. En este caso, ambas variables se pasan por valor; sólo se pasan copias de sus valores a `Cuadrado`. Como resultado, los valores de `y` y de `z` siguen siendo 25 y 36, respectivamente. Las líneas 32-33 muestran en pantalla los valores de `y` y `z` para mostrar que no se modificaron.



Error común de programación 7.12

Los argumentos `ref` y `out` en la llamada a un método deben concordar con los parámetros especificados en la declaración del método; en caso contrario, se produce un error de compilación.



Observación de ingeniería de software 7.6

De manera predeterminada, C# no le permite elegir si va a pasar cada argumento por valor o por referencia. Los tipos por valor se pasan por valor. Los objetos no se pasan a los métodos, sino que se pasan referencias a esos objetos. Las referencias en sí se pasan por valor. Cuando un método recibe una referencia a un objeto, el método puede manipular el objeto en forma directa, pero el valor por referencia no se puede modificar para que haga referencia al nuevo objeto. En la sección 8.8 veremos que las referencias también se pueden pasar por referencia.

7.15 (Opcional) Caso de estudio de ingeniería de software: identificación de las operaciones de las clases en el sistema ATM

En las secciones tituladas “Caso de estudio de ingeniería de software” al final de los capítulos 4-6, llevamos a cabo los primeros pasos en el diseño orientado a objetos de nuestro sistema ATM. En el capítulo 4 identificamos las clases que muy probablemente necesitaremos implementar, y creamos nuestro primer diagrama de clases. En el capítulo 5 describimos varios atributos de nuestras clases. En el capítulo 6 examinamos los estados de nuestros objetos y modelamos sus transiciones de estado y actividades. En esta sección determinaremos algunas de las operaciones (o comportamientos) de las clases que son necesarias para implementar el sistema ATM.

Identificar las operaciones

Una operación es un servicio que proporcionan los objetos de una clase a los clientes de esa clase. Considere las operaciones de algunos objetos reales. Las operaciones de un radio incluyen el sintonizar su estación y ajustar su volumen (que por lo general lo hace una persona que ajusta los controles del radio). Las operaciones de un auto incluyen acelerar (operación invocada por el conductor cuando oprime el pedal del acelerador), desacelerar (operación invocada por el conductor cuando oprime el pedal del freno o cuando suelta el pedal del acelerador), dar vuelta y cambiar velocidades. Los objetos de software también pueden ofrecer operaciones; por ejemplo, un objeto de gráficos de software podría ofrecer operaciones para dibujar un círculo, dibujar una línea y dibujar un cuadrado. Un objeto de software de hoja de cálculo podría ofrecer operaciones como imprimir la hoja de cálculo, totalizar los elementos en una fila o columna, y graficar la información de la hoja de cálculo en un gráfico de barras o de pastel.

Podemos derivar muchas de las operaciones de las clases en nuestro sistema ATM mediante un análisis de los verbos y las frases verbales en el documento de requerimientos. Después relacionamos cada una de ellas con las clases específicas en nuestro sistema (figura 7.19). Las frases verbales en la figura 7.19 nos ayudan a determinar las operaciones de nuestras clases.

Clase	Verbos y frases verbales
ATM	ejecuta transacciones financieras
SolicitudSaldo	[ninguna en el documento de requerimientos]
Retiro	[ninguna en el documento de requerimientos]
Deposito	[ninguna en el documento de requerimientos]
BaseDeDatosBanco	autentica a un usuario, extrae el saldo de una cuenta, abona a una cuenta, carga a una cuenta
Cuenta	extrae el saldo de una cuenta, abona un monto de depósito a una cuenta, carga un monto de retiro a una cuenta
Pantalla	muestra un mensaje al usuario
Teclado	recibe entrada numérica del usuario
DispensadorEfectivo	dispensa efectivo, indica si contiene suficiente efectivo para satisfacer una solicitud de retiro
RanuraDeposito	recibe un sobre de depósito

Figura 7.19 | Verbos y frases verbales para cada clase en el sistema ATM.

Modelar las operaciones

Para identificar las operaciones, analizamos las frases verbales que se listan para cada clase en la figura 7.19. La frase “ejecuta transacciones financieras” asociada con la clase ATM implica que la clase ATM instruye a las transacciones a que se ejecuten. Por lo tanto, cada una de las clases `SolicitudSaldo`, `Retiro` y `Deposito` necesitan una operación para proporcionar este servicio al ATM. Colocamos esta operación (que hemos nombrado `Ejecutar`) en el tercer compartimiento de las tres clases de transacciones en el diagrama de clases actualizado de la figura 7.20. Durante una sesión con el ATM, el objeto ATM invocará a la operación `Ejecutar` de cada objeto de transacción, para decirle que se ejecute.

Para representar las operaciones (que se implementan en forma de métodos en C#), UML lista el nombre de la operación, seguido de una lista separada por comas de parámetros entre paréntesis, un signo de punto y coma y el tipo de valor de retorno:

nombreOperación(parámetro1, parámetro2, ..., parámetroN) : tipoRetorno

Cada parámetro en la lista separada por comas consiste en un nombre de parámetro, seguido de un signo de punto y coma y del tipo del parámetro:

nombreParámetro : tipoParámetro

Por el momento, no listamos los parámetros de nuestras operaciones; en breve identificaremos y modelaremos los parámetros de algunas de las operaciones. Para algunas de estas operaciones no conocemos todavía los tipos de valores de retorno, por lo que también las omitiremos del diagrama. Estas omisiones son perfectamente normales en este punto. A medida que avancemos en nuestro proceso de diseño e implementación, agregaremos el resto de los tipos de valores de retorno.

Operaciones de las clases `BaseDatosBanco` y `Cuenta`

La figura 7.19 lista la frase “autentica a un usuario” enseguida de la clase `BaseDatosBanco`; la base de datos es el objeto que contiene la información necesaria de la cuenta para determinar si el número de cuenta y el NIP introducidos por un usuario concuerdan con los de una cuenta en el banco. Por lo tanto, la clase `BaseDatosBanco` necesita una operación que proporcione un servicio de autenticación al ATM. Colocamos la operación `AutenticarUsuario` en el tercer compartimiento de la clase `BaseDatosBanco` (figura 7.20). No obstante, un objeto de la clase `Cuenta` y no de la clase `BaseDatosBanco` es el que almacena el número de cuenta y el NIP a los que se debe acceder para autenticar a un usuario, por lo que la clase `Cuenta` debe proporcionar un servicio para validar un NIP obtenido como entrada del usuario, y compararlo con un NIP almacenado en un objeto `Cuenta`. Por ende, agregamos una operación `ValidarNIP` a la clase `Cuenta`. Observe que especificamos un tipo de valor de retorno `bool` para las operaciones `AutenticarUsuario` y `ValidarNIP`. Cada operación devuelve un valor que indica que la operación tuvo éxito al realizar su tarea (es decir, un valor de retorno `true`) o que no tuvo éxito (es decir, un valor de retorno `false`).

La figura 7.19 lista varias frases verbales adicionales para la clase `BaseDatosBanco`: “extrae el saldo de una cuenta”, “abona a una cuenta” y “carga a una cuenta”. Al igual que “autentica a un usuario”, estas frases restantes se refieren a los servicios que debe proporcionar la base de datos al ATM, ya que ésta almacena todos los datos de las cuentas que se utilizan para autenticar a un usuario y realizar transacciones con el ATM. No obstante, los objetos de la clase `Cuenta` son los que en realidad realizan las operaciones a las que se refieren estas frases. Por ello, tanto la clase `BaseDatosBanco` como la clase `Cuenta` necesitan operaciones que correspondan a cada una de estas frases. En la sección 4.11 vimos que, como una cuenta de banco contiene información delicada, no permitimos que el ATM acceda a las cuentas en forma directa. La base de datos actúa como un intermediario entre el ATM y los datos de la cuenta, evitando el acceso no autorizado. Como veremos en la sección 8.14, la clase ATM invoca las operaciones de la clase `BaseDatosBanco`, cada una de las cuales a su vez invoca a las operaciones correspondientes (que son descriptores de acceso `get` de propiedades de sólo lectura) en la clase `Cuenta`.

La frase “extrae el saldo de una cuenta” sugiere que las clases `BaseDatosBanco` y `Cuenta` necesitan una operación para obtener el saldo. Sin embargo, recuerde que la figura 5.16 especificó dos atributos en la clase `Cuenta` para representar un saldo: `saldoDisponible` y `saldoTotal`. Una solicitud de saldo requiere el acceso a estos

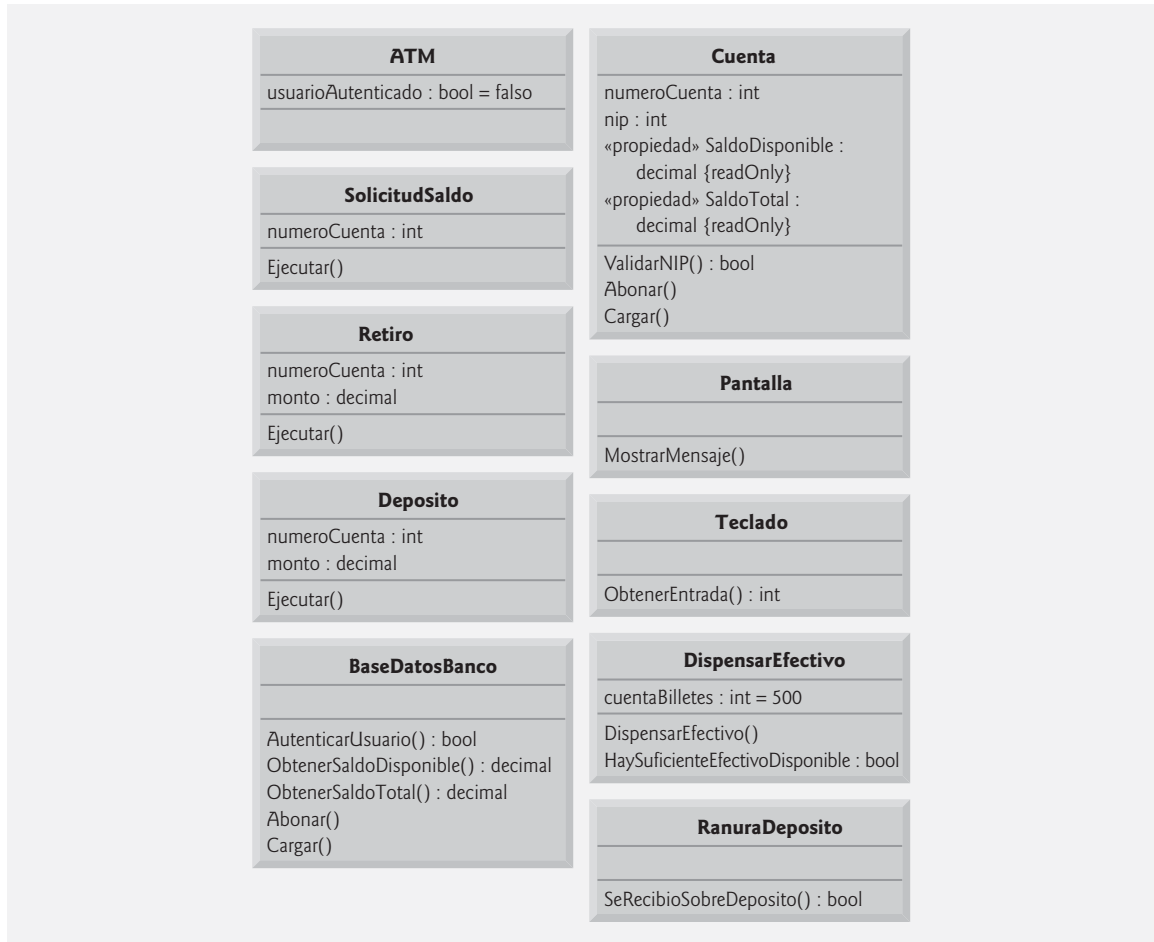


Figura 7.20 | Clases en el sistema ATM con atributos y operaciones.

dos atributos del saldo, de manera que pueda mostrarlos al usuario, pero un retiro sólo requiere verificar el valor de `saldoDisponible`. Para permitir que los objetos en el sistema obtengan estos atributos de saldo en forma individual de un objeto `Cuenta` específico en la clase `BaseDatosBanco`, agregamos las operaciones `ObtenerSaldoDisponible` y `ObtenerSaldoTotal` al tercer compartimiento de la clase `BaseDatosBanco` (figura 7.20). Especificamos un tipo de valor de retorno `decimal` para cada una de estas operaciones, debido a que los saldos que van a extraer son de tipo `decimal`.

Una vez que la `BaseDatosBanco` sabe a cuál `Cuenta` acceder, debe ser capaz de obtener cada atributo de saldo de manera individual de esa `Cuenta`. Para este propósito podríamos agregar las operaciones `ObtenerSaldoDisponible` y `ObtenerSaldoTotal` al tercer compartimiento de la clase `Cuenta` (figura 7.20). No obstante, en C# las operaciones simples como obtener el valor de un atributo, por lo general, se realizan mediante el descriptor de acceso `get` de una propiedad (por lo menos cuando esa clase específica “posee” el atributo subyacente). Como este diseño es para una aplicación en C#, en vez de modelar las operaciones `ObtenerSaldoDisponible` y `ObtenerSaldoTotal`, modelaremos las propiedades `decimal SaldoDisponible` y `SaldoTotal` en la clase `Cuenta`. Las propiedades se colocan en el segundo compartimiento de un diagrama de clases. Estas propiedades sustituyen a los atributos `saldoDisponible` y `saldoTotal` que modelamos para la clase `Cuenta` en la figura 5.16. En el capítulo 4 vimos que los descriptors de acceso de una propiedad son implícitos; por ende, no se modelan en un diagrama de clases. La figura 7.19 no menciona la necesidad de establecer los saldos, por lo que la figura 7.20 muestra las propiedades `SaldoDisponible` y `SaldoTotal` como propiedades de sólo lectura (es decir, sólo tienen

descriptores de acceso `get`). Para indicar una propiedad de sólo lectura en UML, colocamos “{readOnly}” después del tipo de la propiedad.

Tal vez se pregunte por qué modelamos las *propiedades* `SaldoDisponible` y `SaldoTotal` en la clase `Cuenta` y las *operaciones* `ObtenerSaldoDisponible` y `ObtenerSaldoTotal` en la clase `BaseDatosBanco`. Como puede haber muchos objetos `Cuenta` en la `BaseDatosBanco`, el ATM debe especificar a qué `Cuenta` acceder cuando invoca a las operaciones `ObtenerSaldoDisponible` y `ObtenerSaldoTotal` de `BaseDatosBanco`. Para ello, el ATM pasa un argumento de número de cuenta a cada operación de `BaseDatosBanco`. Los descriptores de acceso `get` de las propiedades que hemos visto en el código de C# no pueden recibir argumentos. Por lo tanto, modelamos a `ObtenerSaldoDisponible` y `ObtenerSaldoTotal` como operaciones en la clase `BaseDatosBanco`, de manera que podamos especificar parámetros a los cuales el ATM pueda pasar argumentos. Además, los atributos de saldo subyacentes no son propiedad de `BaseDatosBanco`, por lo que los descriptores de acceso `get` no son apropiados aquí. En breve hablaremos sobre los parámetros para las operaciones de `BaseDatosBanco`.

Las frases “acredita una cuenta” y “carga a una cuenta” indican que las clases `BaseDatosBanco` y `Cuenta` deben realizar operaciones para actualizar una cuenta durante los depósitos y los retiros, en forma respectiva. Por lo tanto, asignamos las operaciones `Abona` y `Carga` a las clases `BaseDatosBanco` y `Cuenta`. Tal vez recuerde que cuando se abona a una cuenta (como en un depósito) se suma un monto sólo al saldo total de la `Cuenta`. Por otro lado, cuando se carga a una cuenta (como en un retiro) se resta el monto tanto del saldo total como del saldo disponible. Ocultamos estos detalles de implementación dentro de la clase `Cuenta`. Éste es un buen ejemplo de encapsulamiento y ocultamiento de información.

Si éste fuera un sistema ATM real, las clases `BaseDatosBanco` y `Cuenta` también proporcionarían un conjunto de operaciones para permitir que otro sistema bancario actualizara el saldo de la cuenta de un usuario después de confirmar o rechazar todo, o parte de, un depósito. Por ejemplo, la operación `ConfirmarMontoDeposito` sumaría un monto al saldo disponible de la `Cuenta` y haría que los fondos depositados estuvieran disponibles para retiro. La operación `RechazarMontoDeposito` restaría un monto del saldo total de la `Cuenta` para indicar que se invalidó un monto especificado (que se había depositado recientemente a través del ATM y se había sumado al saldo total de la `Cuenta`), o que tal vez los cheques “rebotaron”. El banco invocaría a la operación `RechazarMontoDeposito` después de determinar que el usuario no incluyó el monto correcto de efectivo o que algún cheque no fue validado (es decir, que “rebotó”). Aunque al agregar estas operaciones nuestro sistema estaría más completo, no las incluiremos en nuestros diagramas de clases ni en nuestra implementación, ya que se encuentran más allá del alcance de este caso de estudio.

Operaciones de la clase Pantalla

La clase `Pantalla` “muestra un mensaje al usuario” en diversos momentos durante una sesión con el ATM. Toda la salida visual se produce a través de la pantalla del ATM. El documento de requerimientos describe muchos tipos de mensajes (por ejemplo, un mensaje de bienvenida, uno de error, uno de agradecimiento) que la pantalla muestra al usuario. El documento de requerimientos también indica que la pantalla muestra indicadores y menús al usuario. No obstante, un indicador es en realidad sólo un mensaje que describe lo que el usuario debe introducir a continuación, y un menú es en esencia un tipo de indicador que consiste en una serie de mensajes (es decir, las opciones del menú) que se muestran en forma consecutiva. Por lo tanto, en vez de proveer a la clase `Pantalla` una operación individual para mostrar cada tipo de indicador de mensajes y menú, basta con crear una operación que pueda mostrar cualquier mensaje especificado por un parámetro. Colocamos esta operación (`MostrarMensaje`) en el tercer compartimiento de la clase `Pantalla` en nuestro diagrama de clases (figura 7.20). Observe que no nos preocupa el parámetro de esta operación en estos momentos; lo modelaremos dentro de un momento.

Operaciones de la clase Teclado

De la frase “recibe entrada numérica del usuario” listada por la clase `TecLado` en la figura 7.19, podemos concluir que la clase `TecLado` debe realizar una operación `ObtenerEntrada`. A diferencia del teclado de una computadora, el teclado del ATM sólo contiene los números del 0 al 9, por lo que especificamos que esta operación devuelve un valor entero. Si recuerda, en el documento de requerimientos vimos que en distintas situaciones, tal vez se requiera que el usuario introduzca un tipo distinto de número (por ejemplo, un número de cuenta, un NIP, el número de una opción del menú, un monto de depósito como número de centavos). La clase `TecLado` tan sólo obtiene un valor numérico para un cliente de la clase; no determina si el valor cumple con algún criterio específico.

Cualquier clase que utilice esta operación debe verificar que el usuario haya introducido números apropiados y, en caso contrario, debe mostrar mensajes de error a través de la clase `Pantalla`. [Nota: cuando implementemos el sistema, simularemos el teclado del ATM con el de una computadora y, por cuestión de simpleza, asumiremos que el usuario no escribirá su entrada numérica con teclas en el teclado de la computadora que no aparezcan en el teclado del ATM. En el capítulo 16, Cadenas, caracteres y expresiones regulares, veremos cómo examinar las entradas para determinar si son de ciertos tipos específicos.]

Operaciones de las clases `DispensadorEfectivo` y `RanuraDeposito`

La figura 7.19 lista la frase “dispensa efectivo” para la clase `DispensadorEfectivo`. Por lo tanto, creamos la operación `DispensarEfectivo` y la listamos bajo la clase `DispensadorEfectivo` en la figura 7.20. La clase `DispensadorEfectivo` también “indica si contiene suficiente efectivo para satisfacer una solicitud de retiro”. Para esto incluimos a `HaySuficienteDineroDisponible` (una operación que devuelve un valor de tipo `bool`) en la clase `DispensadorEfectivo`. La figura 7.19 también lista la frase “recibe un sobre de depósito” para la clase `RanuraDeposito`. La ranura de depósito debe indicar si recibió un sobre, por lo que colocamos la operación `SeRecibioSobre`, la cual devuelve un valor `bool`, en el tercer compartimiento de la clase `RanuraDeposito`. [Nota: es muy probable que una ranura de depósito de hardware real envíe una señal al ATM para indicarle que se recibió un sobre. No obstante, simularemos este comportamiento con una operación en la clase `RanuraDeposito` que la clase ATM pueda invocar para averiguar si la ranura de depósito recibió un sobre.]

Operaciones de la clase ATM

No listamos ninguna operación para la clase ATM en este momento. Todavía no sabemos de algún servicio que proporcione la clase ATM a otras clases en el sistema. No obstante, cuando implementemos el sistema en C# (apéndice J, Código del caso de estudio del ATM) tal vez se vuelvan aparentes las operaciones de esta clase junto con las operaciones adicionales de las demás clases en el sistema.

Identificar y modelar los parámetros de operación

Hasta ahora no nos hemos preocupado por los parámetros de nuestras operaciones; sólo hemos tratado de obtener una comprensión básica de las operaciones de cada clase. Ahora vamos a dar un vistazo más de cerca a varios parámetros de operación. Para identificar los parámetros de una operación, analizamos qué datos requiere la operación para realizar su tarea asignada.

Considere la operación `AutenticaUsuario` de la clase `BaseDatosBanco`. Para autenticar a un usuario, esta operación debe conocer el número de cuenta y el NIP que suministra el usuario. Por lo tanto, especificamos que la operación `AutenticaUsuario` debe tomar los parámetros `int usuarioNumeroCuenta` y `nipUsuario`, que la operación debe comparar con el número de cuenta y el NIP de un objeto `Cuenta` en la base de datos. Vamos a colocar después de estos nombres de parámetros la palabra `Usuario`, para evitar confusión entre los nombres de los parámetros de la operación y los nombres de los atributos que pertenecen a la clase `Cuenta`. Listamos estos parámetros en el diagrama de clases de la figura 7.21, el cual modela sólo a la clase `BaseDatosBanco`. [Nota: es perfectamente normal modelar sólo una clase en un diagrama de clases. En este caso lo que más nos preocupa es analizar los parámetros de esta clase específica, por lo que omitimos las demás clases. Más adelante en los diagramas de clase de este caso de estudio, los parámetros dejarán de ser el centro de nuestra atención, por lo que omitiremos los parámetros

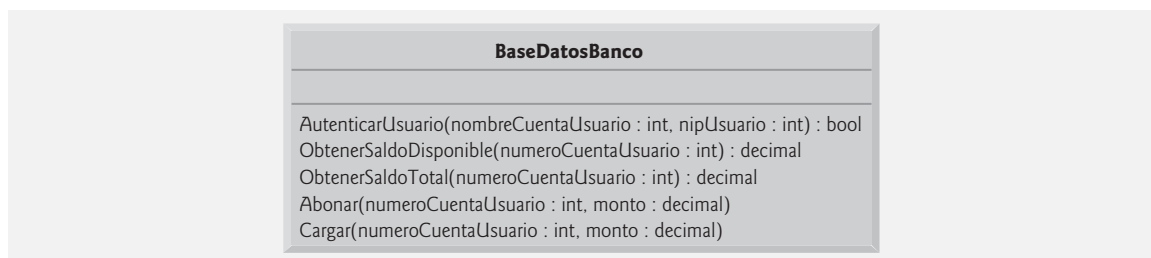


Figura 7.21 | La clase `BaseDatosBanco` con parámetros de operación.

para ahorrar espacio. No obstante, recuerde que las operaciones que se listan en estos diagramas siguen teniendo parámetros.]

Recuerde que para modelar a cada parámetro en una lista de parámetros separados por comas, UML lista el nombre del parámetro, seguido de un signo de punto y coma y el tipo del parámetro. Así, la figura 7.21 especifica, por ejemplo, que la operación `AutenticarUsuario` recibe dos parámetros: `numeroCuentaUsuario` y `nipUsuario`, ambos de tipo `int`.

Las operaciones `ObtenerSaldoDisponible`, `ObtenerSaldoTotal`, `Abonar` y `Cargar` de la clase `BaseDatosBanco` también requieren un parámetro `nombreCuentaUsuario` para identificar la cuenta a la cual la base de datos debe aplicar las operaciones; por lo que incluimos estos parámetros en el diagrama de clases. Además, las operaciones `Abonar` y `Cargar` requieren un parámetro decimal llamado `monto`, para especificar el monto de dinero que se va a abonar o a cargar, respectivamente.

El diagrama de clases de la figura 7.22 modela los parámetros de las operaciones de la clase `Cuenta`. La operación `validarNIP` sólo requiere un parámetro `nipUsuario`, el cual contiene el NIP especificado por el usuario, que se va a comparar con el NIP asociado a la cuenta. Al igual que sus contrapartes en la clase `BaseDatosBanco`, las operaciones `Abonar` y `Cargar` en la clase `Cuenta` requieren un parámetro decimal llamado `monto`, el cual indica la cantidad de dinero involucrada en la operación. Observe que las operaciones de la clase `Cuenta` no requieren un parámetro de número de cuenta; cada una de estas operaciones se puede invocar sólo en el objeto `Cuenta` en el que se ejecutan, por lo que no es necesario incluir un parámetro para especificar una `Cuenta`.

La figura 7.23 modela la clase `Pantalla` con un parámetro especificado para la operación `MostrarMensaje`. Esta operación requiere sólo el parámetro `string` llamado `mensaje`, el cual indica el texto que debe mostrarse en pantalla.

El diagrama de clases de la figura 7.24 especifica que la operación `DispensarEfectivo` de la clase `DispensadorEfectivo` recibe el parámetro decimal llamado `monto` para indicar el monto de efectivo (en

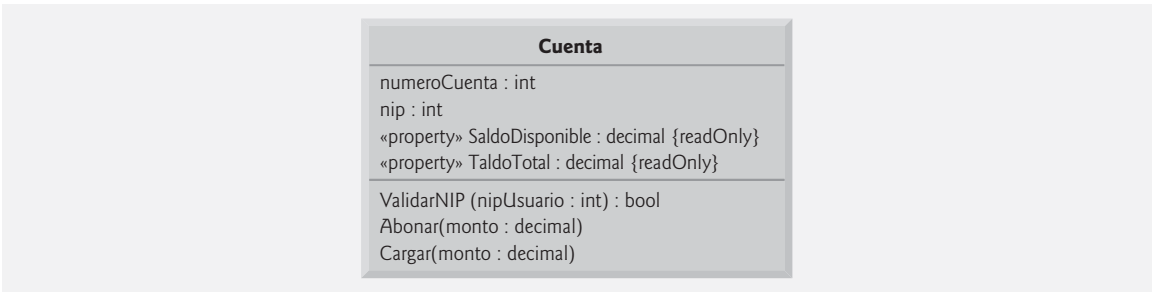


Figura 7.22 | La clase `Cuenta` con parámetros de operación.

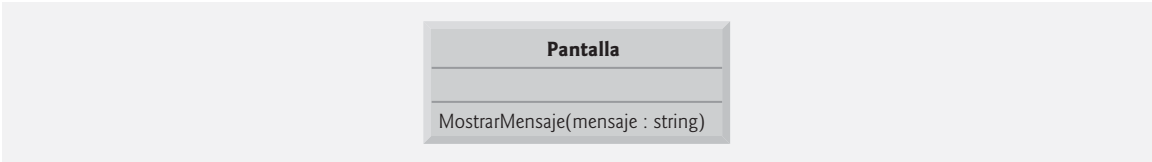


Figura 7.23 | La clase `Pantalla` con parámetros de operación.

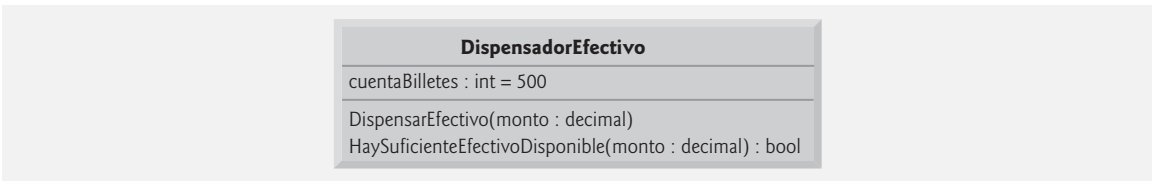


Figura 7.24 | La clase `DispensadorEfectivo` con parámetros de operación.

pesos) que se va a dispensar al usuario. La operación `HaySuficienteEfectivoDisponible` también recibe el parámetro `decimal` llamado `monto` para indicar el monto de efectivo en cuestión.

Observe que no hablamos sobre los parámetros para la operación `Ejecutar` de las clases `SolicitudSaldo`, `Retiro` y `Depósito`, de la operación `ObtenerEntrada` de la clase `Teclado` y la operación `SeRecibioSobre` de la clase `RanuraDeposito`. En este punto de nuestro proceso de diseño, no podemos determinar si estas operaciones requieren datos adicionales para realizar sus tareas, por lo que dejaremos sus listas de parámetros vacías. A medida que avancemos por el caso de estudio, tal vez decidamos agregar parámetros a estas operaciones.

En esta sección hemos determinado muchas de las operaciones que realizan las clases en el sistema ATM. Identificamos los parámetros y los tipos de valores de retorno de algunas operaciones. A medida que continuemos con nuestro proceso de diseño, el número de operaciones que pertenezcan a cada clase puede variar; podríamos descubrir que se necesitan nuevas operaciones o que ciertas operaciones actuales no son necesarias; y podríamos determinar que algunas de las operaciones de nuestras clases necesitan parámetros adicionales y tipos de valores de retorno distintos. De nuevo, todo esto es perfectamente normal.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 7.1** ¿Cuál de las siguientes opciones no es un comportamiento?
- a) leer datos de un archivo
 - b) imprimir los resultados
 - c) imprimir texto
 - d) obtener la entrada del usuario
- 7.2** Si quisiera agregar al sistema ATM una operación que devuelva el atributo `monto` de la clase `Retiro`, ¿cómo y en dónde especificaría esta operación en el diagrama de clases de la figura 7.20?
- 7.3** Describa el significado del siguiente listado de operaciones, el cual podría aparecer en un diagrama de clases para el diseño orientado a objetos de una calculadora:
- ```
Sumar(x : int, y : int) : int
```

### ***Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software***

- 7.1** c.
- 7.2** Una operación que extrae el atributo `monto` de la clase `Retiro` se implementaría, por lo general, como un descriptor de acceso `get` de una propiedad de la clase `Retiro`. Lo siguiente sustituye al atributo `monto` en el compartimiento de atributos (es decir, el segundo) de la clase `Retiro`:
- ```
«property» Monto : decimal
```
- 7.3** Es una operación llamada `Sumar`, la cual recibe los parámetros `int x` y `y`, y devuelve un valor `int`. Lo más probable es que esta operación sume sus parámetros `x` y `y`, y que después devuelva el resultado.

7.16 Conclusión

En este capítulo hablamos sobre la diferencia entre los métodos `static` y los métodos no `static`, y le mostramos cómo llamar a los métodos `static`, anteponiendo al nombre del método el nombre de la clase en la cual aparece, y el operador punto (`.`). También vio que la clase `Math` de la Biblioteca de clases del .NET Framework proporciona muchos métodos `static` para realizar cálculos matemáticos. Presentamos varios espacios de nombres de uso común de la FCL. Aprendió a utilizar el operador `+` para realizar concatenaciones de objetos `string`. Además, aprendió a declarar valores constantes de dos formas: con la palabra clave `const` y los tipos `enum`. Demostramos las técnicas de simulación y utilizamos la clase `Random` para generar conjuntos de números aleatorios. Hablamos sobre el alcance de los campos y las variables locales en una clase. También vio cómo sobrecargar los métodos en una clase al proporcionar métodos con el mismo nombre pero con distintas firmas. Vimos cómo los métodos recursivos se llaman a sí mismos, con lo cual se dividen los problemas más grandes en subproblemas más pequeños hasta que se resuelva el problema original, en un momento dado. Conoció también las diferencias entre los tipos por valor y los tipos por referencia, con respecto a la forma en que se pasan a los métodos, y cómo utilizar las palabras clave `ref` y `out` para pasar argumentos por referencia.

En el capítulo 8 aprenderá a mantener listas y tablas de datos en arreglos. Verá una implementación más elegante de la aplicación que tira un dado 6000 veces, y dos versiones mejoradas de nuestro caso de estudio LibroCalificaciones. También aprenderá cómo acceder a los argumentos de línea de comandos de una aplicación, los cuales se pasan al método `Main` cuando una aplicación de controla comienza su ejecución.

8

Arreglos

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Utilizar arreglos para almacenar datos y recuperarlos de listas y tablas de valores.
- Declarar arreglos, inicializarlos y hacer referencia a elementos individuales de los arreglos.
- Utilizar la instrucción `foreach` para iterar a través de los arreglos.
- Pasar arreglos a los métodos.
- Declarar y manipular arreglos multidimensionales.
- Escribir métodos que utilicen listas de argumentos de longitud variable.
- Leer los argumentos de línea de comandos en una aplicación.

*Ahora ve, escríbelo ante ellos
en una tabla, y anótalo en
un libro.*

—Isaías 30:8

*Ir más allá es tan malo
como no llegar.*

—Confucio

*Comienza en el principio...
y continúa hasta que llegues
al final; después detente.*

—Lewis Carroll

- 8.1 Introducción
- 8.2 Arreglos
- 8.3 Declaración y creación de arreglos
- 8.4 Ejemplos acerca del uso de los arreglos
- 8.5 Caso de estudio: simulación para barajar y repartir cartas
- 8.6 Instrucción `foreach`
- 8.7 Paso de arreglos y elementos de arreglos a los métodos
- 8.8 Paso de arreglos por valor y por referencia
- 8.9 Caso de estudio: la clase `LibroCalificaciones` que usa un arreglo para ordenar calificaciones
- 8.10 Arreglos multidimensionales
- 8.11 Caso de estudio: la clase `LibroCalificaciones` que usa un arreglo rectangular
- 8.12 Listas de argumentos de longitud variable
- 8.13 Uso de argumentos de línea de comandos
- 8.14 (Opcional) Caso de estudio de ingeniería de software: colaboración entre los objetos en el sistema ATM
- 8.15 Conclusión

8.1 Introducción

En este capítulo presentamos el importante tema de las *estructuras de datos*: colecciones de elementos de datos relacionados. Los *arreglos* son estructuras de datos que consisten en elementos de datos del mismo tipo relacionados. Los arreglos son entidades de longitud fija; conservan la misma longitud una vez que se crean, aunque puede reasignarse una variable tipo arreglo de tal forma que haga referencia a un nuevo arreglo de distinta longitud.

Después de hablar acerca de cómo se declaran, crean e inicializan los arreglos, presentaremos una serie de ejemplos que demuestran varias manipulaciones comunes de los arreglos. Además, analizaremos un caso de estudio en el que se utilizan arreglos para simular los procesos de barajar y repartir cartas, para utilizarlos en aplicaciones de juegos de cartas. En este capítulo se demuestra la última instrucción de control estructurada de C#: la instrucción de repetición `foreach`, que ofrece una notación concisa para acceder a los datos en los arreglos (y en otras estructuras de datos, como veremos en el capítulo 26, Colecciones). Hay dos secciones de este capítulo en las que se amplía el caso de estudio `LibroCalificaciones` de los capítulos 4-6. En especial, utilizaremos los arreglos para permitir que la clase mantenga un conjunto de calificaciones en memoria y analizar las calificaciones que obtuvieron los estudiantes en distintos exámenes. Éstos y otros ejemplos demostrarán las formas en las que los arreglos nos permiten organizar y manipular datos.

8.2 Arreglos

Un arreglo es un grupo de variables (llamadas *elementos*) que contienen valores y todos son del mismo tipo. Recuerde que los tipos se dividen en dos categorías: por valor y por referencia. Los arreglos son tipos por referencia. Más adelante veremos que lo que consideramos comúnmente como un arreglo es en realidad una referencia a la instancia de un arreglo en memoria. Los elementos de un arreglo pueden ser tipos por valor o tipos por referencia (incluyendo otros arreglos, como veremos en la sección 8.10). Para referirnos a un elemento en especial en un arreglo, especificamos el nombre de la referencia al arreglo y el número de la posición de ese elemento en el arreglo. Al número de la posición se le conoce como el *índice* del elemento.

La figura 8.1 muestra una representación lógica de un arreglo tipo entero llamado `c`, que contiene 12 elementos. Una aplicación hace referencia a cualquiera de estos elementos mediante una *expresión de acceso a un arreglo*, la cual incluye el nombre del arreglo, seguido del índice del elemento específico entre *corchetes* (`[]`). El primer elemento en cualquier arreglo tiene el *índice cero*, al cual se le conoce como *elemento cero*. Por ende, los elementos del arreglo `c` son `c[0]`, `c[1]`, `c[2]` y así en lo sucesivo. El índice más alto en el arreglo `c` es 11, uno menos que el número de elementos en el arreglo, ya que los índices empiezan desde 0. Los nombres de los arreglos siguen las mismas convenciones que los nombres para las demás variables.

Nombre del arreglo (c) →	c[0]	-45
	c[1]	6
	c[2]	0
	c[3]	72
	c[4]	1543
	c[5]	-89
	c[6]	0
	c[7]	62
	c[8]	-3
	c[9]	1
	c[10]	6453
Índice (o subíndice) del elemento en el arreglo c →	c[11]	78

Figura 8.1 | Un arreglo con 12 elementos.

Un índice debe ser un entero no negativo; también puede ser una expresión. Por ejemplo, si asumimos que la variable *a* es igual a 5 y *b* es igual a 6, entonces la instrucción

```
c[ a + b ] += 2;
```

suma 2 al elemento *c*[11] del arreglo. Observe que el nombre del arreglo con índice es una expresión de acceso al arreglo. Dichas expresiones pueden utilizarse en el lado izquierdo de una asignación para colocar un nuevo valor en un elemento del arreglo. El índice del arreglo debe ser un valor de tipo *int*, *uint*, *long* o *ulong*, o un valor de un tipo que pueda promoverse en forma implícita a uno de estos tipos.

Examinaremos el arreglo *c* de la figura 8.1 de manera más detallada. El **nombre** del arreglo es *c*. Cada instancia de un arreglo conoce su propia longitud y proporciona acceso a esta información a través de la propiedad *Length*. Por ejemplo, la expresión *c.Length* utiliza la propiedad *Length* del arreglo *c* para determinar la longitud del arreglo. Observe que la propiedad *Length* de un arreglo no puede cambiarse, ya que no proporciona un descriptor de acceso *set*. La manera en que se hace referencia a los 12 elementos de este arreglo es: *c*[0], *c*[1], *c*[2], ..., *c*[11]. Sería un error hacer referencia a los elementos fuera de este rango, como *c*[-1] o *c*[12]. El valor de *c*[0] es -45, el de *c*[1] es 6, el de *c*[2] es 0, el de *c*[7] es 62 y el valor de *c*[11] es 78. Para calcular la suma de los valores contenidos en los primeros tres elementos del arreglo *c* y almacenar el resultado en la variable *suma*, escribiríamos lo siguiente:

```
suma = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

Para dividir el valor de *c*[6] entre 2 y asignar el resultado a la variable *x*, escribiríamos lo siguiente:

```
x = c[ 6 ] / 2;
```

8.3 Declaración y creación de arreglos

Las instancias de los arreglos ocupan espacio en memoria. Al igual que los objetos, los arreglos se crean con la palabra clave *new*. Para crear una instancia de un arreglo, se especifica el tipo y el número de elementos del arreglo, y el número de elementos como parte de una **expresión de creación de arreglos** que utiliza la palabra clave *new*. Dicha expresión devuelve una referencia que puede almacenarse en una variable tipo arreglo. La siguiente expresión de declaración y creación de arreglos crea un objeto arreglo que contiene 12 elementos *int*, y almacena la referencia al arreglo en la variable *c*:

```
int[] c = new int[ 12 ];
```

Esta expresión puede usarse para crear el arreglo que se muestra en la figura 8.1 (pero no los valores iniciales del arreglo; en breve le mostraremos cómo inicializar los elementos de un arreglo). Esta tarea también puede llevarse a cabo en dos pasos, como se muestra a continuación:

```
int[] c; // declara la variable tipo arreglo
c = new int[ 12 ]; // crea el arreglo; lo asigna a la variable tipo arreglo
```

En la declaración, los corchetes que van después del tipo de variable `int` indican que `c` es una variable que hará referencia a un arreglo de valores `int` (es decir, `c` almacenará una referencia a un objeto arreglo). En la instrucción de asignación, la variable arreglo `c` recibe la referencia a un nuevo objeto arreglo de 12 elementos `int`. Cuando se crea un arreglo, cada uno de sus elementos recibe un valor predeterminado: 0 para los elementos numéricos de tipo simple, `false` para los elementos `bool` y `null` para las referencias. Como veremos más adelante, a la hora de crear un arreglo podemos proporcionar valores iniciales específicos no predeterminados para los elementos.



Error común de programación 8.1

En la declaración de un arreglo, si se especifica el número de elementos en los corchetes de la declaración (por ejemplo, `int[ 12 ] c;`) se produce un error de sintaxis.

Una aplicación puede crear varios arreglos en una sola declaración. La siguiente declaración reserva 100 elementos para el arreglo `string b` y 27 elementos para `string x`:

```
string[] b = new string[ 100 ], x = new string[ 27 ];
```

En esta declaración se aplica `string[]` a cada variable. Por cuestión de legibilidad, es preferible declarar sólo una variable en cada declaración, como en:

```
string[] b = new string[ 100 ]; // crea el arreglo string b
string[] x = new string[ 27 ]; // crea el arreglo string x
```



Buena práctica de programación 8.1

Por cuestión de legibilidad, declare sólo una variable en cada declaración. Mantenga cada declaración en una línea separada e incluya un comentario que describa a la variable que está declarando.

Una aplicación puede declarar arreglos de elementos de tipo por valor, o elementos de tipo por referencia. Por ejemplo, cada elemento de un arreglo `int` es un valor `int`, y cada elemento de un arreglo `string` es una referencia a un objeto `string`.

8.4 Ejemplos acerca del uso de los arreglos

En esta sección presentaremos varios ejemplos que demuestran cómo declarar arreglos, crearlos, inicializarlos y manipular sus elementos.

Creación e inicialización de un arreglo

La aplicación de la figura 8.2 utiliza la palabra clave `new` para crear un arreglo de 10 elementos `int`, los cuales tienen un valor de 0 al principio (el valor predeterminado para las variables `int`).

La línea 9 declara a `arreglo`: una referencia capaz de hacer referencia a un arreglo de elementos `int`. La línea 12 crea el objeto arreglo de 10 elementos y asigna su referencia a la variable `arreglo`. La línea 14 muestra en pantalla los encabezados de las columnas. La primera columna contiene el índice (0–9) de cada elemento del arreglo; la segunda columna contiene el valor predeterminado (0) de cada elemento del arreglo y tiene una anchura de campo de 8.

La instrucción `for` en las líneas 17–18 muestra en pantalla el número de índice (representado por contador) y el valor (representado por `arreglo[ contador ]`) de cada elemento del arreglo. Observe que la variable de control de ciclo `contador` es 0 al principio; los valores de los índices empiezan en 0, por lo que al utilizar un conteo con base cero se permite al ciclo acceder a todos los elementos del arreglo. La condición de continuación de ciclo de la instrucción `for` utiliza la propiedad `arreglo.Length` (línea 17) para obtener la longitud del arreglo. En

```

1  // Fig. 8.2: InicArreglo.cs
2  // Creación de un arreglo.
3  using System;
4
5  public class InicArreglo
6  {
7      public static void Main( string[] args )
8      {
9          int[] arreglo; // declara un arreglo llamado arreglo
10
11         // crea el espacio para el arreglo y lo inicializa con ceros predeterminados
12         arreglo = new int[ 10 ]; // 10 elementos int
13
14         Console.WriteLine( "{0}{1,8}", "Índice", "Valor" ); // encabezados
15
16         // imprime en pantalla el valor de cada elemento del arreglo
17         for ( int contador = 0; contador < arreglo.Length; contador++ )
18             Console.WriteLine( "{0,5}{1,8}", contador, arreglo[ contador ] );
19     } // fin de Main
20 } // fin de la clase InicArreglo

```

Índice	Valor
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Figura 8.2 | Creación de un arreglo.

este ejemplo la longitud del arreglo es de 10, por lo que el ciclo continúa ejecutándose mientras que el valor de la variable de control contador sea menor que 10. El valor más alto para el índice de un arreglo de 10 elementos es 9, por lo que al utilizar el operador “menor que” en la condición de continuación de ciclo se garantiza que el ciclo no trate de acceder a un elemento más allá del final del arreglo (es decir, durante la iteración final del ciclo, contador es 9). Pronto veremos lo que ocurre cuando se encuentra un índice fuera de rango en tiempo de ejecución.

Uso de un inicializador de arreglos

Una aplicación puede crear un arreglo e inicializar sus elementos con un *inicializador de arreglos*, que es una lista de expresiones separada por comas (conocida como *lista inicializadora*) encerradas entre llaves. En este caso, la longitud del arreglo se determina en base al número de elementos en la lista inicializadora. Por ejemplo, la declaración:

```
int[] n = { 10, 20, 30, 40, 50 };
```

crea un arreglo de cinco elementos con los valores de índices 0, 1, 2, 3 y 4. El elemento `n[ 0 ]` se inicializa con 10, `n[ 1 ]` se inicializa con 20, y así en lo sucesivo. Esta declaración no requiere a `new` para crear el objeto arreglo. Cuando el compilador encuentra la declaración de un arreglo que incluye una lista inicializadora, cuenta el número de inicializadores en la lista para determinar el tamaño del arreglo y después establece la operación `new` apropiada “detrás de las cámaras”.

La aplicación en la figura 8.3 inicializa un arreglo entero con 10 valores (línea 10) y muestra en pantalla el arreglo, en formato tabular. El código para mostrar en pantalla los elementos del arreglo (líneas 15-16) es idéntico al de la figura 8.2 (líneas 17-18).

```

1 // Fig. 8.3: InicArreglo.cs
2 // Inicialización de los elementos de un arreglo, mediante un inicializador de arreglos.
3 using System;
4
5 public class InicArreglo
6 {
7     public static void Main( string[] args )
8     {
9         // la lista inicializadora especifica el valor para cada elemento
10        int[] arreglo = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
11
12        Console.WriteLine( "{0}{1,8}", "Índice", "Valor" ); // encabezados
13
14        // imprime en pantalla el valor de cada elemento del arreglo
15        for ( int contador = 0; contador < arreglo.Length; contador++ )
16            Console.WriteLine( "{0,5}{1,8}", contador, arreglo[ contador ] );
17    } // fin de Main
18 } // fin de la clase InicArreglo

```

Índice	Valor
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

Figura 8.3 | Inicialización de los elementos de un arreglo, mediante un inicializador de arreglos.

Cálculo del valor a guardar en cada elemento del arreglo

Algunas aplicaciones calculan el valor a guardar en cada elemento del arreglo. La aplicación en la figura 8.4 crea un arreglo de 10 elementos y asigna a cada elemento uno de los enteros pares del 2 al 20 (2, 4, 6, ..., 20). Después, la aplicación muestra el arreglo en pantalla, en formato tabular. La instrucción `for` en las líneas 13-14 calcula el valor de un elemento del arreglo mediante la multiplicación del valor actual de la variable de control `contador` del ciclo `for` por 2, y después le suma 2.

La línea 9 utiliza el modificador `const` para declarar la constante `LONGITUD_ARREGLO`, cuyo valor es de 10. Las constantes deben inicializarse al momento de ser declaradas y no pueden modificarse a partir de ese momento. Observe que, por convención, las constantes, al igual que las constantes `enum` que vimos en la sección 7.10, se declaran con letras mayúsculas para hacer que resalten en el código.



Buena práctica de programación 8.2

Las constantes también se conocen como **constantes con nombre**. Con frecuencia, dichas variables mejoran la legibilidad de una aplicación, en comparación con las aplicaciones que utilizan valores literales (por ejemplo, 10); una constante con nombre como `LONGITUD_ARREGLO` indica sin duda su propósito, mientras que un valor literal podría tener distintos significados, con base en el contexto en el que se utiliza. Otra ventaja del uso de constantes con nombre es que, si el valor de la constante debe modificarse, es necesario modificarla sólo en la declaración, con lo cual se reduce el costo de mantener el código.



Error común de programación 8.2

Asignar un valor a una constante con nombre después de inicializarla es un error de compilación.

```

1  // Fig. 8.4: InicArreglo.cs
2  // Cálculo de los valores a colocar en los elementos de un arreglo.
3  using System;
4
5  public class InicArreglo
6  {
7      public static void Main( string[] args )
8      {
9          const int LONGITUD_ARREGLO = 10; // crea una constante con nombre
10         int[] arreglo = new int[ LONGITUD_ARREGLO ]; // crea el arreglo
11
12         // calcula el valor para cada elemento del arreglo
13         for ( int contador = 0; contador < arreglo.Length; contador++ )
14             arreglo[ contador ] = 2 + 2 * contador;
15
16         Console.WriteLine( "{0}{1,8}", "Índice", "Valor" ); // encabezados
17
18         // imprime en pantalla el valor de cada elemento del arreglo
19         for ( int contador = 0; contador < arreglo.Length; contador++ )
20             Console.WriteLine( "{0,5}{1,8}", contador, arreglo[ contador ] );
21     } // fin de Main
22 } // fin de la clase InicArreglo

```

Índice	Valor
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

Figura 8.4 | Cálculo de los valores a colocar en los elementos de un arreglo.



Error común de programación 8.3

Tratar de declarar una constante con nombre sin inicializarla es un error de compilación.

Suma de los elementos de un arreglo

A menudo, los elementos de un arreglo representan una serie de valores a utilizar en un cálculo. Por ejemplo, si los elementos de un arreglo representan las calificaciones de un examen, tal vez un instructor desee totalizar los elementos del arreglo y usar ese total para calcular el promedio de la clase para ese examen. Los ejemplos de LibroCalificaciones que veremos más adelante en este capítulo (figuras 8.15 y 8.20) utilizan esta técnica.

La aplicación en la figura 8.5 suma los valores contenidos en un arreglo entero de 10 elementos. La aplicación declara, crea e inicializa el arreglo en la línea 9. La instrucción `for` realiza los cálculos. [Nota: por lo general, los valores suministrados como inicializadores del arreglo se introducen como entrada en una aplicación, en vez de especificarlos en una lista inicializadora. Por ejemplo, una aplicación podría recibir como entrada los valores de un usuario, o de un archivo en el disco (como veremos en el capítulo 18, Archivos y flujos). Al introducir los datos en una aplicación se aumenta su capacidad de reutilización, ya que puede utilizarse con distintos conjuntos de datos.]

Uso de gráficos de barra para mostrar los datos de un arreglo en forma gráfica

Muchas aplicaciones presentan datos a los usuarios en forma gráfica. Por ejemplo, los valores numéricos con frecuencia se muestran como barras en un gráfico de barras. En dicho gráfico, las barras más largas representan


```

1 // Fig. 8.5: SumaArreglo.cs
2 // Cálculo de la suma de los elementos de un arreglo.
3 using System;
4
5 public class SumaArreglo
6 {
7     public static void Main( string[] args )
8     {
9         int[] arreglo = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10        int total = 0;
11
12        // sumar el valor de cada elemento al total
13        for ( int contador = 0; contador < arreglo.Length; contador++ )
14            total += arreglo[ contador ];
15
16        Console.WriteLine( "Total de los elementos del arreglo: {0}", total );
17    } // fin de Main
18 } // fin de la clase SumaArreglo

```

Total de los elementos del arreglo: 849

Figura 8.5 | Cálculo de la suma de los elementos de un arreglo.

valores numéricos más grandes en forma proporcional. Una manera sencilla de mostrar los datos numéricos en forma gráfica es mediante un gráfico de barras que muestre cada valor numérico en forma de una barra de asteriscos (*).

A los instructores les gusta examinar a menudo la distribución de las calificaciones en un examen. Un instructor podría graficar el número de calificaciones en cada una de varias categorías, para visualizar la distribución de las calificaciones para el examen. Suponga que las calificaciones en un examen fueron 87, 68, 94, 100, 83, 78, 85, 91, 76 y 87. Observe que hubo una calificación de 100, dos calificaciones en el rango de 90 a 99, cuatro calificaciones en el rango de 80 a 89, dos en el rango de 70 a 79, una en el rango de 60 a 69 y ninguna por debajo de 60. Nuestra siguiente aplicación (figura 8.6) almacena estos datos de distribución de las calificaciones en un arreglo de 11 elementos, cada uno de los cuales corresponde a una categoría de calificaciones. Por ejemplo, `arreglo[0]` indica el número de calificaciones en el rango de 0 a 9, `arreglo[7]` indica el número de calificaciones en el rango de 70 a 79 y `arreglo[10]` indica el número de calificaciones de 100. Las dos versiones de la clase `LibroCalificaciones` que veremos más adelante en este capítulo (figuras 8.15 y 8.20) contienen código para calcular estas frecuencias de calificaciones, con base en un conjunto de calificaciones. Por ahora crearemos el arreglo en forma manual, mediante un análisis del conjunto de calificaciones e inicializaremos los elementos de arreglo con el número de valores en cada rango (línea 9).

```

1 // Fig. 8.6: GraficoBarras.cs
2 // Aplicación para imprimir gráficos de barras.
3 using System;
4
5 public class GraficoBarras
6 {
7     public static void Main( string[] args )
8     {
9         int[] arreglo = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
10
11        Console.WriteLine( "Distribución de las calificaciones:" );
12
13        // para cada elemento del arreglo, mostrar en pantalla una barra del gráfico

```

Figura 8.6 | Aplicación para imprimir gráficos de barras. (Parte I de 2).

```

14     for ( int contador = 0; contador < arreglo.Length; contador++ )
15     {
16         // muestra etiquetas de las barras ( "00-09: ", ..., "90-99: ", "100: " )
17         if ( contador == 10 )
18             Console.Write( " 100: " );
19         else
20             Console.Write( "{0:D2}-{1:D2}: ",
21                             contador * 10, contador * 10 + 9 );
22
23         // imprime barra de asteriscos
24         for ( int estrellas = 0; estrellas < arreglo[ contador ]; estrellas++ )
25             Console.Write( "*" );
26
27         Console.WriteLine(); // inicia una nueva línea de salida
28     } // fin de for externo
29 } // fin de Main
30 } // fin de la clase GraficoBarras

```

Distribución de las calificaciones:

```

00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *

```

Figura 8.6 | Aplicación para imprimir gráficos de barras. (Parte 2 de 2).

La aplicación lee los números del arreglo y coloca la información en un gráfico de barras. Cada rango de calificaciones va seguido de una barra de asteriscos que indican el número de calificaciones en ese rango. Para etiquetar cada barra, las líneas 17-21 imprimen un rango de calificaciones (por ejemplo, "70-79 ") con base en el valor actual de contador. Cuando contador es 10, la línea 18 imprime el mensaje " 100: " para alinear el signo de dos puntos con las otras etiquetas de las barras. Cuando contador no es 10, la línea 20 utiliza los elementos de formato {0:D2} y {1:D2} para imprimir la etiqueta del rango de calificaciones. El especificador de formato D indica que el valor se debe formatear como un entero y el número después de la D indica cuántos dígitos debe contener este entero con formato. El 2 indica que los valores con menos de dos dígitos deben empezar con un 0 a la izquierda.

La instrucción for anidada (líneas 24-25) imprime las barras en pantalla. Observe la condición de continuación de ciclo en la línea 24 (`estrellas < arreglo[ contador ]`). Cada vez que la aplicación llega al for interno, el ciclo cuenta desde 0 hasta un valor menos que el del `arreglo[ contador ]`, con lo cual utiliza un valor en `arreglo` para determinar el número de asteriscos a mostrar en pantalla. En este ejemplo, los valores de `arreglo[ 0 ]` hasta `arreglo[ 5 ]` son 0, ya que ningún estudiante recibió una calificación menor de 60. Por lo tanto, la aplicación no muestra asteriscos enseguida de los primeros seis rangos de calificaciones.

Uso de los elementos de un arreglo como contadores

En ocasiones, las aplicaciones utilizan variables tipo contador para sintetizar datos, como los resultados de una encuesta. En la figura 7.8 utilizamos contadores separados en nuestra aplicación para tirar dados, para rastrear el número de veces que aparecía cada una de las caras de un dado con seis lados al tiempo que la aplicación tiraba el dado 6000 veces. En la figura 8.7 se muestra una versión de la aplicación de la figura 7.8, esta vez usando un arreglo.

```

1  // Fig. 8.7: TirarDado.cs
2  // Tirar 6000 veces un dado de seis lados.
3  using System;
4
5  public class TirarDado
6  {
7      public static void Main( string[] args )
8      {
9          Random numerosAleatorios = new Random(); // generador de números aleatorios
10         int[] frecuencia = new int[ 7 ]; // arreglo de contadores de frecuencia
11
12         // tira el dado 6000 veces; usa el valor del dado como subíndice de frecuencia
13         for ( int tiro = 1; tiro <= 6000; tiro++ )
14             ++frecuencia[ numerosAleatorios.Next( 1, 7 ) ];
15
16         Console.WriteLine( "{0}{1,10}", "Cara", "Frecuencia" );
17
18         // imprime en pantalla el valor de cada elemento del arreglo
19         for ( int cara = 1; cara < frecuencia.Length; cara++ )
20             Console.WriteLine( "{0,4}{1,10}", cara, frecuencia[ cara ] );
21     } // fin de Main
22 } // fin de la clase TirarDado

```

Cara	Frecuencia
1	959
2	1032
3	986
4	1016
5	987
6	1020

Figura 8.7 | Tirar 6000 veces un dado de seis lados.

La figura 8.7 utiliza el arreglo `frecuencia` (línea 10) para contar las ocurrencias de cada lado del dado. La instrucción individual en la línea 14 de esta aplicación sustituye a las líneas 23-46 de la figura 7.8. La línea 14 utiliza el valor aleatorio para determinar qué elemento de `frecuencia` debe incrementar durante cada iteración del ciclo. El cálculo en la línea 14 produce números aleatorios del 1 al 6, por lo que el arreglo `frecuencia` debe ser lo bastante grande como para poder almacenar seis contadores. Utilizamos un arreglo de siete elementos, en el cual ignoramos `frecuencia[ 0 ]`; es más lógico que el valor de cara 1 incremente a `frecuencia[ 1 ]` que a `frecuencia[ 0 ]`. Por ende, cada valor de cara se utiliza como índice para el arreglo `frecuencia`. También sustituimos las líneas 50-52 de la figura 7.8 por un ciclo a través del arreglo `frecuencia` para imprimir los resultados en pantalla (figura 8.7, líneas 19-20).

Uso de arreglos para analizar los resultados de una encuesta

Nuestro siguiente ejemplo utiliza arreglos para sintetizar los resultados de los datos recolectados en una encuesta:

Se pidió a 40 estudiantes que calificaran, en una escala del 1 al 10 (en donde 1 significa pésimo y 10 significa excelente), la calidad de la comida en la cafetería estudiantil. Coloque las 40 respuestas en un arreglo entero y sintetice los resultados de la encuesta.

Ésta es una típica aplicación de procesamiento de arreglos (vea la figura 8.8). Deseamos resumir el número de respuestas de cada tipo (es decir, del 1 al 10). El arreglo `respuestas` (líneas 10-12) es un arreglo `int` de 40 elementos, y contiene las respuestas de los estudiantes. Utilizamos un arreglo de 11 elementos llamado `frecuencia` (línea 13) para contar el número de ocurrencias de cada respuesta. Cada elemento del arreglo se utiliza como un contador para una de las respuestas, y se inicializa con 0 de manera predeterminada. Al igual que en la figura 8.7, ignoramos el primer elemento (`frecuencia[ 0 ]`).

```

1  // Fig. 8.8: EncuestaEstudiantil.cs
2  // Aplicación para analizar encuestas.
3  using System;
4
5  public class EncuestaEstudiantil
6  {
7      public static void Main( string[] args )
8      {
9          // arreglo de respuestas a la encuesta
10         int[] respuestas = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
11                             10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
12                             4, 8, 6, 8, 10 };
13         int[] frecuencia = new int[ 11 ]; // arreglo de contadores de frecuencia
14
15         // para cada respuesta, selecciona el elemento de respuestas y usa ese valor
16         // como subíndice de frecuencia para determinar el elemento a incrementar
17         for ( int respuesta = 0; respuesta < respuestas.Length; respuesta++ )
18             ++frecuencia[ respuestas[ respuesta ] ];
19
20         Console.WriteLine( "{0}{1,11}", "Calificación", "Frecuencia" );
21
22         // imprime en pantalla el valor de cada elemento del arreglo
23         for ( int calificacion = 1; calificacion < frecuencia.Length; calificacion++ )
24             Console.WriteLine( "{0,12}{1,11}", calificacion, frecuencia[ calificacion ] );
25     } // fin de Main
26 } // fin de la clase EncuestaEstudiantil

```

Calificación	Frecuencia
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

Figura 8.8 | Aplicación para analizar encuestas.

El ciclo `for` en las líneas 17-18 recibe las respuestas del arreglo `respuestas` una a la vez, e incrementa uno de los 10 contadores en el arreglo `frecuencia` (de `frecuencia[ 1 ]` a `frecuencia[ 10 ]`). La instrucción clave en el ciclo es la línea 18, la cual incrementa el contador de frecuencia apropiado, dependiendo del valor de `respuestas[ respuesta ]`.

Consideraremos varias iteraciones del ciclo `for`. Cuando la variable de control `respuesta` es 0, el valor de `respuestas[ respuesta ]` es el valor de `respuestas[ 0 ]` (es decir, 1 en la línea 10), por lo que la aplicación interpreta a `++frecuencia[ respuestas[ respuesta ] ]` como

```
++frecuencia[ 1 ]
```

con lo cual se incrementa el valor en el primer elemento del arreglo `frecuencia`. Para evaluar la expresión, empiece con el valor en el conjunto más interno de corchetes: `respuesta`. Una vez que conozca el valor de `respuesta` (que es el valor de la variable de control de ciclo en la línea 17), insértelo en la expresión y evalúe el siguiente conjunto más externo de corchetes: `respuestas[ respuesta ]`, que es un valor seleccionado del arreglo `respuestas` en las líneas 10-12. Después utilice el valor resultante como índice del arreglo `frecuencia` para especificar cuál contador se incrementará (línea 18).

Cuando `respuesta` es 1, `respuestas[ respuesta ]` es el valor de `respuestas[ 1 ]`, que es 2, por lo que la aplicación interpreta a `++frecuencia[ respuestas[ respuesta ] ]` como

```
++frecuencia[ 2 ]
```

con lo cual se incrementa el segundo elemento del arreglo `frecuencia`.

Cuando `respuesta` es 2, `respuestas[ respuesta ]` es el valor de `respuestas[ 2 ]`, que es 6, por lo que la aplicación interpreta a `++frecuencia[ respuestas[ respuesta ] ]` como

```
++frecuencia[ 6 ]
```

con lo cual se incrementa el sexto elemento del arreglo `frecuencia`, y así en lo sucesivo. Sin importar el número de respuestas procesadas en la encuesta, la aplicación sólo requiere un arreglo de 11 elementos (en el cual se ignora el elemento 0) para resumir los resultados, ya que todos los valores de las respuestas se encuentran entre 1 y 10, inclusive, y los valores de índice para un arreglo de 11 elementos son del 0 al 10.

Si los datos en el arreglo `respuestas` tuvieran valores inválidos como 13, la aplicación trataría de sumar 1 a `frecuencia[ 13 ]`, lo cual se encuentra fuera de los límites del arreglo. En muchos lenguajes de programación como C y C++, se permite escribir fuera de los límites de un arreglo, pero se corre el riesgo de sobrescribir información en la memoria, lo que a menudo produce resultados desastrosos. C# no permite esto; el acceso a cualquier elemento de un arreglo obliga a realizar una comprobación en el índice del arreglo, para asegurar que sea válido (es decir, debe ser mayor o igual a 0 y menor que la longitud del arreglo). A esto se le conoce como *comprobación de límites*. Si una aplicación utiliza un índice inválido, el entorno en tiempo de ejecución (Common Language Runtime) genera una excepción (específicamente, una ***IndexOutOfRangeException***) para indicar que se produjo un error en la aplicación en tiempo de ejecución. La condición en una instrucción de control podría determinar si un índice es válido antes de permitir que se utilice en una *expresión de acceso a un arreglo*, con lo que se evitaría la excepción.



Tip de prevención de errores 8.1

Una excepción indica que ocurrió un error en una aplicación. A menudo usted puede escribir código para recuperarse de una excepción y continuar ejecutando de la aplicación, en vez de terminarla en forma anormal. En el capítulo 12 hablaremos sobre el manejo de excepciones.



Tip de prevención de errores 8.2

Al escribir código para iterar a través de un arreglo, es necesario asegurar que el índice sea mayor o igual a 0 y menor que la longitud del arreglo. La condición de continuación de ciclo debe evitar el acceso a elementos fuera de este rango.

8.5 Caso de estudio: simulación para barajar y repartir cartas

Hasta ahora, en los ejemplos en este capítulo hemos utilizado arreglos que contienen elementos de tipo por valor. En esta sección utilizaremos la generación de números aleatorios y un arreglo de elementos de tipo por referencia (a saber, objetos que representan cartas de juego) para desarrollar una clase que simule los procesos de barajar y repartir cartas. Después podremos utilizar esta clase para implementar aplicaciones que jueguen cartas.

Primero desarrollaremos la clase `Carta` (figura 8.9), la cual representa una carta de juego que tiene una cara ("As", "Dos", "Tres", ... "Joto", "Quina", "Rey") y un palo ("Corazones", "Diamantes", "Tréboles", "Espadas"). Después desarrollaremos la clase `PaqueteDeCartas` (figura 8.10), que crea un paquete de 52 cartas en las que cada elemento es un objeto `Carta`. Luego construiremos una aplicación de prueba (figura 8.11) para demostrar las capacidades de la clase `PaqueteDeCartas` para barajar y repartir cartas.

La clase `Carta`

La clase `Carta` (figura 8.9) contiene dos variables de instancia `string` (`cara` y `palo`) que se utilizan para almacenar referencias al valor de la cara y al valor del palo para una `Carta` específica. El constructor de la clase (líneas 9-13) recibe dos objetos `string` que utiliza para inicializar `cara` y `palo`. El método `ToString` (líneas 16-19) crea un objeto `string` que consiste en la cara de la carta, la cadena "de" y el palo de la carta. En el capítulo 7 vimos que el operador `+` puede utilizarse para concatenar (es decir, combinar) varios objetos `string` para formar uno

```

1 // Fig. 8.9: Carta.cs
2 // La clase Carta representa una carta de juego.
3 public class Carta
4 {
5     private string cara; // cara de la carta ("As", "Dos", ...)
6     private string palo; // palo de la carta ("Corazones", "Diamantes", ...)
7
8     // el constructor con dos parámetros inicializa la cara y el palo de la carta
9     public Carta( string caraCarta, string paloCarta )
10    {
11        cara = caraCarta; // inicializa la cara de la carta
12        palo = paloCarta; // inicializa el palo de la carta
13    } // fin del constructor de Carta con dos parámetros
14
15    // devuelve representación de cadena del objeto Carta
16    public override string ToString()
17    {
18        return cara + " de " + palo;
19    } // fin del método ToString
20 } // fin de la clase Carta

```

Figura 8.9 | La clase Carta representa una carta de juego.

más grande. El método ToString de Carta puede invocarse en forma explícita para obtener la representación de cadena de un objeto Carta (por ejemplo, "As de Espadas"). El método ToString de un objeto se llama en forma implícita en muchos casos en los que el objeto se utiliza en donde se espera un objeto string (por ejemplo, cuando WriteLine imprime en pantalla el objeto con un elemento de formato, o cuando el objeto se concatena con un objeto string mediante el operador +). Para que ocurra este comportamiento, ToString debe declararse con el mismo encabezado exacto que se muestra en la línea 16 de la figura 8.9. Explicaremos de manera detallada el propósito de la palabra clave override cuando hablemos sobre la herencia en el capítulo 10.

```

1 // Fig. 8.10: PaqueteDeCartas.cs
2 // La clase PaqueteDeCartas representa un paquete de cartas de juego.
3 using System;
4
5 public class PaqueteDeCartas
6 {
7     private Carta[] paquete; // arreglo de objetos Carta
8     private int cartaActual; // subíndice de la siguiente Carta a repartir
9     private const int NUMERO_DE_CARTAS = 52; // constante para el número de objetos Carta
10    private Random numerosAleatorios; // generador de números aleatorios
11
12    // el constructor llena el paquete de objetos Carta
13    public PaqueteDeCartas()
14    {
15        string[] caras = { "As", "Dos", "Tres", "Cuatro", "Cinco", "Seis",
16                           "Siete", "Ocho", "Nueve", "Diez", "Joto", "Quina", "Rey" };
17        string[] palos = { "Corazones", "Diamantes", "Tréboles", "Espadas" };
18
19        paquete = new Carta[ NUMERO_DE_CARTAS ]; // crea un arreglo de objetos Carta
20        cartaActual = 0; // establece cartaActual para que la primera Carta repartida sea
                       paquete[ 0 ]

```

Figura 8.10 | La clase PaqueteDeCartas representa un paquete de cartas de juego. (Parte I de 2).

```

21     numerosAleatorios = new Random(); // crea el generador de números aleatorios
22
23     // llena el paquete con objetos Carta
24     for ( int cuenta = 0; cuenta < paquete.Length; cuenta++ )
25         paquete[ cuenta ] =
26             new Carta( caras[ cuenta % 13 ], palos[ cuenta / 13 ] );
27 } // fin del constructor PaqueteDeCartas
28
29 // baraja el paquete de objetos Carta con un algoritmo de una pasada
30 public void Barajar()
31 {
32     // después de barajar, la repartición debe empezar otra vez en paquete[ 0 ]
33     cartaActual = 0; // reinicializa cartaActual
34
35     // para cada Carta, selecciona otra Carta aleatoria y las intercambia
36     for ( int primera = 0; primera < paquete.Length; primera++ )
37     {
38         // selecciona un número aleatorio entre 0 y 51
39         int segunda = numerosAleatorios.Next( NUMERO_DE_CARTAS );
40
41         // intercambia la Carta actual con la Carta seleccionada al azar
42         Carta temp = paquete[ primera ];
43         paquete[ primera ] = paquete[ segunda ];
44         paquete[ segunda ] = temp;
45     } // fin de for
46 } // fin del método Barajar
47
48 // reparte una Carta
49 public Carta RepartirCarta()
50 {
51     // determina si hay objetos Carta por repartir
52     if ( cartaActual < paquete.Length )
53         return paquete[ cartaActual++ ]; // devuelve la Carta actual en el arreglo
54     else
55         return null; // devuelve null para indicar que se repartieron todos los
                    // objetos Carta
56 } // fin del método RepartirCarta
57 } // fin de la clase PaqueteDeCartas

```

Figura 8.10 | La clase PaqueteDeCartas representa un paquete de cartas de juego. (Parte 2 de 2).

Clase PaqueteDeCartas

La clase PaqueteDeCartas (figura 8.10) declara un arreglo de variables de instancia llamado `paquete`, el cual contiene objetos Carta (línea 7). Al igual que las declaraciones de arreglos de tipos simples, la declaración de un arreglo de objetos incluye el tipo de elementos en el arreglo, seguido de corchetes y del nombre de la variable del arreglo (por ejemplo `Carta[] paquete`). La clase PaqueteDeCartas también declara la variable de instancia `int` llamada `cartaActual` (línea 8), que representa la siguiente Carta a repartir del arreglo `paquete`, y la constante con nombre `NUMERO_DE_CARTAS` (línea 9) indica el número de objetos Carta en el paquete (52).

El constructor de la clase crea una instancia del arreglo `paquete` (línea 19) con un tamaño igual a `NUMERO_DE_CARTAS`. Cuando se crea por primera vez el arreglo `paquete`, sus elementos son `null` de manera predeterminada, por lo que el constructor utiliza una instrucción `for` (líneas 24-26) para llenar el arreglo `paquetes` con objetos Carta. La instrucción `for` inicializa la variable de control `cuenta` con 0 e itera mientras `cuenta` sea menor que `paquete.Length`, lo cual hace que `cuenta` tome el valor de cada entero del 0 al 51 (los índices del arreglo `paquete`). Cada objeto Carta se instancia y se inicializa con dos objetos `string`: uno del arreglo `caras` (que contiene los objetos `string` del "As" hasta el "Rey") y uno del arreglo `palos` (que contiene los objetos `string` "Corazones", "Diamantes", "Tréboles" y "Espadas"). El cálculo `cuenta % 13` siempre produce un

valor de 0 a 12 (los 13 índices del arreglo caras en las líneas 15-16), y el cálculo `cuenta / 13` siempre produce un valor de 0 a 3 (los cuatro índices del arreglo palos en la línea 17). Cuando se inicializa el arreglo paquete, contiene los objetos Carta con las caras del "As" al "Rey" en orden para cada palo.

El método `Barajar` (líneas 30-46) baraja los objetos Carta en el paquete. El método itera a través de los 52 objetos Carta (índices 0 a 51 del arreglo). Para cada objeto Carta se elige al azar un número entre 0 y 51 para elegir otro objeto Carta. A continuación, el objeto Carta actual y el objeto Carta seleccionado al azar se intercambian en el arreglo. Este intercambio se realiza mediante las tres asignaciones en las líneas 42-44. La variable extra `temp` almacena en forma temporal uno de los dos objetos Carta que se van a intercambiar. El intercambio no se puede realizar sólo con las dos instrucciones

```
paquete[ primera ] = paquete[ segunda ];
paquete[ segunda ] = paquete[ primera ];
```

Si `paquete[ primera ]` es el "As" de "Espadas" y `paquete[ segunda ]` es la "Quina" de "Corazones", entonces después de la primera asignación, ambos elementos contienen la "Reina" de "Corazones" y se pierde el "As" de "Espadas"; es por ello que se necesita la variable extra `temp`. Una vez que termina el ciclo `for`, los objetos Carta se ordenan al azar. Sólo se realizan 52 intercambios en una sola pasada del arreglo completo, y el arreglo de objetos Carta se baraja.

El método `RepartirCarta` (líneas 49-56) reparte un objeto Carta en el arreglo. Recuerde que `cartaActual` indica el índice del siguiente objeto Carta que se repartirá (es decir, la Carta en la parte superior del paquete). Por ende, la línea 52 compara `cartaActual` con la longitud del arreglo paquete. Si el paquete no está vacío (es decir, si `cartaActual` es menor a 52), la línea 53 regresa el objeto Carta superior e incrementa `cartaActual` para prepararse para la siguiente llamada a `RepartirCarta`; en caso contrario, se devuelve `null`.

Barajar y repartir cartas

La aplicación de la figura 8.11 demuestra las capacidades de barajar y repartir cartas de la clase `PaqueteDeCartas` (figura 8.10). La línea 10 crea un objeto `PaqueteDeCartas` llamado `miPaqueteDeCartas`. Recuerde que el constructor `PaqueteDeCartas` crea el paquete con los 52 objetos Carta, en orden por palo y por cara. La línea 11 invoca el método `Barajar` de `miPaqueteDeCartas` para reordenar los objetos Carta. La instrucción `for` en las líneas 14-20 reparte los 52 objetos Carta en el paquete y los imprime en cuatro columnas, cada una con 13 objetos Carta. Las líneas 17-19 reparten e imprimen en pantalla cuatro objetos Carta, cada uno de ellos se obtiene mediante la invocación al método `RepartirCarta` de `miPaqueteDeCartas`. Cuando `WriteLine` imprime en pantalla un objeto Carta con formato `string`, el método `ToString` de Carta (declarado en las líneas 16-19 de la figura 8.9) se invoca en forma implícita. Como la anchura de campo es negativa, el resultado se justifica a la izquierda en una anchura de campo de 20.

```
1 // Fig. 8.11: PruebaPaqueteDeCartas.cs
2 // Aplicación para barajar y repartir cartas.
3 using System;
4
5 public class PruebaPaqueteDeCartas
6 {
7     // ejecuta la aplicación
8     public static void Main( string[] args )
9     {
10         PaqueteDeCartas miPaqueteDeCartas = new PaqueteDeCartas();
11         miPaqueteDeCartas.Barajar(); // coloca las cartas en orden aleatorio
12
13         // imprime las 52 cartas en el orden en el que se reparten
14         for ( int i = 0; i < 13; i++ )
15         {
16             // reparte e imprime 4 objetos Carta
```

Figura 8.11 | Aplicación para barajar y repartir cartas. (Parte I de 2).


```

17      Console.WriteLine( "{0,-20}{1,-20}{2,-20}{3,-20}",
18      miPaqueteDeCartas.RepartirCarta(), miPaqueteDeCartas.RepartirCarta(),
19      miPaqueteDeCartas.RepartirCarta(), miPaqueteDeCartas.RepartirCarta() );
20    } // fin de for
21  } // fin de Main
22 } // fin de la clase PruebaPaqueteDeCartas

```

Ocho de Diamantes	Qüina de Tréboles	Diez de Corazones	Diez de Espadas
Joto de Tréboles	Seis de Tréboles	Ocho de Corazones	Seis de Diamantes
Dos de Espadas	Cinco de Tréboles	As de Diamantes	Siete de Tréboles
Dos de Corazones	Tres de Diamantes	Siete de Espadas	Cinco de Diamantes
Joto de Espadas	Siete de Diamantes	Joto de Corazones	Tres de Tréboles
Dos de Diamantes	Ocho de Espadas	Qüina de Diamantes	Qüina de Corazones
Rey de Espadas	Tres de Corazones	Diez de Diamantes	As de Espadas
Ocho de Tréboles	Cuatro de Tréboles	Siete de Corazones	Cuatro de Espadas
Cuatro de Diamantes	Rey de Tréboles	Nueve de Tréboles	Nueve de Diamantes
Cinco de Espadas	Cuatro de Corazones	As de Corazones	Joto de Diamantes
Qüina de Espadas	Rey de Diamantes	Tres de Espadas	Rey de Corazones
Cinco de Corazones	Dos de Tréboles	Seis de Corazones	Diez de Tréboles
Nueve de Corazones	Nueve de Espadas	Seis de Espadas	As de Tréboles

Figura 8.11 | Aplicación para barajar y repartir cartas. (Parte 2 de 2).

8.6 Instrucción foreach

En ejemplos anteriores, demostramos cómo utilizar las instrucciones `for` controladas por un contador para iterar a través de los elementos en un arreglo. En esta sección presentaremos la **instrucción `foreach`**, que itera a través de los elementos de un arreglo o colección completa. Esta sección habla acerca de cómo utilizar la instrucción `foreach` para iterar a través de un arreglo. En el capítulo 26, Colecciones, veremos cómo utilizar la instrucción `foreach` con colecciones. La sintaxis de una instrucción `foreach` es:

```
foreach ( tipo identificador in nombreArreglo )
    instrucción
```

en donde *tipo* e *identificador* son el tipo y el nombre (por ejemplo, `int numero`) de la **variable de iteración**, y *nombreArreglo* es el arreglo a través del cual se va a iterar. El tipo de la variable de iteración debe concordar con el tipo de los elementos en el arreglo. Como se muestra en el siguiente ejemplo, la variable de iteración representa valores sucesivos en el arreglo, en iteraciones sucesivas de la instrucción `foreach`.

La figura 8.12 utiliza la instrucción `foreach` (líneas 13-14) para calcular la suma de los enteros en un arreglo de calificaciones de estudiantes. El tipo especificado es `int`, ya que `arreglo` contiene valores `int`; por lo tanto, el ciclo seleccionará un valor `int` del arreglo durante cada iteración. La instrucción `foreach` itera a través de valores sucesivos en el arreglo, uno por uno. El encabezado `foreach` se puede leer de manera concisa como “para cada iteración, asignar el siguiente elemento de `arreglo` a la variable `int numero`, después ejecutar la siguiente instrucción”. Por lo tanto, para cada iteración, el identificador `numero` representa el siguiente valor `int`. Las líneas 13-14 son equivalentes a la siguiente repetición controlada por un contador que se utiliza en las líneas 13-14 de la figura 8.5, para totalizar los enteros en el arreglo:

```
for ( int contador = 0; contador < arreglo.Length; contador++ )
    total += arreglo[ contador ];
```

La instrucción `foreach` simplifica el código para iterar a través de un arreglo. No obstante, observe que la instrucción `foreach` sólo puede utilizarse para acceder a los elementos del arreglo; y no para modificar los elementos. Cualquier intento por modificar el valor de la variable de iteración en el cuerpo de una instrucción `foreach` producirá un error de compilación. Si su aplicación necesita modificar elementos, use la instrucción `for`.

La instrucción `foreach` se puede utilizar en lugar de la instrucción `for` cuando el código que itera a través de un arreglo no requiere acceso al contador que indica el índice del elemento actual del arreglo. Por ejemplo, para totalizar los enteros en un arreglo se requiere acceso sólo a los valores de los elementos; el índice de cada elemento

```

1  // Fig. 8.12: PruebaForEach.cs
2  // Uso de la instrucción foreach para totalizar los enteros en un arreglo.
3  using System;
4
5  public class PruebaForEach
6  {
7      public static void Main( string[] args )
8      {
9          int[] arreglo = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10         int total = 0;
11
12         // suma el valor de cada elemento a total
13         foreach ( int numero in arreglo )
14             total += numero;
15
16         Console.WriteLine( "Total de elementos en el arreglo: {0}", total );
17     } // fin de Main
18 } // fin de la clase PruebaForEach

```

Total de elementos en el arreglo: 849

Figura 8.12 | Uso de la instrucción foreach para totalizar los enteros en un arreglo.

es irrelevante. No obstante, si una aplicación debe utilizar un contador por alguna razón que no sea tan sólo iterar a través de un arreglo (por ejemplo, imprimir un número de índice al lado del valor de cada elemento del arreglo, como en los primeros ejemplos en este capítulo), use la instrucción for.

8.7 Paso de arreglos y elementos de arreglos a los métodos

Para pasar un argumento tipo arreglo a un método, se especifica el nombre del arreglo sin corchetes. Por ejemplo, si el arreglo `temperaturasPorHora` se declara como

```
double[] temperaturasPorHora = new double[ 24 ];
```

entonces la llamada al método

```
ModificarArreglo( temperaturasPorHora );
```

pasa la referencia del arreglo `temperaturasPorHora` al método `ModificarArreglo`. Todo objeto arreglo “conoce” su propia longitud (que está disponible a través de su propiedad `Length`). Por ende, cuando pasamos a un método la referencia a un objeto arreglo, no necesitamos pasar la longitud del arreglo como un argumento adicional.

Para que un método reciba una referencia a un arreglo a través de una llamada a ese método, la lista de parámetros del método debe especificar un parámetro tipo arreglo. Por ejemplo, el encabezado para el método `ModificarArreglo` podría escribirse así:

```
void ModificarArreglo( double[] b )
```

lo cual indica que `ModificarArreglo` recibe la referencia de un arreglo de elementos `double` en el parámetro `b`. La llamada a este método pasa la referencia al arreglo `temperaturaPorHoras`, de manera que cuando el método llamado utiliza la variable `b` tipo arreglo, hace referencia al mismo objeto arreglo como `temperaturasPorHora` en el método que hizo la llamada.

Cuando un argumento para un método es todo un arreglo (o un elemento individual de un arreglo) de un tipo por referencia, el método llamado recibe una copia de la referencia. Sin embargo, cuando un argumento para un método es un elemento individual de un arreglo de tipo por valor, el método llamado recibe una copia del valor del elemento. Para pasar un elemento individual de un arreglo a un método, use el nombre indexado del arreglo como argumento en la llamada al método. Si desea pasar un elemento de un arreglo de tipo por valor

a un método por referencia, debe utilizar la palabra clave `ref`, como se indica en la sección 7.14, Paso de argumentos: diferencia entre paso por valor y paso por referencia.

La figura 8.13 demuestra la diferencia entre pasar a un método todo un arreglo y pasar un elemento de un arreglo de tipo por valor. La instrucción `foreach` en las líneas 17-18 imprime en pantalla los cinco elementos de `arreglo` (un arreglo de valores `int`). La línea 20 invoca al método `ModificarArreglo`, pasando `arreglo` como argumento. El método `ModificarArreglo` (líneas 37-41) recibe una copia de la referencia a `arreglo` y utiliza esta referencia para multiplicar cada uno de los elementos de `arreglo` por 2. Para demostrar que se modificaron los elementos de `arreglo` (en `Main`), la instrucción `foreach` en las líneas 24-25 imprime, en pantalla, nuevamente los cinco elementos de `arreglo`. Como se muestra en la salida, el método `ModificarArreglo` duplicó el valor de cada elemento.

La figura 8.13 demuestra que, cuando se pasa una copia de un elemento individual de un arreglo de tipo por valor a un método, si se modifica la copia en el método que se llamó el valor original de ese elemento no se ve afectado en el arreglo del método que hizo la llamada. Para mostrar el valor de `arreglo[3]` antes de invocar al método `ModificarElemento`, las líneas 27-29 imprimen en pantalla el valor de `arreglo[3]`, que es 8. La línea 31 llama al método `ModificarElemento` y le pasa `arreglo[3]` como argumento. Recuerde que `arreglo[3]` es en realidad un valor `int` (8) en `arreglo`. Por lo tanto, la aplicación pasa una copia del valor de `arreglo[3]`. El método `ModificarElemento` (líneas 44-49) multiplica por 2 el valor recibido como argumento, almacena el resultado en su parámetro `elemento` e imprime en pantalla el valor de `elemento` (16). Como los parámetros de los métodos, al igual que las variables locales, dejan de existir cuando el método en el que se declaran termina su ejecución, el parámetro `elemento` del método se destruye cuando `ModificarElemento` termina. Por lo tanto, cuando la aplicación devuelve el control a `Main`, las líneas 32-33 imprimen en pantalla el valor de `arreglo[3]` que no se modificó (es decir, 8).

```

1  // Fig. 8.13: PasoArreglo.cs
2  // Paso de arreglos y elementos individuales de arreglos a los métodos.
3  using System;
4
5  public class PasoArreglo
6  {
7      // Main crea arreglo y llama a ModificarArreglo y ModificarElemento
8      public static void Main( string[] args )
9      {
10         int[] arreglo = { 1, 2, 3, 4, 5 };
11
12         Console.WriteLine(
13             "Efectos de pasar una referencia a todo un arreglo:\n" +
14             "Los valores del arreglo original son:" );
15
16         // imprime en pantalla elementos del arreglo original
17         foreach ( int valor in arreglo )
18             Console.Write( " {0}", valor );
19
20         ModificarArreglo( arreglo ); // pasa referencia al arreglo
21         Console.WriteLine( "\n\nLos valores del arreglo modificado son:" );
22
23         // imprime en pantalla elementos modificados del arreglo
24         foreach ( int valor in arreglo )
25             Console.Write( " {0}", valor );
26
27         Console.WriteLine(
28             "\n\nEfectos de pasar el valor de un elemento del arreglo:\n" +
29             "arreglo[3] antes de ModificarElemento: {0}", arreglo[ 3 ] );
30
31         ModificarElemento( arreglo[ 3 ] ); // trata de modificar arreglo[ 3 ]

```

Figura 8.13 | Paso de arreglos y elementos individuales de arreglos a los métodos. (Parte 1 de 2).

```

32     Console.WriteLine(
33         "arreglo[3] después de ModificarElemento: {0}", arreglo[ 3 ] );
34 } // fin de Main
35
36 // multiplica cada elemento de un arreglo por 2
37 public static void ModificarArreglo( int[] arreglo2 )
38 {
39     for ( int contador = 0; contador < arreglo2.Length; contador++ )
40         arreglo2[ contador ] *= 2;
41 } // fin del método ModificarArreglo
42
43 // multiplica el argumento por 2
44 public static void ModificarElemento( int elemento )
45 {
46     elemento *= 2;
47     Console.WriteLine(
48         "Valor del elemento en ModificarElemento: {0}", elemento );
49 } // fin del método ModificarElemento
50 } // fin de la clase PasoArreglo

```

Efectos de pasar una referencia a todo un arreglo:

Los valores del arreglo original son:

1 2 3 4 5

Los valores del arreglo modificado son:

2 4 6 8 10

Efectos de pasar el valor de un elemento del arreglo:

arreglo[3] antes de ModificarElemento: 8

Valor del elemento en ModificarElemento: 16

arreglo[3] después de ModificarElemento: 8

Figura 8.13 | Paso de arreglos y elementos individuales de arreglos a los métodos. (Parte 2 de 2).

8.8 Paso de arreglos por valor y por referencia

En C#, una variable que “almacena” un objeto, como un arreglo, en realidad no almacena el objeto en sí. En vez de ello, dicha variable almacena una referencia al objeto (es decir, la ubicación en la memoria de la computadora en donde se almacena el objeto en sí). La distinción entre variables de tipo por referencia y variables de tipo por valor origina ciertas cuestiones sutiles que debemos comprender para crear programas estables y seguros.

Como sabe, cuando una aplicación pasa un argumento a un método, el método que se llamó recibe una copia del valor de ese argumento. Las modificaciones a la copia local en el método que se llamó no afectan a la variable original en el método que hizo la llamada. Si el argumento es de tipo por referencia, el método crea una copia de la referencia, no del objeto actual al que se hace la referencia. La copia local de la referencia también hace referencia al objeto original en memoria, lo que significa que las modificaciones al objeto en el método que se llamó sí afectan al objeto original en memoria.



Tip de rendimiento 8.1

Pasar arreglos y otros objetos por referencia tiene sentido por cuestiones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. En los arreglos grandes que se pasan con frecuencia, esto desperdiciaría tiempo y consumiría una cantidad considerable de almacenamiento para las copias del arreglo; éstos dos problemas ocasionan un rendimiento pobre.

En la sección 7.14 aprendió que C# permite pasar variables por referencia, mediante la palabra clave `ref`. También puede utilizar la palabra clave `ref` para pasar por referencia una variable de tipo por referencia, lo que permite al método que se llamó modificar la variable original en el método que hizo la llamada y se puede hacer

que esa variable haga referencia a un objeto distinto en la memoria. Ésta es una capacidad delicada que, si se le da un mal uso, puede provocar problemas. Por ejemplo, cuando un objeto de tipo por referencia como un arreglo se pasa con `ref`, el método que se llamó recibe el control sobre la referencia en sí, lo que le permite sustituir la referencia original en el método que hizo la llamada con un objeto distinto, o incluso con `null`. Dicho comportamiento puede producir efectos impredecibles, que pueden ser desastrosos en aplicaciones de misión crítica. La aplicación de la figura 8.14 demuestra la sutil diferencia entre pasar una referencia por valor y pasarla por referencia, mediante la palabra clave `ref`.

Las líneas 11 y 14 declaran dos variables arreglo enteras, `primerArreglo` y `copiaPrimerArreglo`. La línea 11 inicializa a `primerArreglo` con los valores 1, 2 y 3. La instrucción de asignación en la línea 14 copia la referencia almacenada en `primerArreglo` a la variable `copiaPrimerArreglo`, lo que hace que estas variables hagan referencia al mismo objeto arreglo en memoria. Creamos la copia de la referencia para poder determinar más adelante si la referencia `primerArreglo` se sobrescribe. La instrucción `for` en las líneas 23-24 imprime el contenido de `primerArreglo` antes de pasarla al método `PrimerDoble` (línea 27), para poder verificar que el arreglo se pase por referencia (es decir, el método que se llamó definitivamente modifica el contenido del arreglo).

```

1  // Fig. 8.14: PruebaReferenciaArreglo.cs
2  // Paso de una referencia a un arreglo
3  // por valor y por referencia.
4  using System;
5
6  public class PruebaReferenciaArreglo
7  {
8      public static void Main( string[] args )
9      {
10         // crea e inicializa primerArreglo
11         int[] primerArreglo = { 1, 2, 3 };
12
13         // copia la referencia en la variable primerArreglo
14         int[] copiaPrimerArreglo = primerArreglo;
15
16         Console.WriteLine(
17             "Prueba: paso de la referencia a primerArreglo por valor" );
18
19         Console.Write( "\nContenido de primerArreglo " +
20             "antes de llamar a PrimerDoble:\n\t" );
21
22         // imprime el contenido de primerArreglo
23         for ( int i = 0; i < primerArreglo.Length; i++ )
24             Console.Write( "{0} ", primerArreglo[ i ] );
25
26         // pasa la variable primerArreglo por valor a PrimerDoble
27         PrimerDoble( primerArreglo );
28
29         Console.Write( "\n\nContenido de primerArreglo después de " +
30             "llamar a PrimerDoble\n\t" );
31
32         // imprime el contenido de primerArreglo
33         for ( int i = 0; i < primerArreglo.Length; i++ )
34             Console.Write( "{0} ", primerArreglo[ i ] );
35
36         // prueba si PrimerDoble cambió la referencia
37         if ( primerArreglo == copiaPrimerArreglo )
38             Console.WriteLine(
39                 "\n\nLas referencias son al mismo arreglo" );
40         else

```

Figura 8.14 | Paso de una referencia a un arreglo por valor y por referencia. (Parte I de 3).

```

41         Console.WriteLine(
42             "\n\nLas referencias son a distintos arreglos" );
43
44         // crea e inicializa segundoArreglo
45         int[] segundoArreglo = { 1, 2, 3 };
46
47         // copia la referencia en la variable segundoArreglo
48         int[] copiaSegundoArreglo = segundoArreglo;
49
50         Console.WriteLine( "\nPrueba: paso de la referencia a segundoArreglo " +
51             "por referencia" );
52
53         Console.Write( "\nContenido de segundoArreglo " +
54             "antes de llamar a SegundoDoble:\n\t" );
55
56         // imprime el contenido de segundoArreglo antes de la llamada al método
57         for ( int i = 0; i < segundoArreglo.Length; i++ )
58             Console.Write( "{0} ", segundoArreglo[ i ] );
59
60         // pasa la variable segundoArreglo por referencia a SegundoDoble
61         SegundoDoble( ref segundoArreglo );
62
63         Console.Write( "\n\nContenido de segundoArreglo " +
64             "antes de llamar a SegundoDoble:\n\t" );
65
66         // imprime el contenido de segundoArreglo antes de la llamada al método
67         for ( int i = 0; i < segundoArreglo.Length; i++ )
68             Console.Write( "{0} ", segundoArreglo[ i ] );
69
70         // prueba si SegundoDoble cambió la referencia
71         if ( segundoArreglo == copiaSegundoArreglo )
72             Console.WriteLine(
73                 "\n\nLas referencias son al mismo arreglo" );
74         else
75             Console.WriteLine(
76                 "\n\nLas referencias son a distintos arreglos" );
77     } // fin del método Main
78
79     // modifica los elementos del arreglo y trata de modificar la referencia
80     public static void PrimerDoble( int[] arreglo )
81     {
82         // duplica el valor de cada elemento
83         for ( int i = 0; i < arreglo.Length; i++ )
84             arreglo[ i ] *= 2;
85
86         // crea nuevo objeto y asigna su referencia a arreglo
87         arreglo = new int[] { 11, 12, 13 };
88     } // fin del método PrimerDoble
89
90     // modifica los elementos de arreglo y modifica la referencia al arreglo
91     // para que haga referencia a un nuevo arreglo
92     public static void SegundoDoble( ref int[] arreglo )
93     {
94         // duplica el valor de cada elemento
95         for ( int i = 0; i < arreglo.Length; i++ )
96             arreglo[ i ] *= 2;
97
98         // crea nuevo objeto y asigna su referencia a arreglo

```

Figura 8.14 | Paso de una referencia a un arreglo por valor y por referencia. (Parte 2 de 3).

```

99     arreglo = new int[] { 11, 12, 13 };
100 } // fin del método SegundoDoble
101 } // fin de la clase PruebaReferenciaArreglo

```

Prueba: paso de la referencia a primerArreglo por valor

Contenido de primerArreglo antes de llamar a PrimerDoble:

1 2 3

Contenido de primerArreglo después de llamar a PrimerDoble:

2 4 6

Las referencias son al mismo arreglo

Prueba: paso de la referencia a segundoArreglo por referencia

Contenido de segundoArreglo antes de llamar a SegundoDoble:

1 2 3

Contenido de segundoArreglo antes de llamar a SegundoDoble:

11 12 13

Las referencias son a distintos arreglos

Figura 8.14 | Paso de una referencia a un arreglo por valor y por referencia. (Parte 3 de 3).

La instrucción `for` en el método `PrimerDoble` (líneas 83-84) multiplica por 2 los valores de todos los elementos en el arreglo. La línea 87 crea un nuevo arreglo que contiene los valores 11, 12 y 13, y asigna la referencia al arreglo al parámetro `arreglo`, en un intento por sobrescribir la referencia `primerArreglo` en el método que hizo la llamada; desde luego que esto no ocurre, ya que la referencia se pasó por valor. Una vez que se ejecuta el método `PrimerDoble`, la instrucción `for` en las líneas 33-34 imprime el contenido de `primerArreglo`, demostrando que el método cambió los valores de los elementos (y confirmando que en C#, los arreglos siempre se pasan por referencia). La instrucción `if...else` en las líneas 37-42 utiliza el operador `==` para comparar las referencias `primerArreglo` (que acabamos de tratar de sobrescribir) y `copiaPrimerArreglo`. La expresión en la línea 37 se evalúa como `true` si los operandos del operador `==` hacen referencia al mismo objeto. En este caso, el objeto representado por `primerArreglo` es el arreglo creado en la línea 11, no el que se crea en el método `PrimerDoble` (línea 87), por lo que no se modificó la referencia original almacenada en `primerArreglo`.

Las líneas 45-76 realizan pruebas similares mediante el uso de las variables `arreglo segundoArreglo` y `copiaSegundoArreglo`, y el método `SegundoDoble` (líneas 92-100). El método `SegundoDoble` realiza las mismas operaciones que `PrimerDoble`, pero recibe su argumento tipo arreglo utilizando la palabra clave `ref`. En este caso, la referencia almacenada en `segundoArreglo` después de la llamada al método es una referencia al arreglo creado en la línea 99 de `SegundoDoble`, lo que demuestra que una variable que se pasa con la palabra clave `ref` puede ser modificada por el método que se llamó, de manera que la variable en el método que hizo la llamada en realidad apunta a un objeto distinto; en este caso, un arreglo creado en `SegundoDoble`. La instrucción `if...else` en las líneas 71-76 confirma que `segundoArreglo` y `copiaSegundoArreglo` ya no hacen referencia al mismo arreglo.



Observación de ingeniería de software 8.1

Cuando un método recibe por valor un parámetro de tipo por referencia, se pasa una copia de la referencia al objeto. Esto evita que un método sobrescriba las referencias que se pasan a ese método. En la mayoría de los casos, proteger la referencia del método que hizo la llamada para que no se modifique es el comportamiento deseado. Si usted se encuentra con una situación en la que realmente desee que el procedimiento que se llamó modifique la referencia al método que hizo la llamada, pase el parámetro de tipo por referencia utilizando la palabra clave `ref`; pero, como dijimos antes, dichas situaciones son muy raras.



Observación de ingeniería de software 8.2

En C#, los objetos (incluyendo arreglos) se pasan por referencia de manera predeterminada. Así, un método que se llamó y que recibe una referencia a un objeto en el método que hizo la llamada, puede modificar el objeto del que hizo la llamada.

8.9 Caso de estudio: la clase LibroCalificaciones que usa un arreglo para ordenar calificaciones

En esta sección desarrollaremos aún más la clase LibroCalificaciones, que presentamos en el capítulo 4 y expandimos en los capítulos 5 y 6. Recuerde que esta clase representa un libro de calificaciones utilizado por un instructor para almacenar y analizar un conjunto de calificaciones de estudiantes. Las versiones anteriores de esta clase procesan un conjunto de calificaciones introducidas por el usuario, pero no mantienen los valores de las calificaciones individuales en variables de instancia de la clase. Por ende, los cálculos repetidos requieren que el usuario introduzca las mismas calificaciones. Una manera de resolver este problema sería almacenar cada calificación introducida por el usuario en una instancia individual de la clase. Por ejemplo, podríamos crear las variables de instancia `calificacion1`, `calificacion2`, ..., `calificacion10` en la clase LibroCalificaciones para almacenar 10 calificaciones de estudiantes. No obstante, el código para totalizar las calificaciones y determinar el promedio de la clase sería voluminoso, y la clase no podría procesar más de 10 calificaciones a la vez. En esta sección resolvemos este problema, mediante el almacenamiento de las calificaciones en un arreglo.

Almacenar las calificaciones de los estudiantes en un arreglo en la clase LibroCalificaciones

La versión de la clase LibroCalificaciones (figura 8.15) que presentamos aquí utiliza un arreglo de enteros para almacenar las calificaciones de varios estudiantes en un solo examen. Esto elimina la necesidad de introducir varias veces el mismo conjunto de calificaciones. El arreglo `calificaciones` se declara como una variable de instancia en la línea 8; por lo tanto, cada objeto LibroCalificaciones mantiene su propio conjunto de calificaciones. El constructor de la clase (líneas 11-15) tiene dos parámetros: el nombre del curso y un arreglo de calificaciones. Cuando una aplicación (por ejemplo, la clase PruebaLibroCalificaciones en la figura 8.16) crea un objeto LibroCalificaciones, la aplicación pasa un arreglo `int` existente al constructor, el cual asigna la referencia del arreglo a la variable de instancia `calificaciones` (línea 14). El tamaño del arreglo `calificaciones` se determina en base a la clase que pasa el arreglo al constructor. Por ende, un objeto LibroCalificaciones puede procesar un número de calificaciones variable; tantos como haya en el arreglo del método que hizo la llamada. Los valores de las calificaciones en el arreglo que se pasa podría introducirlos un usuario desde el teclado o podrían leerse desde un archivo en el disco (como veremos en el capítulo 18). En nuestra aplicación de prueba, simplemente inicializamos un arreglo con un conjunto de valores de calificaciones (figura 8.16, línea 9). Una vez que las calificaciones se almacenan en una variable de instancia llamada `calificaciones` de la clase LibroCalificaciones, todos los métodos de la clase pueden acceder a los elementos de calificaciones según sea necesario para realizar varios cálculos.

El método `ProcesarCalificaciones` (líneas 39-53) contiene una serie de llamadas a métodos que produce un reporte en el que se resumen las calificaciones. La línea 42 llama al método `ImprimirCalificaciones` para imprimir el contenido del arreglo `calificaciones`. Las líneas 136-138 en el método `ImprimirCalificaciones` utilizan una instrucción `for` para imprimir las calificaciones de los estudiantes. En este caso se debe utilizar una instrucción `for` en vez de `foreach`, ya que las líneas 137-138 utilizan el valor de la variable contador `estudiante` para imprimir cada calificación enseguida de un número de estudiante específico (vea la figura 8.16). Aunque los índices de los arreglos empiezan en 0, lo común es que el instructor numere a los estudiantes empezando desde 1. Por ende, las líneas 137-138 imprimen `estudiante + 1` como el número de estudiante para producir las etiquetas "Estudiante 1:", "Estudiante 2:", y así en lo sucesivo.

A continuación, el método `ProcesarCalificaciones` llama al método `ObtenerPromedio` (línea 45) para obtener el promedio de las calificaciones en el arreglo. El método `ObtenerPromedio` (líneas 88-98) utiliza una instrucción `foreach` para totalizar los valores en el arreglo `calificaciones` antes de calcular el promedio. La variable de iteración en el encabezado de la instrucción `foreach` (por ejemplo, `int calificacion`) indica que para cada iteración, la variable `int calificacion` recibe un valor del arreglo `calificaciones`. Observe que el


```

1  // Fig. 8.15: LibroCalificaciones.cs
2  // Libro de calificaciones que utiliza un arreglo para almacenar las calificaciones de
   una prueba.
3  using System;
4
5  public class LibroCalificaciones
6  {
7      private string nombreCurso; // nombre del curso que representa este LibroCalificaciones
8      private int[] calificaciones; // arreglo de calificaciones de los estudiantes
9
10     // el constructor de dos parámetros inicializa nombreCurso y el arreglo calificaciones
11     public LibroCalificaciones( string nombre, int[] arregloCalificaciones )
12     {
13         nombreCurso = nombre; // inicializa nombreCurso
14         calificaciones = arregloCalificaciones; // inicializa el arreglo calificaciones
15     } // fin del constructor LibroCalificaciones con dos parámetros
16
17     // propiedad que obtiene (get) y establece (set) el nombre del curso
18     public string NombreCurso
19     {
20         get
21         {
22             return nombreCurso;
23         } // fin de get
24         set
25         {
26             nombreCurso = value;
27         } // fin de set
28     } // fin de la propiedad NombreCurso
29
30     // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
31     public void MostrarMensaje()
32     {
33         // la propiedad NombreCurso obtiene el nombre del curso
34         Console.WriteLine( "¡Bienvenido al libro de calificaciones para\n{0}!\n",
35                             NombreCurso);
36     } // fin del método MostrarMensaje
37
38     // realiza varias operaciones sobre los datos
39     public void ProcesarCalificaciones()
40     {
41         // imprime en pantalla el arreglo calificaciones
42         ImprimeCalificaciones();
43
44         // llama al método ObtenerPromedio para calcular la calificación promedio
45         Console.WriteLine( "\nEl promedio de la clase es {0:F2}", ObtenerPromedio() );
46
47         // llama a los métodos ObtenerMinimo y ObtenerMaximo
48         Console.WriteLine( "La calificación más baja es {0}\nLa calificación más alta es
49                             {1}\n",
50                             ObtenerMinimo(), ObtenerMaximo() );
51
52         // llama a ImprimirGraficoBarras para imprimir el gráfico de distribución de
53         // calificaciones
54         ImprimirGraficoBarras();
55     } // fin del método ProcesarCalificaciones
56
57     // busca la calificación mínima

```

Figura 8.15 | Libro de calificaciones que utiliza un arreglo para almacenar las calificaciones de una prueba. (Parte I de 3).

```

56 public int ObtenerMinimo()
57 {
58     int califBaja = calificaciones[ 0 ]; // asume que calificaciones[ 0 ] es la más baja
59
60     // itera a través del arreglo calificaciones
61     foreach ( int calificacion in calificaciones )
62     {
63         // si calificacion es menor que califBaja, la asigna a califBaja
64         if ( calificacion < califBaja )
65             califBaja = calificacion; // nueva calificación más baja
66     } // fin de for
67
68     return califBaja; // devuelve la calificación más baja
69 } // fin del método ObtenerMinimo
70
71 // busca la calificación máxima
72 public int ObtenerMaximo()
73 {
74     int califAlta = calificaciones[ 0 ]; // asume que calificaciones[ 0 ] es la más alta
75
76     // itera a través del arreglo calificaciones
77     foreach ( int calificacion in calificaciones )
78     {
79         // si calificacion es mayor que califAlta, la asigna a califAlta
80         if ( calificacion > califAlta )
81             califAlta = calificacion; // nueva calificación más alta
82     } // fin de for
83
84     return califAlta; // devuelve la calificación más alta
85 } // fin del método ObtenerMaximo
86
87 // determina la calificación promedio para la prueba
88 public double ObtenerPromedio()
89 {
90     int total = 0; // inicializa el total
91
92     // suma las calificaciones para un estudiante
93     foreach ( int calificacion in calificaciones )
94         total += calificacion;
95
96     // devuelve el promedio de las calificaciones
97     return ( double ) total / calificaciones.Length;
98 } // fin del método ObtenerPromedio
99
100 // imprime gráfico de barras que muestra la distribución de las calificaciones
101 public void ImprimirGraficoBarras()
102 {
103     Console.WriteLine( "Distribucion de calificaciones:" );
104
105     // almacena la frecuencia de las calificaciones en cada rango de 10 calificaciones
106     int[] frecuencia = new int[ 11 ];
107
108     // para cada calificacion, incrementa la frecuencia apropiada
109     foreach ( int calificacion in calificaciones )
110         ++frecuencia[ calificacion / 10 ];
111
112     // para cada frecuencia de calificaciones, imprime la barra en el gráfico
113     for ( int cuenta = 0; cuenta < frecuencia.Length; cuenta++ )

```

Figura 8.15 | Libro de calificaciones que utiliza un arreglo para almacenar las calificaciones de una prueba. (Parte 2 de 3).

```

114     {
115         // imprime etiqueta de las barras ( "00-09: ", ..., "90-99: ", "100: " )
116         if ( cuenta == 10 )
117             Console.Write( " 100: " );
118         else
119             Console.Write( "{0:D2}-{1:D2}: ",
120                             cuenta * 10, cuenta * 10 + 9 );
121
122         // imprime barra de asteriscos
123         for ( int estrellas = 0; estrellas < frecuencia[ cuenta ]; estrellas++ )
124             Console.Write( "*" );
125
126         Console.WriteLine(); // inicia una nueva línea de salida
127     } // fin de for externo
128 } // fin del método ImprimirGraficoBarras
129
130 // imprime el contenido del arreglo calificaciones
131 public void ImprimeCalificaciones()
132 {
133     Console.WriteLine( "Las calificaciones son:\n" );
134
135     // imprime la calificación de cada estudiante
136     for ( int estudiante = 0; estudiante < calificaciones.Length; estudiante++ )
137         Console.WriteLine( "Estudiante {0,2}: {1,3}",
138                             estudiante + 1, calificaciones[ estudiante ] );
139 } // fin del método ImprimeCalificaciones
140 } // fin de la clase LibroCalificaciones

```

Figura 8.15 | Libro de calificaciones que utiliza un arreglo para almacenar las calificaciones de una prueba. (Parte 3 de 3).

cálculo del promedio en la línea 97 utiliza `calificaciones.Length` para determinar el número de calificaciones que se van a promediar.

Las líneas 48-49 en el método `ProcesarCalificaciones` llaman a los métodos `ObtenerMinimo` y `ObtenerMaximo` para determinar las calificaciones más bajas y más altas de cualquier estudiante en el examen, en forma respectiva. Cada uno de estos métodos utiliza una instrucción `foreach` para iterar a través del arreglo `calificaciones`. Las líneas 61-66 en el método `ObtenerMinimo` iteran a través del arreglo, y las líneas 64-65 comparan cada calificación con `califBaja`. Si una calificación es menor que `califBaja`, se le asigna esa calificación. Cuando la línea 68 se ejecuta, `califBaja` contiene la calificación más baja en el arreglo. El método `ObtenerMaximo` (líneas 72-85) funciona de la misma forma que el método `ObtenerMinimo`.

Por último, la línea 52 en el método `ProcesarCalificaciones` llama al método `ImprimeGraficoBarras` para imprimir un gráfico de distribución de los datos de las calificaciones, mediante el uso de una técnica similar a la de la figura 8.6. En ese ejemplo calculamos en forma manual el número de calificaciones en cada categoría (es decir, 0-9, 10-19, ..., 90-99 y 100) con sólo analizar un conjunto de calificaciones. En este ejemplo, las líneas 109-110 utilizan una técnica similar a la de las figuras 8.7 y 8.8 para calcular la frecuencia de las calificaciones en cada categoría. La línea 106 declara y crea el arreglo `frecuencia` de 11 valores `int` para almacenar la frecuencia de las calificaciones en cada categoría de éstas. Para cada calificación en el arreglo `calificaciones`, las líneas 109-110 incrementan el elemento apropiado del arreglo `frecuencia`. Para determinar qué elemento se debe incrementar, la línea 110 divide la calificación actual entre 10, mediante la división entera. Por ejemplo, si calificación es 85, la línea 110 incrementa `frecuencia[ 8 ]` para actualizar la cuenta de calificaciones en el rango 80-89. Las líneas 113-127 imprimen a continuación el gráfico de barras (vea la figura 8.6) con base en los valores en el arreglo `frecuencia`. Al igual que las líneas 24-25 de la figura 8.6, las líneas 123-124 de la figura 8.15 utilizan un valor en el arreglo `frecuencia` para determinar el número de asteriscos a imprimir en cada barra.

```

1 // Fig. 8.16: PruebaLibroCalificaciones.cs
2 // Crea objeto LibroCalificaciones que utiliza un arreglo de calificaciones.
3 public class GradeBookTest
4 {
5     // El método Main comienza la ejecución de la aplicación
6     public static void Main( string[] args )
7     {
8         // arreglo unidimensional de calificaciones de estudiantes
9         int[] arregloCalificaciones = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10
11         LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
12             "CS101 Introducción a la programación en C#", arregloCalificaciones );
13         miLibroCalificaciones.MostrarMensaje();
14         miLibroCalificaciones.ProcesarCalificaciones();
15     } // fin de Main
16 } // fin de la clase PruebaLibroCalificaciones

```

```

¡Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C#!

```

Las calificaciones son:

```

Estudiante 1: 87
Estudiante 2: 68
Estudiante 3: 94
Estudiante 4: 100
Estudiante 5: 83
Estudiante 6: 78
Estudiante 7: 85
Estudiante 8: 91
Estudiante 9: 76
Estudiante 10: 87

```

```

El promedio de la clase es 84.90
La calificación más baja es 68
La calificación más alta es 100

```

Distribucion de calificaciones:

```

00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *

```

Figura 8.16 | Crea un objeto LibroCalificaciones mediante el uso de un arreglo de calificaciones.

Clase PruebaLibroCalificaciones para demostrar la clase LibroCalificaciones

La aplicación de la figura 8.16 crea un objeto de la clase LibroCalificaciones (figura 8.15) mediante el uso del arreglo arregloCalificaciones (que se declara e inicializa en la línea 9). Las líneas 11-12 pasan el nombre de un curso y arregloCalificaciones al constructor de LibroCalificaciones. La línea 13 imprime un mensaje de bienvenida, y la 14 invoca el método ProcesarCalificaciones del objeto LibroCalificaciones. La salida revela el resumen de las 10 calificaciones en miLibroCalificaciones.



Observación de ingeniería de software 8.3

Un **arnés de prueba** (o aplicación de prueba) es responsable de crear un objeto de la clase que se va a probar y de proporcionarle datos. Estos datos podrían provenir de cualquiera de varias fuentes. Los datos de prueba pueden colocarse directamente en un arreglo con un inicializador de arreglos, puede provenir del usuario mediante el teclado, de un archivo (como veremos en el capítulo 18) o de una red (como veremos en el capítulo 23). Después de pasar estos datos al constructor de la clase para instanciar el objeto, este arnés de prueba debe llamar al objeto para probar sus métodos y manipular sus datos. La recopilación de datos en el arnés de prueba de esta forma permite a la clase manipular datos de varias fuentes.

8.10 Arreglos multidimensionales

Los **arreglos multidimensionales** de dos dimensiones se utilizan con frecuencia para representar **tablas de valores**, que consisten en información ordenada en **filas** y **columnas**. Para identificar una tabla en particular, debemos especificar dos índices. Por convención, el primero identifica la fila del elemento y el segundo su columna. Los arreglos que requieren dos índices para identificar un elemento específico se llaman **arreglos bidimensionales** (los arreglos multidimensionales pueden tener más de dos dimensiones, pero los arreglos con más de dos dimensiones están más allá del alcance de este libro). C# soporta dos tipos de arreglos bidimensionales: **arreglos rectangulares** y **arreglos dentados**.

Arreglos rectangulares

Los arreglos rectangulares se utilizan para representar tablas de información en la forma de filas y columnas, en donde cada fila tiene el mismo número de columnas. La figura 8.17 ilustra un arreglo rectangular llamado **a**, el cual contiene tres filas y cuatro columnas: un arreglo de tres por cuatro. En general, un arreglo con m filas y n columnas se llama **arreglo de m por n** .

Cada elemento en el arreglo **a** se identifica en la figura 8.17 mediante una expresión de acceso a un arreglo de la forma **a[fila, columna]**; **a** es el nombre del arreglo, **fila** y **columna** son los índices que identifican en forma única a cada elemento en el arreglo **a** por número de fila y columna. Observe que los nombres de los elementos en la fila 0 tienen todos un primer índice de 0, y los nombres de los elementos en la columna 3 tienen un segundo índice de 3.

Al igual que los arreglos unidimensionales, los arreglos multidimensionales pueden inicializarse mediante inicializadores de arreglos en las declaraciones. Un arreglo rectangular **b** con dos filas y dos columnas debe declararse e inicializarse con **inicializadores de arreglos anidados**, como se muestra a continuación:

```
int[ , ] b = { { 1, 2 }, { 3, 4 } };
```

Los valores del inicializador se agrupan por fila entre llaves. Así, 1 y 2 inicializan a **b[0, 0]** y **b[0, 1]**, respectivamente; 3 y 4 inicializan a **b[1, 0]** y **b[1, 1]**, respectivamente. El compilador cuenta el número de inicializadores de arreglos anidados (representados por conjuntos de dos llaves internas dentro de las llaves externas) en la

	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	a[0, 0]	a[0, 1]	a[0, 2]	a[0, 3]
Fila 1	a[1, 0]	a[1, 1]	a[1, 2]	a[1, 3]
Fila 2	a[2, 0]	a[2, 1]	a[2, 2]	a[2, 3]

Índice de columna
 Índice de fila
 Nombre de arreglo

Figura 8.17 | Arreglo rectangular con tres filas y cuatro columnas.

declaración del arreglo, para determinar el número de filas en el arreglo b. El compilador cuenta los valores en el inicializador de arreglos anidado de una fila, para determinar el número de columnas (dos) en esa fila. El compilador generará un error si el número de inicializadores en cada fila no es el mismo, ya que cada fila de un arreglo rectangular debe tener la misma longitud.

Arreglos dentados

Un *arreglo dentado* se mantiene como un arreglo unidimensional, en el cual cada elemento hace referencia a un arreglo unidimensional. La forma en que se representan los arreglos dentados los hace bastante flexibles, debido a que las longitudes de las filas en el arreglo no necesitan ser las mismas. Por ejemplo, los arreglos dentados podrían utilizarse para almacenar las calificaciones de los exámenes de un solo estudiante a lo largo de varias clases, en donde el número de exámenes podría variar de una clase a otra.

Para acceder a los elementos en un arreglo dentado, podemos utilizar una expresión de acceso a un arreglo de la forma *nombreArreglo[fila][columna]*; una expresión similar a la expresión para los arreglos rectangulares, pero con un conjunto separado de corchetes para cada dimensión. Un arreglo dentado con tres filas de distintas longitudes podría declararse e inicializarse como se muestra a continuación:

```
int[][] dentado = { new int[] { 1, 2 },
                   new int[] { 3 },
                   new int[] { 4, 5, 6 } };
```

En esta instrucción, el 1 y el 2 inicializan a `dentado[ 0 ][ 0 ]` y `dentado[ 0 ][ 1 ]`, en forma respectiva; el 3 inicializa a `dentado[ 1 ][ 0 ]`; el 4, 5 y 6 inicializan a `dentado[ 2 ][ 0 ]`, `dentado[ 2 ][ 1 ]` y `dentado[ 2 ][ 2 ]`, respectivamente. Por lo tanto, el arreglo `dentado` en la declaración anterior está compuesto de cuatro arreglos unidimensionales separados; uno que representa las filas, otro que contiene los valores en la primera fila ({ 1, 2 }), uno que contiene el valor en la segunda fila ({ 3 }) y otro que contiene los valores en la tercera fila ({ 4, 5, 6 }). Así, el arreglo `dentado` en sí es un arreglo de tres elementos, cada uno de los cuales hace referencia a un arreglo unidimensional de valores `int`.

Observe las diferencias entre las expresiones de creación de arreglos para los arreglos rectangulares y para los arreglos dentados. Dos conjuntos de corchetes van después del tipo de `dentado`, lo que indica que es un arreglo de arreglos `int`. Lo que es más, en el inicializador del arreglo, C# requiere la palabra clave `new` para crear un objeto arreglo para cada fila. La figura 8.18 ilustra la referencia al arreglo `dentado`, una vez que se declara y se inicializa.

Creación de arreglos bidimensionales mediante expresiones de creación de arreglos

Un arreglo rectangular puede crearse mediante una expresión de creación de arreglos. Por ejemplo, las siguientes líneas declaran el arreglo `b` y lo asignan como una referencia a un arreglo rectangular de tres por cuatro:

```
int[ , ] b;
b = new int[ 3, 4 ];
```

En este caso utilizamos los valores literales 3 y 4 para especificar el número de filas y columnas respectivamente, pero esto no es requerido; las aplicaciones también pueden utilizar variables y expresiones para especificar las dimensiones de los arreglos. Al igual que con los arreglos unidimensionales, los elementos de un arreglo rectangular se inicializan cuando se crea el objeto arreglo.

Un arreglo dentado no puede crearse por completo con una sola expresión de creación de arreglos. La siguiente instrucción es un error de sintaxis:

```
int[][] c = new int[ 2 ][ 5 ]; // error
```

En vez de ello, cada arreglo unidimensional en el arreglo dentado debe inicializarse por separado. Un arreglo dentado puede crearse de la siguiente manera:

```
int[][] c;
c = new int[ 2 ][]; // crea 2 filas
c[ 0 ] = new int[ 5 ]; // crea 5 columnas para la fila 0
c[ 1 ] = new int[ 3 ]; // crea 3 columnas para la fila 1
```

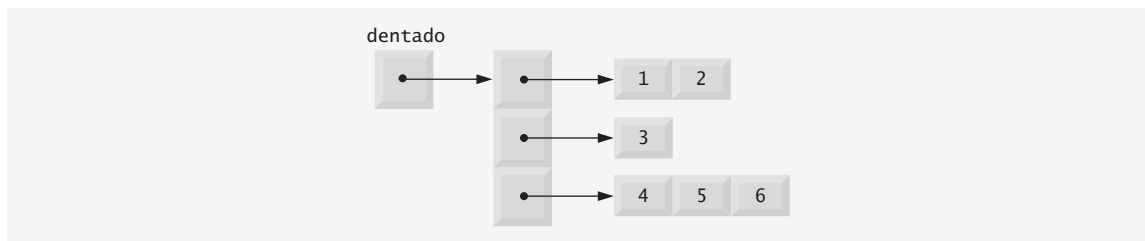


Figura 8.18 | Arreglo dentado con tres filas de distintas longitudes.

Las instrucciones anteriores crean un arreglo dentado con dos filas. La fila 0 tiene cinco columnas y la fila 1 tiene tres columnas.

Ejemplo de arreglos bidimensionales: mostrar los valores de los elementos

La figura 8.19 demuestra la inicialización de arreglos rectangulares y dentados con inicializadores de arreglos y mediante el uso de ciclos for anidados para *recorrer* los arreglos (es decir, visitar cada elemento de cada arreglo).

El método Main de la clase InicArreglo declara dos arreglos. La declaración de `rectangular` (línea 12) utiliza inicializadores de arreglos anidados para inicializar la fila 0 del arreglo con los valores 1, 2 y 3, y la fila 1 con los valores 4, 5 y 6. La declaración de `dentado` (líneas 17-19) utiliza inicializadores anidados de distintas longitudes. En este caso, el inicializador utiliza la palabra clave `new` para crear un arreglo unidimensional para cada fila. La fila 0 se inicializa para tener dos elementos con los valores 1 y 2, respectivamente; la fila 1, para tener un elemento con el valor 3; y la fila 2, para tener tres elementos con los valores 4, 5 y 6.

```

1  // Fig. 8.19: InicArreglo.cs
2  // Inicialización de arreglos rectangulares y dentados.
3  using System;
4
5  public class InicArreglo
6  {
7      // crea e imprime arreglos rectangulares y dentados
8      public static void Main( string[] args )
9      {
10         // con los arreglos rectangulares,
11         // cada columna debe tener la misma longitud.
12         int[ , ] rectangular = { { 1, 2, 3 }, { 4, 5, 6 } };
13
14         // con los arreglos dentados,
15         // necesitamos usar "new int[]" para cada fila,
16         // pero cada columna no necesita tener la misma longitud.
17         int[][] dentado = { new int[] { 1, 2 },
18                             new int[] { 3 },
19                             new int[] { 4, 5, 6 } };
20
21         ImprimirArreglo( rectangular ); // muestra el arreglo rectangular por fila
22         Console.WriteLine(); // imprime una línea en blanco
23         ImprimirArreglo( dentado ); // muestra el arreglo dentado por fila
24     } // fin de Main
25
26     // imprime filas y columnas de un arreglo rectangular
27     public static void ImprimirArreglo( int[ , ] arreglo )
28     {
29         Console.WriteLine( "Los valores en el arreglo rectangular por fila son" );
  
```

Figura 8.19 | Inicialización de arreglos dentados y rectangulares. (Parte I de 2).

```

30
31 // itera a través de las filas del arreglo
32 for ( int fila = 0; fila < arreglo.GetLength( 0 ); fila++ )
33 {
34     // itera a través de las columnas de la fila actual
35     for ( int columna = 0; columna < arreglo.GetLength( 1 ); columna++ )
36         Console.Write( "{0} ", arreglo[ fila, columna ] );
37
38     Console.WriteLine(); // inicia nueva línea de salida
39 } // fin de for externo
40 } // fin del método ImprimirArreglo
41
42 // imprime filas y columnas de un arreglo dentado
43 public static void ImprimirArreglo( int[][] arreglo )
44 {
45     Console.WriteLine( "Los valores en el arreglo dentado por fila son" );
46
47     // itera a través de las filas del arreglo
48     for ( int fila = 0; fila < arreglo.Length; fila++ )
49     {
50         // itera a través de las columnas de la fila actual
51         for ( int columna = 0; columna < arreglo[ fila ].Length; columna++ )
52             Console.Write( "{0} ", arreglo[ fila ][ columna ] );
53
54         Console.WriteLine(); // inicia nueva línea de salida
55     } // fin de for externo
56 } // fin del método ImprimirArreglo
57 } // fin de la clase InicArreglo

```

Los valores en el arreglo rectangular por fila son

```

1 2 3
4 5 6

```

Los valores en el arreglo dentado por fila son

```

1 2
3
4 5 6

```

Figura 8.19 | Inicialización de arreglos dentados y rectangulares. (Parte 2 de 2).

El método `ImprimirArreglo` está sobrecargado con dos versiones. La primera (líneas 27-40) especifica el parámetro del arreglo como `int[ , ] arreglo` para indicar que recibe un arreglo rectangular. La segunda (líneas 43-56) recibe un arreglo dentado, ya que su parámetro tipo arreglo se lista como `int[][] arreglo`.

La línea 21 invoca al método `ImprimirArreglo` con el argumento rectangular, por lo que se hace una llamada a la versión de `ImprimirArreglo` de las líneas 27-40. La instrucción `for` (líneas 32-39) imprime las filas de un arreglo rectangular. La condición de continuación de ciclo de cada instrucción `for` (líneas 32 y 35) utiliza el método `GetLength` del arreglo rectangular para obtener la longitud de cada dimensión. Las dimensiones se numeran a partir de 0. Entonces, la llamada al método `GetLength( 0 )` de `arreglo` devuelve el tamaño de la primera dimensión del arreglo (el número de filas), y la llamada `GetLength( 1 )` devuelve el tamaño de la segunda dimensión (el número de columnas).

La línea 23 invoca al método `ImprimirArreglo` con el argumento dentado, por lo que se hace una llamada a la versión de `ImprimirArreglo` de las líneas 43-56. La instrucción `for` (líneas 48-55) imprime las filas de un arreglo dentado. En la condición de continuación de ciclo de la instrucción `for` externa (línea 48), utilizamos la propiedad `arreglo.Length` para determinar el número de filas en el arreglo. En la instrucción `for` interna (línea 51), utilizamos la propiedad `arreglo[ fila ].Length` para determinar el número de columnas en la fila actual del arreglo. Esta condición permite que el ciclo determine el número exacto de columnas en cada fila.

Manipulaciones comunes de arreglos multidimensionales mediante el uso de instrucciones for

Muchas manipulaciones comunes de arreglos utilizan instrucciones for. Como ejemplo, la siguiente instrucción establece todos los elementos en la fila 2 del arreglo rectangular a en la figura 8.17 a 0:

```
for ( int columna = 0; columna < a.GetLength( 1 ); columna++ )
    a[ 2, columna ] = 0;
```

Especificamos la fila 2; por lo tanto, sabemos que el primer índice siempre es 2 (0 es la primera fila y 1 es la segunda). Este ciclo for varía sólo el segundo índice (es decir, el índice de columna). La instrucción for anterior es equivalente a las instrucciones de asignación

```
a[ 2, 0 ] = 0;
a[ 2, 1 ] = 0;
a[ 2, 2 ] = 0;
a[ 2, 3 ] = 0;
```

La siguiente instrucción for anidada totaliza los valores de todos los elementos en el arreglo a:

```
int total = 0;

for ( int fila = 0; fila < a.GetLength( 0 ); fila++ )
{
    for ( int columna = 0; columna < a.GetLength( 1 ); columna++ )
        total += a[ fila, columna ];
} // fin de for externo
```

Estas instrucciones for anidadas totalizan los elementos del arreglo, una fila a la vez. La instrucción for externa empieza por establecer el índice de fila a 0, de manera que los elementos de la fila 0 pueden totalizarse mediante la instrucción for interna. Después, la instrucción for externa incrementa fila en 1, para que puedan totalizarse los elementos de la fila 1. Después el for externo incrementa fila en 2, para que los elementos de la fila 2 puedan totalizarse. La variable total puede imprimirse al terminar la instrucción for externa. En el siguiente ejemplo mostraremos cómo procesar un arreglo rectangular en una forma más concisa, mediante el uso de instrucciones foreach.

8.11 Caso de estudio: la clase LibroCalificaciones que usa un arreglo rectangular

En la sección 8.9 presentamos la clase LibroCalificaciones (figura 8.15), la cual utiliza un arreglo unidimensional para almacenar las calificaciones de los estudiantes en un solo examen. En la mayoría de los cursos, los estudiantes presentan varios exámenes. Es probable que los instructores quieran analizar las calificaciones a lo largo de todo el curso, tanto para un solo estudiante como para la clase en general.

Cómo almacenar las calificaciones de los estudiantes en un arreglo rectangular en la clase LibroCalificaciones

La figura 8.20 contiene una versión de la clase LibroCalificaciones que utiliza un arreglo rectangular llamado calificaciones, para almacenar las calificaciones de un número de estudiantes en varios exámenes. Cada fila del arreglo representa las calificaciones de un solo estudiante durante todo el curso, y cada columna representa las calificaciones para la clase completa en uno de los exámenes que presentaron los estudiantes durante el curso. Una aplicación como PruebaLibroCalificaciones (figura 8.21) pasa el arreglo como argumento para el constructor de LibroCalificaciones. En este ejemplo, utilizamos un arreglo de 10 por 3 que contiene 10 calificaciones de los estudiantes en tres exámenes. Cinco métodos realizan manipulaciones de arreglos para procesar las calificaciones. Cada método es similar a su contraparte en la primera versión de la clase LibroCalificaciones con un arreglo unidimensional (figura 8.15). El método ObtenerMinimo (líneas 54-68) determina la calificación más baja de cualquier estudiante durante el semestre. El método ObtenerMaximo (líneas 71-85) determina la calificación más alta de cualquier estudiante durante el semestre. El método ObtenerPromedio (líneas 88-100) determina el promedio semestral de un estudiante específico. El método ImprimirGraficoBarras (líneas 103-132)

```

1  // Fig. 8.20: LibroCalificaciones.cs
2  // Libro de calificaciones que utiliza un arreglo rectangular para almacenar las
   calificaciones.
3  using System;
4
5  public class LibroCalificaciones
6  {
7      private string nombreCurso; // nombre del curso que representa este libro de
   calificaciones
8      private int[ , ] calificaciones; // arreglo rectangular de calificaciones de estudiantes
9
10     // el constructor con dos parámetros inicializa nombreCurso y el arreglo calificaciones
11     public LibroCalificaciones( string nombre, int[ , ] arregloCalificaciones )
12     {
13         NombreCurso = nombre; // inicializa nombreCurso
14         calificaciones = arregloCalificaciones; // inicializa el arreglo calificaciones
15     } // fin del constructor de LibroCalificaciones con dos parámetros
16
17     // propiedad que obtiene (get) y establece (set) el nombre del curso
18     public string NombreCurso
19     {
20         get
21         {
22             return nombreCurso;
23         } // fin de get
24         set
25         {
26             nombreCurso = value;
27         } // fin de set
28     } // fin de la propiedad NombreCurso
29
30     // muestra un mensaje de bienvenida al usuario LibroCalificaciones
31     public void MostrarMensaje()
32     {
33         // la propiedad NombreCurso obtiene el nombre del curso
34         Console.WriteLine( "¡Bienvenido al libro de calificaciones para\n{0}!\n",
35             NombreCurso );
36     } // fin del método MostrarMensaje
37
38     // realiza varias operaciones sobre los datos
39     public void ProcesarCalificaciones()
40     {
41         // imprime el arreglo de calificaciones
42         ImprimirCalificaciones();
43
44         // llama a los métodos ObtenerMinima y ObtenerMaxima
45         Console.WriteLine( "\n{0} {1}\n{2} {3}\n",
46             "La calificación más baja en el libro de calificaciones es", ObtenerMinima(),
47             "La calificación más alta en el libro de calificaciones es", ObtenerMaxima() );
48
49         // imprime la gráfica de distribución de todas las calificaciones en todas las
   pruebas
50         ImprimirGraficoBarras();
51     } // fin del método ProcesarCalificaciones
52
53     // busca la calificación más baja
54     public int ObtenerMinima()
55     {

```

Figura 8.20 | Libro de calificaciones que utiliza un arreglo rectangular para almacenar calificaciones. (Parte I de 3).

```

56 // asume que el primer elemento del arreglo calificaciones es el más bajo
57 int califBaja = calificaciones[ 0, 0 ];
58
59 // itera a través de los elementos del arreglo rectangular calificaciones
60 foreach ( int calificacion in calificaciones )
61 {
62     // si calificacion es menor que califBaja, la asigna a califBaja
63     if ( calificacion < califBaja )
64         califBaja = calificacion;
65 } // fin de foreach
66
67 return califBaja; // devuelve la calificación más baja
68 } // fin del método ObtenerMinima
69
70 // busca la calificación más alta
71 public int ObtenerMaxima()
72 {
73     // asume que el primer elemento del arreglo calificaciones es el más alto
74     int califAlta = calificaciones[ 0, 0 ];
75
76     // itera a través de los elementos del arreglo rectangular calificaciones
77     foreach ( int calificacion in calificaciones )
78     {
79         // si calificacion es mayor que califAlta, la asigna a califAlta
80         if ( calificacion > califAlta )
81             califAlta = calificacion;
82     } // fin de foreach
83
84     return califAlta; // devuelve la calificación más alta
85 } // fin del método ObtenerMaxima
86
87 // determina la calificación promedio para un estudiante específico
88 public double ObtenerPromedio( int estudiante )
89 {
90     // obtiene el número de calificaciones por estudiante
91     int monto = calificaciones.GetLength( 1 );
92     int total = 0; // inicializa el total
93
94     // suma las calificaciones para un estudiante
95     for ( int examen = 0; examen < monto; examen++ )
96         total += calificaciones[ estudiante, examen ];
97
98     // devuelve el promedio de las calificaciones
99     return ( double ) total / monto;
100 } // fin del método ObtenerPromedio
101
102 // imprime gráfico de barras que muestra la distribución total de calificaciones
103 public void ImprimirGraficoBarras()
104 {
105     Console.WriteLine( "Distribución total de calificaciones:" );
106
107     // almacena la frecuencia de las calificaciones en cada rango de 10 calificaciones
108     int[] frecuencia = new int[ 11 ];
109
110     // para cada calificacion en LibroCalificaciones, incrementa la frecuencia apropiada
111     foreach ( int calificacion in calificaciones )
112     {
113         ++frecuencia[ calificacion / 10 ];

```

Figura 8.20 | Libro de calificaciones que utiliza un arreglo rectangular para almacenar calificaciones. (Parte 2 de 3).

```

114 } // fin de foreach
115
116 // para cada frecuencia de calificacion, imprime la barra en el gráfico
117 for ( int cuenta = 0; cuenta < frecuencia.Length; cuenta++ )
118 {
119     // imprime etiqueta de barras ( "00-09: ", ..., "90-99: ", "100: " )
120     if ( cuenta == 10 )
121         Console.Write( " 100: " );
122     else
123         Console.Write( "{0:D2}-{1:D2}: ",
124             cuenta * 10, cuenta * 10 + 9 );
125
126     // imprime barra de asteriscos
127     for ( int estrellas = 0; estrellas < frecuencia[ cuenta ]; estrellas++ )
128         Console.Write( "*" );
129
130     Console.WriteLine(); // inicia una nueva línea de salida
131 } // fin de for externo
132 } // fin del método ImprimirGraficoBarras
133
134 // imprime el contenido del arreglo calificaciones
135 public void ImprimirCalificaciones()
136 {
137     Console.WriteLine( "Las calificaciones son:\n" );
138     Console.Write( " " ); // alinea encabezados de columna
139
140     // crea un encabezado de columna para cada una de las pruebas
141     for ( int prueba = 0; prueba < calificaciones.GetLength( 1 ); prueba++ )
142         Console.Write( "Prueba {0} ", prueba + 1 );
143
144     Console.WriteLine( "Promedio" ); // encabezado de columna de promedio de estudiante
145
146     // crea filas/columnas de texto que representan el arreglo calificaciones
147     for ( int estudiante = 0; estudiante < calificaciones.GetLength( 0 ); estudiante++ )
148     {
149         Console.Write( "Estudiante {0,2}", estudiante + 1 );
150
151         // imprime las calificaciones de los estudiantes
152         for ( int calificacion = 0; calificacion < calificaciones.GetLength( 1 );
153             calificacion++ )
154             Console.Write( "{0,8}", calificaciones[ estudiante, calificacion ] );
155
156         // llama al método ObtenerPromedio para calcular la calificación promedio de
157         // los estudiantes;
158         // pasa el número de fila como argumento para ObtenerPromedio
159         Console.WriteLine( "{0,9:F2}", ObtenerPromedio( estudiante ) );
160     } // fin de for externo
161 } // fin del método ImprimirCalificaciones
162 } // fin de la clase LibroCalificaciones

```

Figura 8.20 | Libro de calificaciones que utiliza un arreglo rectangular para almacenar calificaciones. (Parte 3 de 3).

imprime un gráfico de barras de la distribución de todas las calificaciones de los estudiantes durante el semestre. El método `ImprimirCalificaciones` (líneas 135-159) imprime el arreglo bidimensional en formato tabular, junto con el promedio semestral de cada estudiante.

Cada uno de los métodos `ObtenerMinimo`, `ObtenerMaximo` e `ImprimirGraficoBarras` itera a través del arreglo `calificaciones` mediante el uso de la instrucción `foreach`; por ejemplo, la instrucción `foreach` del método `ObtenerMinimo` (líneas 60-65). Para buscar la calificación más baja, esta instrucción `foreach` itera

a través del arreglo rectangular calificaciones y compara cada elemento con la variable califBaja. Si una calificación es menor que califBaja, a califBaja se le asigna esa calificación.

Cuando la instrucción `foreach` recorre los elementos del arreglo `calificaciones`, analiza cada elemento de la primera fila en orden por índice, después cada elemento de la segunda fila en orden por índice, y así en lo sucesivo. La instrucción `foreach` en las líneas 60-65 recorre los elementos de `calificacion` en el mismo orden que el siguiente código equivalente, expresado con instrucciones `for` anidadas:

```
for ( int fila = 0; fila < calificaciones.GetLength( 0 ); fila++ )
    for ( int columna = 0; columna < calificaciones.GetLength( 1 ); columna++ )
    {
        if ( calificaciones[ fila, columna ] < califBaja )
            califBaja = calificaciones[ fila, columna ];
    }
```

Cuando se completa la instrucción `foreach`, `califBaja` contiene la calificación más baja en el arreglo rectangular. El método `ObtenerMaximo` funciona de manera similar al método `ObtenerMinimo`.

El método `ImprimirGraficoBarras` en la figura 8.20 muestra la distribución de calificaciones en forma de un gráfico de barras. Observe que la sintaxis de la instrucción `foreach` (líneas 111-114) es idéntica para los arreglos unidimensionales y bidimensionales.

El método `ImprimirCalificaciones` (líneas 135-159) utiliza instrucciones `for` anidadas para imprimir valores del arreglo `calificaciones`, además del promedio semestral de cada estudiante. La salida en la figura 8.21 muestra el resultado, que se asemeja al formato tabular del libro de calificaciones real de un instructor. Las líneas 141-142 imprimen los encabezados de columna para cada prueba. Aquí utilizamos la instrucción `for` en vez de la instrucción `foreach`, para poder identificar cada prueba con un número. De manera similar, la instrucción `for` en las líneas 147-158 imprime primero una etiqueta de fila mediante el uso de una variable contador para identificar a cada estudiante (línea 149). Aunque los índices de los arreglos empiezan en 0, observe que las líneas 142 y 149 imprimen `prueba + 1` y `estudiante + 1` en forma respectiva, para producir números de prueba y estudiante que empiecen en 1 (vea la figura 8.21). La instrucción `for` interna en las líneas 152-153 utiliza la variable contador `estudiante` de la instrucción `for` externa para iterar a través de una fila específica del arreglo `calificaciones` e imprime la calificación de la prueba de cada estudiante. Por último, la línea 157 obtiene el promedio semestral de cada estudiante, para lo cual pasa el índice de fila de `calificaciones` (es decir, `estudiante`) al método `ObtenerPromedio`.

El método `ObtenerPromedio` (líneas 88-100) recibe un argumento: el índice de fila para un estudiante específico. Cuando la línea 157 llama a `ObtenerPromedio`, el argumento es el valor `int` de `estudiante`, el cual especifica la fila particular del arreglo rectangular `calificaciones`. El método `ObtenerPromedio` calcula la suma de los elementos del arreglo en esta fila, divide el total entre el número de resultados de la prueba y devuelve el resultado de punto flotante como un valor `double` (línea 99).

Clase PruebaLibroCalificaciones que demuestra la clase LibroCalificaciones

La aplicación en la figura 8.21 crea un objeto de la clase `LibroCalificaciones` (figura 8.20) mediante el uso del arreglo bidimensional de valores `int` llamado `arregloCalificaciones` (el cual se declara y se inicializa en las líneas 9-18). Las líneas 20-21 pasan el nombre de un curso y `arregloCalificaciones` al constructor de `LibroCalificaciones`. Después, las líneas 22-23 invocan a los métodos `MostrarMensaje` y `ProcesarCalificaciones` de `miLibroCalificaciones` para mostrar un mensaje de bienvenida y obtener un reporte que sintetice las calificaciones de los estudiantes para el semestre, en forma respectiva.

8.12 Listas de argumentos de longitud variable

Las *listas de argumentos de longitud variable* le permiten crear métodos que reciben un número arbitrario de argumentos. Un argumento tipo arreglo unidimensional que va precedido por la palabra clave ***params*** en la lista de parámetros del método indica que éste recibe un número variable de argumentos con el tipo de los elementos del arreglo. Este uso de un modificador ***params*** puede ocurrir sólo en la última entrada de la lista de parámetros. Aunque podemos utilizar la sobrecarga de métodos y el paso de arreglos para realizar gran parte de lo que se logra con los “varargs” (otro nombre para las listas de argumentos de longitud variable), es más conciso utilizar el modificador ***params***.

```

1 // Fig. 8.21: PruebaLibroCalificaciones.cs
2 // Crea objeto LibroCalificaciones que utiliza un arreglo rectangular de calificaciones.
3 public class PruebaLibroCalificaciones
4 {
5     // el método Main comienza la ejecución de la aplicación
6     public static void Main( string[] args )
7     {
8         // arreglo rectangular de calificaciones de estudiantes
9         int[ , ] arregloCalificaciones = { { 87, 96, 70 },
10                                            { 68, 87, 90 },
11                                            { 94, 100, 90 },
12                                            { 100, 81, 82 },
13                                            { 83, 65, 85 },
14                                            { 78, 87, 65 },
15                                            { 85, 75, 83 },
16                                            { 91, 94, 100 },
17                                            { 76, 72, 84 },
18                                            { 87, 93, 73 } };
19
20         LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
21             "CS101 Introducción a la programación en C#", arregloCalificaciones );
22         miLibroCalificaciones.MostrarMensaje();
23         miLibroCalificaciones.ProcesarCalificaciones();
24     } // fin de Main
25 } // fin de la clase PruebaLibroCalificaciones

```

¡Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C#!

Las calificaciones son:

	Prueba 1	Prueba 2	Prueba 3	Promedio
Estudiante 1	87	96	70	84.33
Estudiante 2	68	87	90	81.67
Estudiante 3	94	100	90	94.67
Estudiante 4	100	81	82	87.67
Estudiante 5	83	65	85	77.67
Estudiante 6	78	87	65	76.67
Estudiante 7	85	75	83	81.00
Estudiante 8	91	94	100	95.00
Estudiante 9	76	72	84	77.33
Estudiante 10	87	93	73	84.33

La calificación más baja en el libro de calificaciones es 65

La calificación más alta en el libro de calificaciones es 100

Distribución total de calificaciones:

00-09:

10-19:

20-29:

30-39:

40-49:

50-59:

60-69: ***

70-79: *****

80-89: *****

90-99: *****

100: ***

Figura 8.21 | Crea un objeto LibroCalificaciones que utiliza un arreglo rectangular de calificaciones.

La figura 8.22 demuestra el método Promedio (líneas 8-17), que recibe una secuencia de longitud variable de valores `double` (línea 8). C# trata a la lista de argumentos de longitud variable como un arreglo unidimensional cuyos elementos son todos del mismo tipo. Así, el cuerpo del método puede manipular el parámetro `numeros` como un arreglo de valores `double`. Las líneas 13-14 utilizan el ciclo `foreach` para recorrer el arreglo y calcular el total de los valores `double` en el arreglo. La línea 16 accede a `numeros.Length` para obtener el tamaño del arreglo `numeros` y usarlo en el cálculo del promedio. Las líneas 31, 33 y 35 en `Main` llaman al método `Promedio` con dos, tres y cuatro argumentos, en forma respectiva. El método `Promedio` tiene una lista de argumentos de longitud variable, por lo que puede promediar todos los argumentos `double` que le pase el método que hace la llamada. La salida revela que cada llamada al método `Promedio` devuelve el valor correcto.

```

1  // Fig. 8.22: PruebaVarArgs.cs
2  // Uso de listas de argumentos de longitud variable.
3  using System;
4
5  public class PruebaVarArgs
6  {
7      // calcula el promedio
8      public static double Promedio( params double[] numeros )
9      {
10         double total = 0.0; // inicializa el total
11
12         // calcula el total usando la instrucción foreach
13         foreach ( double d in numeros )
14             total += d;
15
16         return total / numeros.Length;
17     } // fin del método Promedio
18
19     public static void Main( string[] args )
20     {
21         double d1 = 10.0;
22         double d2 = 20.0;
23         double d3 = 30.0;
24         double d4 = 40.0;
25
26         Console.WriteLine(
27             "d1 = {0:F1}\nd2 = {1:F1}\nd3 = {2:F1}\nd4 = {3:F1}\n",
28             d1, d2, d3, d4 );
29
30         Console.WriteLine( "El promedio de d1 y d2 es {0:F1}",
31             Promedio( d1, d2 ) );
32         Console.WriteLine( "El promedio de d1, d2 y d3 es {0:F1}",
33             Promedio( d1, d2, d3 ) );
34         Console.WriteLine( "El promedio de d1, d2, d3 y d4 es {0:F1}",
35             Promedio( d1, d2, d3, d4 ) );
36     } // fin de Main
37 } // fin de la clase PruebaVarArgs

```

```

d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

```

```

El promedio de d1 y d2 es 15.0
El promedio de d1, d2 y d3 es 20.0
El promedio de d1, d2, d3 y d4 es 25.0

```

Figura 8.22 | Uso de listas de argumentos de longitud variable.



Error común de programación 8.4

Utilizar el modificador `params` con un parámetro en medio de la lista de parámetros de un método es un error de sintaxis. El modificador `params` puede usarse sólo con el último parámetro de la lista de parámetros.

8.13 Uso de argumentos de línea de comandos

En muchos sistemas es posible pasar argumentos desde la línea de comandos (estos se conocen como *argumentos de línea de comandos*) a una aplicación; para ello se incluye un parámetro de tipo `string[]` (es decir, un arreglo de objetos `string`) en la lista de parámetros de `Main`, de la misma forma exacta que como lo hemos hecho en cada aplicación de este libro. Por convención, este parámetro se llama `args` (figura 8.23, línea 7). Cuando se ejecuta una aplicación desde el Símbolo del sistema, el entorno de ejecución pasa los argumentos de línea de comandos que aparecen después del nombre de la aplicación al método `Main` de la aplicación, como objetos `string` en el arreglo unidimensional `args`. El número de argumentos que se pasan desde la línea de comandos se obtiene al acceder a la propiedad `Length` del arreglo. Por ejemplo, el comando "`MiAplicacion a b`" pasa dos argumentos de línea de comandos a la aplicación `MiAplicacion`. Observe que los argumentos de línea de comandos se separan por espacio en blanco, no por comas. Cuando se ejecuta el comando anterior, el punto de entrada del método `Main` recibe el arreglo de dos elementos `args` (es decir, `args.Length` es 2), en donde `args[0]` contiene el objeto `string` "`a`" y `args[1]` contiene el objeto `string` "`b`". Los usos comunes de los argumentos de línea de comandos incluyen el paso de opciones y nombres de archivo a las aplicaciones.

```

1  // Fig. 8.23: InicArreglo.cs
2  // Uso de argumentos de línea de comandos para inicializar un arreglo.
3  using System;
4
5  public class InicArreglo
6  {
7      public static void Main( string[] args )
8      {
9          // revisa el número de argumentos de línea de comandos
10         if ( args.Length != 3 )
11             Console.WriteLine(
12                 "Error: por favor reintroduzca todo el comando, incluyendo\n" +
13                 "un tamaño de arreglo, valor inicial e incremento." );
14         else
15         {
16             // obtiene tamaño del arreglo del primer argumento de línea de comandos
17             int longitudArreglo = Convert.ToInt32( args[ 0 ] );
18             int[] arreglo = new int[ longitudArreglo ]; // crea el arreglo
19
20             // obtiene valor inicial e incremento del argumento de línea de comandos
21             int valorInicial = Convert.ToInt32( args[ 1 ] );
22             int incremento = Convert.ToInt32( args[ 2 ] );
23
24             // calcula el valor para cada elemento del arreglo
25             for ( int contador = 0; contador < arreglo.Length; contador++ )
26                 arreglo[ contador ] = valorInicial + incremento * contador;
27
28             Console.WriteLine( "{0}{1,8}", "Índice", "Valor" );
29
30             // muestra subíndice y valor del arreglo
31             for ( int contador = 0; contador < arreglo.Length; contador++ )
32                 Console.WriteLine( "{0,5}{1,8}", contador, arreglo[ contador ] );
33         } // fin de else

```

Figura 8.23 | Uso de argumentos de línea de comandos para inicializar un arreglo. (Parte I de 2).


```

34     } // fin de Main
35 } // fin de la clase InicArreglo

```

C:\Ejemplos\cap08\fig08_21>InicArreglo.exe

Error: por favor reintroduzca todo el comando, incluyendo un tamaño de arreglo, valor inicial e incremento.

C:\Ejemplos\cap08\fig08_21>InicArreglo.exe 5 0 4

Índice	Valor
0	0
1	4
2	8
3	12
4	16

C:\Ejemplos\cap08\fig08_21>InicArreglo.exe 10 1 2

Índice	Valor
0	1
1	3
2	5
3	7
4	9
5	11
6	13
7	15
8	17
9	19

Figura 8.23 | Uso de argumentos de línea de comandos para inicializar un arreglo. (Parte 2 de 2).

La figura 8.23 utiliza tres argumentos de línea de comandos para inicializar un arreglo. Cuando se ejecuta la aplicación, si `args.Length` no es 3, la aplicación imprime un mensaje de error y termina (líneas 10-13). En caso contrario, las líneas 16-32 inicializan e imprimen en pantalla el arreglo, con base en los valores de los argumentos de línea de comandos.

Estos argumentos están disponibles para `Main` como objetos `string` en `args`. La línea 17 obtiene `args[ 0 ]` (un objeto `string` que especifica el tamaño del arreglo) y lo convierte en un valor `int`, que la aplicación utiliza para crear el arreglo en la línea 18. El método `static ToInt32` de la clase `Convert` convierte su argumento `string` en un `int`.

Las líneas 21-22 convierten los argumentos de línea de comandos `args[ 1 ]` y `args[ 2 ]` en valores `int` y los almacena en `valorInicial` e `incremento`, respectivamente. Las líneas 25-26 calculan el valor para cada elemento del arreglo.

La salida de la primera ejecución de ejemplo indica que la aplicación recibió un número insuficiente de argumentos de línea de comandos. La segunda ejecución de ejemplo utiliza los argumentos de línea de comandos 5, 0 y 4 para especificar el tamaño del arreglo (5), el valor del primer elemento (0) y el incremento de cada valor en el arreglo (4), respectivamente. La salida correspondiente indica que estos valores crearon un arreglo que contiene los enteros 0, 4, 8, 12 y 16. La salida de la tercera ejecución de ejemplo ilustra que los argumentos de línea de comando 10, 1 y 2 producen un arreglo cuyos 10 elementos son los enteros impares no negativos del 1 al 19.

8.14 (Opcional) Caso de estudio de ingeniería de software: colaboración entre los objetos en el sistema ATM

Cuando dos objetos se comunican entre sí para realizar una tarea, se dice que *colaboran*. Una *colaboración* es cuando un objeto de una clase envía un *mensaje* a un objeto de otra clase. En C#, los mensajes se envían a

través de llamadas a métodos. En esta sección nos concentramos en las colaboraciones (interacciones) entre los objetos en nuestro sistema ATM.

En la sección 7.15 determinamos muchas de las operaciones de las clases en nuestro sistema. En esta sección nos concentramos en los mensajes que invocan a estas operaciones. Para identificar las colaboraciones en el sistema, regresamos al documento de requerimientos de la sección 3.10. Recuerde que este documento especifica las actividades que ocurren durante una sesión con el ATM (por ejemplo, autenticar un usuario, realizar transacciones). Los pasos que se utilizan para describir la forma en que el sistema debe realizar cada una de estas tareas son nuestra primera indicación de las colaboraciones en nuestro sistema. A medida que avancemos a través de estas últimas secciones del Caso de estudio de ingeniería de software, tal vez descubramos colaboraciones adicionales.

Identificar las colaboraciones en un sistema

Para empezar a identificar las colaboraciones en el sistema, leeremos con cuidado las secciones del documento de requerimientos que especifican lo que debe hacer el ATM para autenticar a un usuario y para realizar cada tipo de transacción. Para cada acción o paso descrito en el documento de requerimientos, decidimos qué objetos en nuestro sistema deben interactuar para lograr el resultado deseado. Identificamos un objeto como el emisor (es decir, el objeto que envía el mensaje) y otro como el receptor (es decir, el objeto que ofrece esa operación a los clientes de la clase). Después seleccionamos una de las operaciones del objeto receptor (identificadas en la sección 7.15) que el objeto emisor debe invocar para producir el comportamiento apropiado. Por ejemplo, el ATM muestra un mensaje de bienvenida cuando está inactivo. Sabemos que un objeto de la clase `Pantalla` muestra un mensaje al usuario a través de su operación `MostrarMensaje`. Por ende, decidimos que el sistema puede mostrar un mensaje de bienvenida si empleamos una colaboración entre el ATM y la `Pantalla`, en donde el ATM envía un mensaje `MostrarMensaje` a la `Pantalla` mediante la invocación de la operación `MostrarMensaje` de la clase `Pantalla`. [Nota: para evitar repetir la frase “un objeto de la clase...”, nos referiremos a cada objeto sólo utilizando su nombre de clase, precedido por un artículo (por ejemplo, “un”, “una”, “el” o “la”); por ejemplo, “el ATM” hace referencia a un objeto de la clase `ATM`.]

La figura 8.24 lista las colaboraciones que pueden derivarse del documento de requerimientos. Para cada objeto emisor, listamos las colaboraciones en el orden en el que se describen en el documento de requerimientos. Listamos cada colaboración en la que se involucre un emisor único, un mensaje y un receptor sólo una vez, aun y cuando la colaboración puede ocurrir varias veces durante una sesión con el ATM. Por ejemplo, la primera fila en la figura 8.24 indica que el objeto `ATM` colabora con el objeto `Pantalla` cada vez que el ATM necesita mostrar un mensaje al usuario.

Consideraremos las colaboraciones en la figura 8.24. Antes de permitir que un usuario realice transacciones, el ATM debe pedirle que introduzca un número de cuenta y que después introduzca un NIP. Para realizar cada una de estas tareas envía un mensaje a través de `MostrarMensaje` a la `Pantalla`. Ambas acciones se refieren a la misma colaboración entre el ATM y la `Pantalla`, que ya se listan en la figura 8.24. El ATM obtiene la entrada en respuesta a un indicador, mediante el envío de un mensaje `ObtenerEntrada` al `TecLado`. A continuación, el ATM debe determinar si el número de cuenta especificado por el usuario y el NIP concuerdan con los de una cuenta en la base de datos. Para ello envía un mensaje `AutenticarUsuario` a la `BaseDatosBanco`. Recuerde que `BaseDatosBanco` no puede autenticar a un usuario en forma directa; sólo la `Cuenta` del usuario (es decir, la `Cuenta` que contiene el número de cuenta especificado por el usuario) puede acceder al NIP del usuario para autenticarlo. Por lo tanto, la figura 8.24 lista una colaboración en la que `BaseDatosBanco` envía un mensaje `ValidarNIP` a una `Cuenta`.

Una vez autenticado el usuario, el ATM muestra el menú principal enviando una serie de mensajes `MostrarMensaje` a la `Pantalla` y obtiene la entrada que contiene una selección de menú; para ello envía un mensaje `ObtenerEntrada` al `TecLado`. Ya hemos tomado en cuenta estas colaboraciones. Una vez que el usuario selecciona un tipo de transacción a realizar, el ATM ejecuta la transacción enviando un mensaje `Execute` a un objeto de la clase de transacción apropiada (es decir, un objeto `SolicitudSaldo`, `Retiro` o `Deposito`). Por ejemplo, si el usuario elige realizar una solicitud de saldo, el ATM envía un mensaje `Execute` a un objeto `SolicitudSaldo`.

Un análisis más a fondo del documento de requerimientos revela las colaboraciones involucradas en la ejecución de cada tipo de transacción. Un objeto `SolicitudSaldo` extrae la cantidad de dinero disponible en la cuenta del usuario, al enviar un mensaje `ObtenerSaldoDisponible` al objeto `BaseDatosBanco`, el cual envía un mensaje `get` a la propiedad `SaldoDisponible` de un objeto `Cuenta` para acceder al saldo disponible. De manera similar, el objeto `SolicitudSaldo` extrae la cantidad de dinero depositado al enviar un mensaje `ObtenerSaldo-`

Un objeto de la clase...	envía el mensaje...	a un objeto de la clase...
ATM	MostrarMensaje	Pantalla
	ObtenerEntrada	Tecclado
	AutenticarUsuario	BaseDatosBanco
	Execute	SolicitudSaldo
	Execute	Retiro
	Execute	Deposito
SolicitudSaldo	ObtenerSaldoDisponible	BaseDatosBanco
	ObtenerSaldoTotal	BaseDatosBanco
	MostrarMensaje	Pantalla
Retiro	MostrarMensaje	Pantalla
	ObtenerEntrada	Tecclado
	ObtenerSaldoDisponible	BaseDatosBanco
	HaySuficienteEfectivoDisponible	DispensadorEfectivo
	Cargar	BaseDatosBanco
	DispensarEfectivo	DispensadorEfectivo
Deposito	MostrarMensaje	Pantalla
	ObtenerEntrada	Tecclado
	SeRecibioSobreDeposito	RanuraDeposito
	Abonar	BaseDatosBanco
BaseDatosBanco	ValidarNIP	Cuenta
	SaldoDisponible (get)	Cuenta
	SaldoTotal (get)	Cuenta
	Cargar	Cuenta
	Abonar	Cuenta

Figura 8.24 | Colaboraciones en el sistema ATM.

Total al objeto BaseDatosBanco, el cual envía un mensaje get a la propiedad SaldoTotal de un objeto Cuenta para acceder al saldo total depositado. Para mostrar en pantalla ambas cantidades del saldo del usuario al mismo tiempo, el objeto SolicitudSaldo envía mensajes MostrarMensaje a la Pantalla.

Un objeto Retiro envía mensajes MostrarMensaje a la Pantalla para mostrar un menú de montos estándar de retiro (es decir, \$20, \$40, \$60, \$100, \$200). El objeto Retiro envía un mensaje ObtenerEntrada al Tecclado para obtener la selección del menú elegida por el usuario. A continuación, el objeto Retiro determina si el monto de retiro solicitado es menor o igual al saldo de la cuenta del usuario. Para obtener el monto de dinero disponible en la cuenta del usuario, el objeto Retiro envía un mensaje ObtenerSaldoDisponible al objeto BaseDatosBanco. Después el objeto Retiro evalúa si el dispensador contiene suficiente efectivo enviando un mensaje HaySuficienteEfectivoDisponible al DispensadorEfectivo. Un objeto Retiro envía un mensaje Cargar al objeto BaseDatosBanco para reducir el saldo de la cuenta del usuario. El objeto BaseDatosBanco envía a su vez el mismo mensaje al objeto Cuenta apropiado. Recuerde que al hacer un cargo a una Cuenta se reduce tanto el saldo total como el saldo disponible. Para dispensar la cantidad solicitada de efectivo, el objeto Retiro envía un mensaje DispensarEfectivo al objeto DispensadorEfectivo. Por último, el objeto Retiro envía un mensaje MostrarMensaje a la Pantalla, instruyendo al usuario para que tome el efectivo.

Para responder a un mensaje Execute, un objeto Deposito primero envía un mensaje MostrarMensaje a la Pantalla para pedir al usuario que introduzca un monto a depositar. El objeto Deposito envía un mensaje ObtenerEntrada al Tecclado para obtener la entrada del usuario. Después, el objeto Deposito envía un mensaje MostrarMensaje a la Pantalla para pedir al usuario que inserte un sobre de depósito. Para determinar si la ranura de depósito recibió un sobre de depósito entrante, el objeto Deposito envía un mensaje SeRecibioSobreDeposito al objeto RanuraDeposito. El objeto Deposito actualiza la cuenta del usuario enviando un mensaje Abonar al objeto BaseDatosBanco, el cual a su vez envía un mensaje Abonar al objeto Cuenta del usuario. Recuerde que al abonar a una Cuenta se incrementa el saldo total, pero no el saldo disponible.

Diagramas de interacción

Ahora que identificamos un conjunto de posibles colaboraciones entre los objetos en nuestro sistema ATM, modelaremos en forma gráfica estas interacciones. UML cuenta con varios tipos de *diagramas de interacción*, que para modelar el comportamiento de un sistema modelan la forma en que los objetos interactúan entre sí. El *diagrama de comunicación* enfatiza *cuáles objetos* participan en las colaboraciones. [Nota: los diagramas de comunicación se llamaban *diagramas de colaboración* en versiones anteriores de UML.] Al igual que el diagrama de comunicación, el *diagrama de secuencia* muestra las colaboraciones entre los objetos, pero enfatiza *cuándo* se deben enviar los mensajes entre los objetos.

Diagramas de comunicación

La figura 8.25 muestra un diagrama de comunicación que modela la forma en que el ATM ejecuta una *SolicitudSaldo*. Los objetos se modelan en UML como rectángulos que contienen nombres de la forma *nombreObjeto : NombreClase*. En este ejemplo, que involucra sólo a un objeto de cada tipo, descartamos el nombre del objeto y listamos sólo un signo de dos puntos (:) seguido del nombre de la clase. Se recomienda especificar el nombre de cada objeto en un diagrama de comunicación cuando se modelan varios objetos del mismo tipo. Los objetos que se comunican se conectan con líneas sólidas y los mensajes se pasan entre los objetos a lo largo de estas líneas, en la dirección mostrada por las flechas rellenas. El nombre del mensaje, que aparece enseguida de la flecha, es el nombre de una operación (es decir, un método) que pertenece al objeto receptor; considere el nombre como un servicio que el objeto receptor proporciona a los objetos emisores (sus “clientes”).

La flecha rellena en la figura 8.25 representa un mensaje (o *llamada síncrona*) en UML y una llamada a un método en C#. Esta flecha indica que el flujo de control va desde el objeto emisor (el ATM) hasta el objeto receptor (una *SolicitudSaldo*). Como ésta es una llamada síncrona, el objeto emisor no puede enviar otro mensaje, ni hacer cualquier otra cosa, hasta que el objeto receptor procese el mensaje y devuelva el control (y posiblemente un valor de retorno) al objeto emisor. El emisor sólo espera. Por ejemplo, en la figura 8.25 el objeto ATM llama al método *Execute* de un objeto *SolicitudSaldo* y no puede enviar otro mensaje sino hasta que *Execute* termine y devuelva el control al objeto ATM. [Nota: si ésta fuera una *llamada asíncrona*, representada por una flecha, el objeto emisor no tendría que esperar a que el objeto receptor devolviera el control; continuaría enviando mensajes adicionales inmediatamente después de la llamada asíncrona. Dichas llamadas se encuentran fuera del alcance de este libro.]

Secuencia de mensajes en un diagrama de comunicación

La figura 8.26 muestra un diagrama de comunicación que modela las interacciones entre los objetos en el sistema, cuando se ejecuta un objeto de la clase *SolicitudSaldo*. Asumimos que el atributo *numeroCuenta* del objeto contiene el número de cuenta del usuario actual. Las colaboraciones en la figura 8.26 empiezan después de que el objeto ATM envía un mensaje *Execute* a un objeto *SolicitudSaldo* (es decir, la interacción modelada en la figura 8.25). El número a la izquierda del nombre de un mensaje indica el orden en el que éste se pasa. La *secuencia de mensajes* en un diagrama de comunicación progresa en orden numérico, de menor a mayor. En este diagrama, la numeración comienza con el mensaje 1 y termina con el mensaje 3. El objeto *SolicitudSaldo* envía primero un mensaje *ObtenerSaldoDisponible* al objeto *BaseDatosBanco* (mensaje 1), después envía un mensaje *ObtenerSaldoTotal* al objeto *BaseDatosBanco* (mensaje 2). Dentro de los paréntesis que van después del nombre de un mensaje, podemos especificar una lista separada por comas de los nombres de los argumentos que se envían con el mensaje (es decir, los argumentos en una llamada a un método en C#); el objeto *SolicitudSaldo* pasa el atributo *numeroCuenta* con sus mensajes al objeto *BaseDatosBanco* para indicar de cuál objeto *Cuenta* se extraerá la información del saldo. En la figura 7.22 vimos que las operaciones *ObtenerSaldoDisponible* y *ObtenerSaldoTotal* de la clase *BaseDatosBanco* requieren cada uno de ellos un parámetro para identificar una cuenta. El objeto *SolicitudSaldo* muestra a continuación el saldo disponible y el saldo total al usuario; para

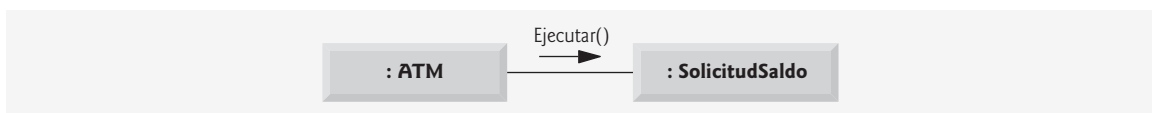


Figura 8.25 | Diagrama de comunicación del ATM, ejecutando una solicitud de saldo.

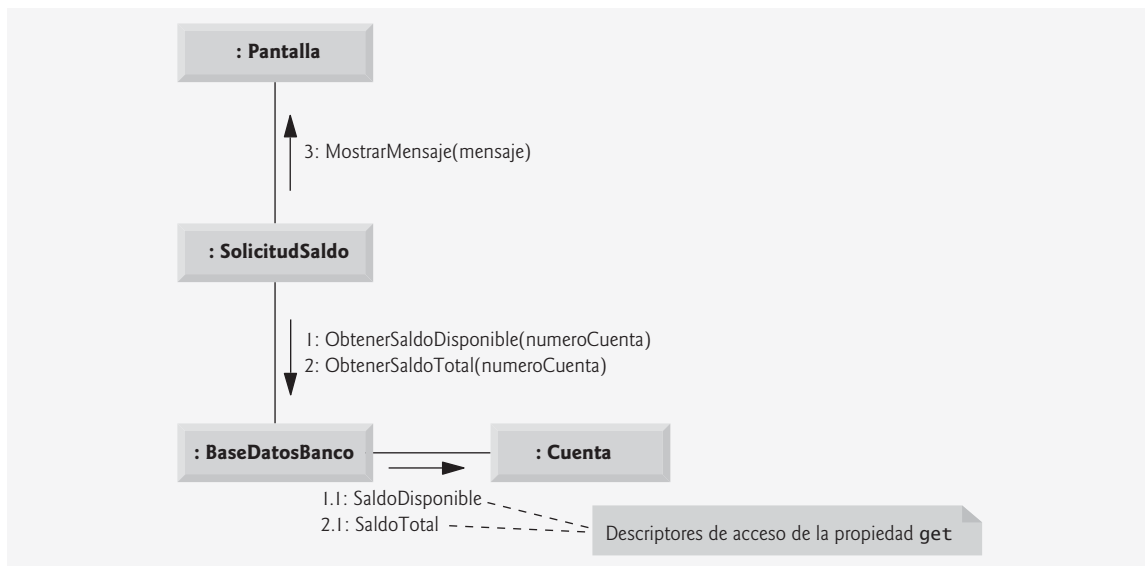


Figura 8.26 | Diagrama de comunicación para ejecutar una *SolicitudSaldo*.

ello pasa un mensaje *MostrarMensaje* a la *Pantalla* (mensaje 3) que incluye un parámetro, el cual indica el mensaje a mostrar.

Observe que la figura 8.26 modela dos mensajes adicionales que se pasan del objeto *BaseDatosBanco* a un objeto *Cuenta* (mensaje 1.1 y mensaje 2.1). Para proveer al ATM los dos saldos de la *Cuenta* del usuario (según lo solicitado por los mensajes 1 y 2), el objeto *BaseDatosBanco* debe enviar mensajes *get* a las propiedades *SaldoDisponible* y *SaldoTotal* del objeto *Cuenta*. Un mensaje que se pasa dentro del manejo de otro mensaje se llama *mensaje anidado*. UML recomienda utilizar un esquema de numeración decimal para indicar mensajes anidados. Por ejemplo, el mensaje 1.1 es el primer mensaje anidado en el mensaje 1; el objeto *BaseDatosBanco* envía el mensaje *get* a la propiedad *SaldoDisponible* del objeto *Cuenta* durante el procesamiento de un mensaje *ObtenerSaldoDisponible* por parte del objeto *BaseDatosBanco*. [Nota: Si el objeto *BaseDatosBanco* necesita pasar un segundo mensaje anidado mientras procesa el mensaje 1, el segundo mensaje se numera como 1.2.] Un mensaje puede pasarse sólo cuando se han pasado ya todos los mensajes anidados del mensaje anterior. Por ejemplo, el objeto *SolicitudSaldo* pasa el mensaje 3 a la *Pantalla* sólo hasta que se han pasado los mensajes 2 y 2.1, en ese orden.

El esquema de numeración anidado que se utiliza en los diagramas de comunicación ayuda a aclarar con precisión cuándo y en qué contexto se pasa cada mensaje. Por ejemplo, si numeramos los cinco mensajes de la figura 8.26 usando un esquema de numeración plano (es decir, 1, 2, 3, 4, 5), podría ser posible que alguien que viera el diagrama no pudiera determinar que el objeto *BaseDatosBanco* pasa el mensaje *get* a la propiedad *SaldoDisponible* de un objeto *Cuenta* (mensaje 1.1) *durante* el procesamiento del mensaje 1 por parte del objeto *BaseDatosBanco*, en vez de hacerlo *después* de completar el procesamiento del mensaje 1. Los números decimales anidados hacen ver que el mensaje *get* (mensaje 1.1) se pasa a la propiedad *SaldoDisponible* de un objeto *Cuenta* dentro del manejo del mensaje *ObtenerSaldoDisponible* (mensaje 1) por parte del objeto *BaseDatosBanco*.

Diagramas de secuencia

Los diagramas de comunicación enfatizan los participantes en las colaboraciones, pero modelan su sincronización de una forma bastante extraña. Un diagrama de secuencia ayuda a modelar la sincronización de las colaboraciones con más claridad. La figura 8.27 muestra un diagrama de secuencia que modela la secuencia de las interacciones que ocurren cuando se ejecuta un *Retiro*. La línea punteada que se extiende hacia abajo desde el rectángulo de un objeto es la *línea de vida* de ese objeto, la cual representa la evolución en el tiempo. Por lo general, las acciones ocurren a lo largo de la línea de vida de un objeto, en orden cronológico de arriba hacia abajo; una acción cerca de la parte superior ocurre antes que una cerca de la parte inferior.

El paso de mensajes en los diagramas de secuencia es similar al paso de mensajes en los diagramas de comunicación. Una flecha con punta rellena, que se extiende desde el objeto emisor hasta el objeto receptor, representa un mensaje entre dos objetos. La punta de flecha apunta a una activación en la línea de vida del objeto receptor. Una *activación*, que se muestra como un rectángulo vertical delgado, indica que se está ejecutando un objeto. Cuando un objeto devuelve el control, un mensaje de retorno (representado como una línea punteada con una punta de flecha) se extiende desde la activación del objeto que devuelve el control hasta la activación del objeto que envió originalmente el mensaje. Para eliminar el desorden, omitimos las flechas de los mensajes de retorno; UML permite esta práctica para que los diagramas sean más legibles. Al igual que los diagramas de comunicación, los diagramas de secuencia pueden indicar parámetros de mensaje entre los paréntesis que van después del nombre de un mensaje.

La secuencia de mensajes de la figura 8.27 empieza cuando un objeto *Retiro* pide al usuario que seleccione un monto de retiro; para ello envía un mensaje *MostrarMensaje* a la *Pantalla*. Después el objeto *Retiro* envía un mensaje *ObtenerEntrada* al *Teclado*, el cual obtiene los datos de entrada del usuario. En el diagrama de la figura 6.22 ya hemos modelado la lógica de control involucrada en un objeto *Retiro*, por lo que no mostraremos esta lógica en el diagrama de secuencia de la figura 8.27. En vez de ello modelaremos el escenario para el mejor

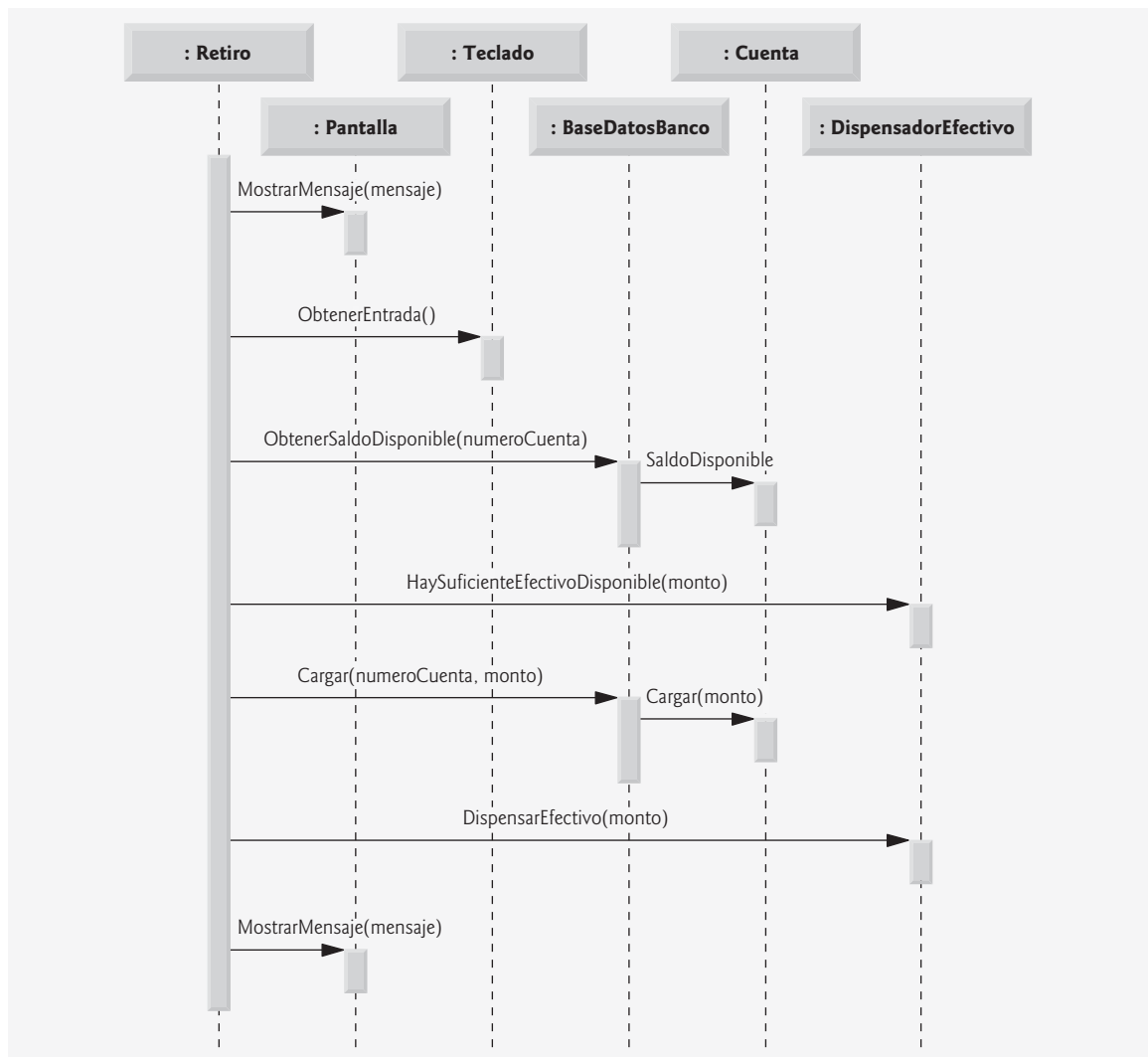


Figura 8.27 | Diagrama de secuencia que modela la ejecución de un *Retiro*.

caso, en el cual el saldo de la cuenta del usuario es mayor o igual al monto de retiro seleccionado, y el dispensador de efectivo contiene un monto de efectivo suficiente como para satisfacer la solicitud. Para obtener información acerca de cómo modelar la lógica de control en un diagrama de secuencia, consulte los recursos Web y las lecturas recomendadas que se listan al final de la sección 3.10.

Después de obtener un monto de retiro, el objeto `Retiro` envía un mensaje `ObtenerSaldoDisponible` al objeto `BaseDatosBanco`, el cual a su vez envía un mensaje `get` a la propiedad `SaldoDisponible` del objeto `Cuenta`. Suponiendo que la cuenta del usuario tiene suficiente dinero disponible para permitir la transacción, el objeto `Retiro` envía a continuación un mensaje `HaySuficienteEfectivoDisponible` al objeto `DispensadorEfectivo`. Suponiendo que hay suficiente efectivo disponible, el objeto `Retiro` reduce el saldo de la cuenta del usuario (tanto el saldo total como el saldo disponible) enviando un mensaje `Cargar` al objeto `BaseDatosBanco`. Este objeto responde enviando un mensaje `Cargar` al objeto `Cuenta` del usuario. Por último, el objeto `Retiro` envía un mensaje `DispensarEfectivo` al usuario para extraer el efectivo de la máquina.

Hemos identificado las colaboraciones entre los objetos en el sistema ATM, y modelamos algunas de estas colaboraciones usando los diagramas de interacción de UML: los diagramas de comunicación y los diagramas de secuencia. En la siguiente sección del Caso de estudio de ingeniería de software (sección 9.17), mejoraremos la estructura de nuestro modelo para completar un diseño orientado a objetos preliminar, cuando empecemos a implementar el sistema ATM en C#.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 8.1 Un(a) _____ consiste en que un objeto de una clase envía un mensaje a un objeto de otra clase.
- asociación
 - agregación
 - colaboración
 - composición
- 8.2 ¿Cuál forma de diagrama de interacción es la que enfatiza *qué* colaboraciones se llevan a cabo? ¿Cuál forma enfatiza *cuándo* ocurren las interacciones?
- 8.3 Cree un diagrama de secuencia para modelar las interacciones entre los objetos del sistema ATM, que ocurran cuando se ejecute un `Deposito` con éxito. Explique la secuencia de los mensajes modelados por el diagrama.

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 8.1 c.
- 8.2 Los diagramas de comunicación enfatizan *qué* colaboraciones se llevan a cabo. Los diagramas de secuencia enfatizan *cuándo* ocurren las colaboraciones.
- 8.3 La figura 8.28 presenta un diagrama de secuencia que modela las interacciones entre objetos en el sistema ATM, que ocurren cuando un `Deposito` se ejecuta con éxito. La figura 8.28 indica que un `Deposito` primero envía un mensaje `MostrarMensaje` a la `Pantalla` (para pedir al usuario que introduzca un monto de depósito). A continuación, el `Deposito` envía un mensaje `ObtenerEntrada` al `TecLado` para recibir el monto que el usuario depositará. Después, el `Deposito` pide al usuario que inserte un sobre de depósito; para ello envía un mensaje `MostrarMensaje` a la `Pantalla`. Luego, el `Deposito` envía un mensaje `SeRecibioSobreDeposito` al objeto `RanuraDeposito` para confirmar que el ATM haya recibido el sobre de depósito. Por último, el objeto `Deposito` incrementa el saldo total (pero no el saldo disponible) de la `Cuenta` del usuario, enviando un mensaje `Abonar` al objeto `BaseDatosBanco`. El objeto `BaseDatosBanco` responde enviando el mismo mensaje a la `Cuenta` del usuario.

8.15 Conclusión

En este capítulo empezó nuestra introducción a las estructuras de datos. Aquí exploramos el uso de los arreglos para almacenar datos y extraerlos de listas y tablas de valores. Los ejemplos de este capítulo demostraron cómo declarar arreglos, inicializarlos y hacer referencia a elementos individuales de los arreglos. Se introdujo la instrucción `foreach` como un medio adicional (además de la instrucción `for`) para iterar a través de los arreglos. Le mostramos cómo pasar arreglos a los métodos, y cómo declarar y manipular arreglos multidimensionales. Por último, en este capítulo se demostró cómo escribir métodos que utilizan listas de argumentos de longitud variable, y cómo leer argumentos que se pasan a una aplicación desde la línea de comandos.

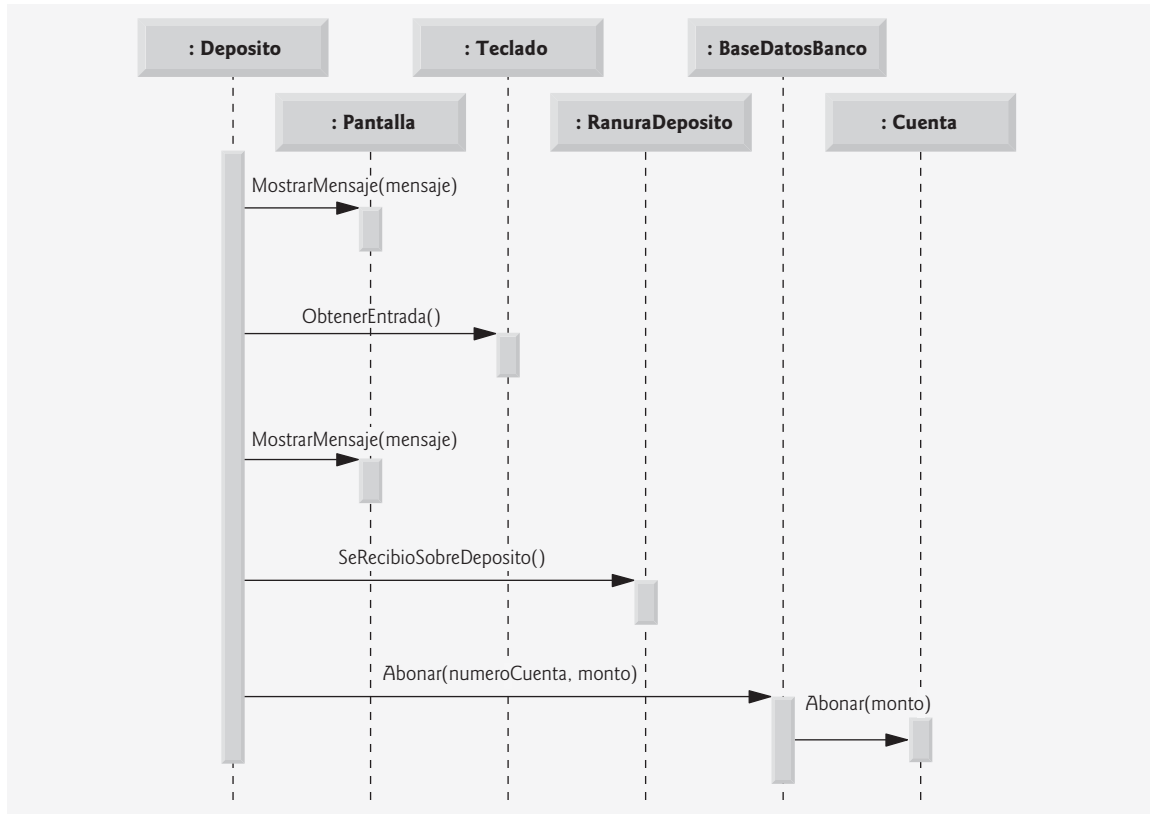


Figura 8.28 | Diagrama de secuencia que modela la ejecución de un Depósito.

En el capítulo 24, Estructuras de datos, continuaremos nuestra cobertura de las estructuras de datos. Presentaremos las estructuras dinámicas de datos, como listas, colas, pilas y árboles, que pueden aumentar y reducir su tamaño a medida que se ejecuta una aplicación. En el capítulo 25, Genéricos, presentaremos una de las nuevas características de C#: los genéricos, que proporcionan los medios para crear modelos generales de métodos y clases que pueden declararse una vez, pero utilizarse con muchos tipos de datos distintos. En el capítulo 26, Colecciones, veremos una introducción a las clases de estructuras de datos que proporciona el .NET Framework, algunas de las cuales utilizan genéricos para permitirle especificar los tipos exactos de objetos que almacenará una estructura de datos específica. Puede utilizar estas estructuras de datos predefinidas en vez de crear las suyas. En el capítulo 26 hablaremos sobre muchas clases de estructuras de datos, incluyendo `Hashtable` y `ArrayList`, que son estructuras de datos similares a los arreglos y pueden aumentar y reducir su tamaño en respuesta al cambio en los requerimientos de almacenamiento de una aplicación. El .NET Framework también proporciona la clase `Array`, que contiene métodos utilitarios para la manipulación de arreglos. El capítulo 26 utiliza varios métodos `static` de la clase `Array` para realizar manipulaciones tales como ordenar y buscar en los datos de un arreglo.

Ya le hemos presentado los conceptos básicos de las clases, los objetos, las instrucciones de control, los métodos y los arreglos. En el capítulo 9 analizaremos con más detalle las clases y los objetos.

9

Clases y objetos: un análisis más detallado

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Acerca del encapsulamiento y el ocultamiento de información.
- Los conceptos de abstracción de datos y tipos de datos abstractos (ADTs).
- Utilizar la palabra clave `this`.
- A utilizar indexadores para acceder a los miembros de una clase.
- Cómo usar variables y métodos `static`.
- Utilizar campos `readonly`.
- Sacar ventaja de las características de administración de memoria de C#.
- Crear una biblioteca de clases.
- Cuándo utilizar el modificador de acceso `internal`.

*En vez de esta absurda
división entre sexos, deberían
clasificar a las personas como
estáticas y dinámicas.*

—Evelyn Waugh

*¿Es éste un mundo en el cual
se deben ocultar las virtudes?*

—William Shakespeare

*Por encima de todo:
hay que ser sinceros
con nosotros mismos.*

—William Shakespeare

*No hay que ser “duros”,
sino simplemente sinceros.*

—Oliver Wendell Holmes, Jr.

- 9.1 Introducción
- 9.2 Caso de estudio de la clase `Tiempo`
- 9.3 Control del acceso a los miembros
- 9.4 Referencias a los miembros del objeto actual mediante `this`
- 9.5 Indexadores
- 9.6 Caso de estudio de la clase `Tiempo`: constructores sobrecargados
- 9.7 Constructores predeterminados y sin parámetros
- 9.8 Composición
- 9.9 Recolección de basura y destructores
- 9.10 Miembros de clase `static`
- 9.11 Variables de instancia `readonly`
- 9.12 Reutilización de software
- 9.13 Abstracción de datos y encapsulamiento
- 9.14 Caso de estudio de la clase `Tiempo`: creación de bibliotecas de clases
- 9.15 Acceso `internal`
- 9.16 Vista de clases y Examinador de objetos
- 9.17 (Opcional) Caso de estudio de ingeniería de software: inicio de la programación de las clases del sistema ATM
- 9.18 Conclusión

9.1 Introducción

En los capítulos anteriores hablamos acerca de las aplicaciones orientadas a objetos, presentamos muchos conceptos básicos y terminología en relación con la programación orientada a objetos (POO) en C#. También hablamos sobre nuestra metodología para desarrollar aplicaciones: seleccionamos variables y métodos apropiados para cada aplicación y especificamos la manera en que un objeto de nuestra clase debería colaborar con los objetos de las clases en la Biblioteca de clases del .NET Framework para realizar los objetivos generales de la aplicación.

En este capítulo analizaremos, de manera más detallada, la creación de clases, el control del acceso a los miembros de una clase y la creación de constructores. Hablaremos sobre la composición: una capacidad que permite a una clase tener referencias a objetos de otras clases como miembros. Analizaremos nuevamente el uso de las propiedades y exploraremos los indexadores como una notación alternativa para acceder a los miembros de una clase. Además, hablaremos a detalle sobre los miembros de clase `static` y las variables de instancia `readonly`. Investigaremos cuestiones como la reutilización de software, la abstracción de datos y el encapsulamiento. Por último, explicaremos cómo organizar las clases en ensamblados, para ayudar en la administración de aplicaciones extensas y promover la reutilización; después mostraremos una relación especial entre clases dentro del mismo ensamblado.

En el capítulo 10, Programación orientada a objetos: herencia, y en el capítulo 11, Polimorfismo, interfaces y sobrecarga de operadores, presentaremos dos tecnologías clave de programación orientada a objetos.

9.2 Caso de estudio de la clase `Tiempo`

Declaración de la clase `Tiempo1`

Nuestro primer ejemplo consiste en dos clases: `Tiempo1` (figura 9.1) y `PruebaTiempo1` (figura 9.2). La clase `Tiempo1` representa la hora del día. La clase `PruebaTiempo1` es una clase de prueba en la que el método `Main` crea un objeto de la clase `Tiempo1` e invoca a sus métodos. El resultado de esta aplicación aparece en la figura 9.2.

La clase `Tiempo1` contiene tres variables de instancia `private` de tipo `int` (figura 9.1, líneas 5-7): `hora`, `minuto` y `segundo`. Estas variables representan la hora en formato de tiempo universal (formato de reloj de 24 horas, en el cual las horas se encuentran en el rango de 0 a 23). La clase `Tiempo1` contiene los métodos `public`

```

1  // Fig. 9.1: Tiempo1.cs
2  // La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.
3  public class Tiempo1
4  {
5      private int hora; // 0 - 23
6      private int minuto; // 0 - 59
7      private int segundo; // 0 - 59
8
9      // establece un nuevo valor de tiempo usando la hora universal; asegura que
10     // los datos permanezcan consistentes al establecer los valores inválidos a cero
11     public void EstablecerTiempo( int h, int m, int s )
12     {
13         hora = ( ( h >= 0 && h < 24 ) ? h : 0 ); // valida la hora
14         minuto = ( ( m >= 0 && m < 60 ) ? m : 0 ); // valida los minutos
15         segundo = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida los segundos
16     } // fin del método EstablecerTiempo
17
18     // convierte en string, en formato de hora universal (HH:MM:SS)
19     public string AStringUniversal()
20     {
21         return string.Format( "{0:D2}:{1:D2}:{2:D2}",
22             hora, minuto, segundo );
23     } // fin del método AStringUniversal
24
25     // convierte en string, en formato de hora estándar (H:MM:SS AM o PM)
26     public override string ToString()
27     {
28         return string.Format( "{0}:{1:D2}:{2:D2} {3}",
29             ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ),
30             minuto, segundo, ( hora < 12 ? "AM" : "PM" ) );
31     } // fin del método ToString
32 } // fin de la clase Tiempo1

```

Figura 9.1 | La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.

EstablecerTiempo (líneas 11-16), AStringUniversal (líneas 19-23) y ToString (líneas 26-31). Estos métodos son los *servicios* public o la *interfaz* public que proporciona la clase a sus clientes.

En este ejemplo, la clase Tiempo1 no declara un constructor, por lo que tiene uno predeterminado que le suministra el compilador. Cada variable de instancia recibe en forma implícita el valor predeterminado 0 para un int. Observe que cuando las variables de instancia se declaran en el cuerpo de la clase, pueden inicializarse mediante la misma sintaxis de inicialización que la de una variable local.

El método EstablecerTiempo (líneas 11-16) es public, declara tres parámetros int y los utiliza para establecer la hora. Una expresión condicional evalúa cada argumento para determinar si el valor se encuentra en un rango especificado. Por ejemplo, el valor de hora (línea 13) debe ser mayor o igual que 0 y menor que 24, ya que el formato de hora universal representa las horas como enteros de 0 a 23 (por ejemplo, la 1 PM es la hora 13 y las 11 PM son la hora 23; medianoche es la hora 0 y mediodía es la hora 12). De manera similar, los valores de minuto y segundo (líneas 14 y 15) deben ser mayores o iguales que 0 y menores que 60. Cualquier valor fuera de rango se establece como 0 para asegurar que un objeto Tiempo1 siempre contenga datos consistentes; esto es, los valores de datos del objeto siempre se mantienen en rango, aun si los valores que se proporcionan como argumentos para el método EstablecerTiempo son incorrectos. En este ejemplo, 0 es un valor consistente para hora, minuto y segundo.

Un valor que se pasa a EstablecerTiempo es correcto si se encuentra dentro del rango permitido para el miembro que va a inicializar. Por lo tanto, cualquier número en el rango de 0 a 23 sería un valor correcto para la hora. Un valor correcto siempre es un valor consistente. Sin embargo, un valor consistente no necesariamente es un valor correcto. Si EstablecerTiempo establece hora a 0 debido a que el argumento que recibió se encontraba fuera del rango, entonces EstablecerTiempo está recibiendo un valor incorrecto y lo hace consistente, para que el objeto permanezca en un estado consistente en todo momento. En este caso, la aplicación podría indicar que el objeto es

incorrecto. En el capítulo 12, Manejo de excepciones, aprenderá técnicas que permitirán a sus clases indicar cuándo se reciben valores incorrectos.



Observación de ingeniería de software 9.1

Los métodos y las propiedades que modifican los valores de variables `private` deben verificar que los nuevos valores que se les pretende asignar sean apropiados. Si no lo son, deben colocar las variables `private` en un estado consistente apropiado.

El método `AStringUniversal` (líneas 19-23) no recibe argumentos y devuelve un objeto `string` en formato de hora universal, el cual consta de seis dígitos: dos para la hora, dos para los minutos y dos para los segundos. Por ejemplo, si la hora es 1:30:07 PM, el método `AStringUniversal` devuelve 13:30:07. La instrucción `return` (líneas 21-22) utiliza el método `static Format` de la clase `string` para devolver un objeto `string` que contiene los valores con formato de hora, minuto y segundo, cada uno con dos dígitos y, en donde sea necesario, un 0 a la izquierda (el cual se especifica con el especificador de formato `D2`, para rellenar el entero con 0s si tiene menos de dos dígitos). El método `Format` es similar al formato de objetos `string` en el método `Console.Write`, sólo que `Format` devuelve un objeto `string` con formato, en vez de mostrarlo en una ventana de consola. El método `AStringUniversal` devuelve el objeto `string` con formato.

El método `ToString` (líneas 26-31) no recibe argumentos y devuelve un objeto `String` en formato de hora estándar, que consiste en los valores de hora, minuto y segundo separados por signos de dos puntos (:), y seguido de un indicador AM o PM (por ejemplo, 1:27:06 PM). Al igual que el método `AStringUniversal`, el método `ToString` utiliza el método `static string Format` para dar formato a los valores de minuto y segundo como valores de dos dígitos con 0s a la izquierda, en caso de ser necesario. La línea 29 utiliza un operador condicional (?) para determinar el valor de hora en la cadena; si hora es 0 o 12 (AM o PM), aparece como 12; en cualquier otro caso, aparece como un valor de 1 a 11. El operador condicional en la línea 30 determina si se devolverá AM o PM como parte del objeto `string`.

En la sección 7.4 vimos que todos los objetos en C# tienen un método `ToString` que devuelve una representación `string` del objeto. Optamos por devolver un objeto `string` que contiene la hora en formato estándar. El método `ToString` se llama en forma implícita cuando la representación `string` de un objeto se imprime en pantalla con un elemento de formato, en una llamada a `Console.Write`. Recuerde que para especificar la representación `string` de una clase, necesitamos declarar el método `ToString` con la palabra clave `override`; la razón de esto se aclarará cuando hablemos sobre la herencia en el capítulo 10.

Uso de la clase `Tiempo1`

Como aprendió en el capítulo 4, cada clase que se declara representa un nuevo tipo en C#. Por lo tanto, después de declarar la clase `Tiempo1` podemos utilizarla como un tipo en las declaraciones como

```
Tiempo1 puestasol; // puestasol puede guardar una referencia a un objeto Tiempo1
```

La clase de la aplicación `PruebaTiempo1` (figura 9.2) utiliza la clase `Tiempo1`. La línea 10 crea un objeto `Tiempo1` y lo asigna a la variable local `tiempo`. Observe que `new` invoca al constructor predeterminado de la clase `Tiempo1`, ya que `Tiempo1` no declara constructores. Las líneas 13-17 imprimen en pantalla la hora, primero en formato universal (mediante la invocación al método `AStringUniversal` en la línea 14) y después en formato estándar (mediante la invocación explícita del método `ToString` de `tiempo` en la línea 16) para confirmar que el objeto `Tiempo1` se haya inicializado en forma apropiada.

```
1 // Fig. 9.2: PruebaTiempo1.cs
2 // Uso de un objeto Tiempo1 en una aplicación.
3 using System;
4
5 public class PruebaTiempo1
6 {
7     public static void Main( string[] args )
8     {
```

Figura 9.2 | Uso de un objeto `Tiempo1` en una aplicación. (Parte I de 2).

```

9      // crea e inicializa un objeto Tiempo1
10     Tiempo1 tiempo = new Tiempo1(); // invoca al constructor de Tiempo1
11
12     // imprime representaciones de cadena de la hora
13     Console.Write( "La hora universal inicial es: " );
14     Console.WriteLine( tiempo.AStringUniversal() );
15     Console.Write( "La hora inicial estándar es: " );
16     Console.WriteLine( tiempo.ToString() );
17     Console.WriteLine(); // imprime una línea en blanco
18
19     // cambia la hora e imprime la hora actualizada
20     tiempo.EstablecerTiempo( 13, 27, 6 );
21     Console.Write( "La hora universal después de EstablecerTiempo es: " );
22     Console.WriteLine( tiempo.AStringUniversal() );
23     Console.Write( "La hora estándar después de EstablecerTiempo es: " );
24     Console.WriteLine( tiempo.ToString() );
25     Console.WriteLine(); // imprime una línea en blanco
26
27     // establece la hora con valores inválidos; imprime la hora actualizada
28     tiempo.EstablecerTiempo( 99, 99, 99 );
29     Console.WriteLine( "Después de tratar de asignar configuraciones inválidas:" );
30     Console.Write( "Hora universal: " );
31     Console.WriteLine( tiempo.AStringUniversal() );
32     Console.Write( "Hora estándar: " );
33     Console.WriteLine( tiempo.ToString() );
34 } // fin de Main
35 } // fin de la clase PruebaTiempo1

```

```

La hora universal inicial es: 00:00:00
La hora inicial estándar es: 12:00:00 AM

La hora universal después de EstablecerTiempo es: 13:27:06
La hora estándar después de EstablecerTiempo es: 1:27:06 PM

Después de tratar de asignar configuraciones inválidas:
Hora universal: 00:00:00
Hora estándar: 12:00:00 AM

```

Figura 9.2 | Uso de un objeto Tiempo1 en una aplicación. (Parte 2 de 2).

La línea 20 invoca al método `EstablecerTiempo` del objeto `tiempo` para cambiar la hora. Después, las líneas 21-25 imprimen en pantalla la hora otra vez en ambos formatos, para confirmar que se haya ajustado en forma apropiada.

Para ilustrar que el método `EstablecerTiempo` mantiene el objeto en un estado consistente, la línea 28 llama al método `EstablecerTiempo` con los argumentos inválidos de 99 para la hora, el minuto y el segundo. Las líneas 29-33 imprimen nuevamente el tiempo en ambos formatos, para confirmar que `EstablecerTiempo` mantenga el estado consistente del objeto, y después la aplicación se termina. Las últimas dos líneas de la salida de la aplicación muestran que el tiempo se reestablece a medianoche (el valor inicial de un objeto `Tiempo1`) si tratamos de establecer el tiempo con valores fuera de rango.

Notas acerca de la declaración de la clase `Tiempo1`

Es necesario considerar diversas cuestiones sobre el diseño de clases, en relación con la clase `Tiempo1`. Las variables de instancia `hora`, `minuto` y `segundo` se declaran como `private`. La representación de datos que se utilice dentro de la clase no concierne a los clientes de la misma. Por ejemplo, sería perfectamente razonable que `Tiempo1` representara el tiempo internamente como el número de segundos transcurridos a partir de medianoche, o el número de minutos y segundos transcurridos a partir de medianoche. Los clientes podrían usar los mismos métodos y propiedades `public` para obtener los mismos resultados, sin tener que preocuparse por lo anterior.



Observación de ingeniería de software 9.2

Las clases simplifican la programación, ya que el cliente sólo puede utilizar los miembros `public` expuestos por la clase. Dichos miembros por lo general están orientados a los clientes, en vez de estar orientados a la implementación. Los clientes nunca se percatan de (ni se involucran en) la implementación de una clase; por lo general se preocupan acerca de lo que ésta hace, pero no cómo lo hace. (Desde luego que los clientes se encargan de que la clase opere en forma correcta y eficiente.)



Observación de ingeniería de software 9.3

Las interfaces cambian con menos frecuencia que las implementaciones. Cuando cambia una implementación, el código dependiente de esa ella debe cambiar de manera acorde. El ocultamiento de la implementación reduce la posibilidad de que otras partes de la aplicación dependan de los detalles de la implementación de la clase.

9.3 Control del acceso a los miembros

Los modificadores de acceso `public` y `private` controlan el acceso a las variables y métodos de una clase (en la sección 9.15 y en el capítulo 10, presentaremos los modificadores de accesos adicionales `internal` y `protected`, respectivamente). Como dijimos en la sección 9.2, el principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que proporciona (la interfaz pública de la clase). Los clientes de la clase no necesitan preocuparse por la forma en que ésta realiza sus tareas. Por esta razón, las variables y métodos `private` de la clase (es decir, los detalles de implementación de la clase) no son directamente accesibles para los clientes.

La figura 9.3 demuestra que los miembros de la clase `private` no son directamente accesibles fuera de la clase. Las líneas 9-11 tratan de acceder en forma directa a las variables de instancia `private` `hora`, `minuto` y `segundo` del objeto `tiempo` de `Tiempo1`. Al compilar esta aplicación, el compilador genera mensajes de error que indican que estos miembros `private` no son accesibles. [Nota: esta aplicación asume que se utiliza la clase `Tiempo1` de la figura 9.1.]



Error común de programación 9.1

Cuando un método que no es miembro de una clase trata de acceder a uno `private` de esa clase, se produce un error de compilación.

```

1 // Fig. 9.3: PruebaAccesoMiembros.cs
2 // Los miembros privados de la clase Tiempo1 no son accesibles.
3 public class PruebaAccesoMiembros
4 {
5     public static void Main( string[] args )
6     {
7         Tiempo1 tiempo = new Tiempo1(); // crea e inicializa un objeto Tiempo1
8
9         tiempo.hora = 7; // error: hora tiene acceso privado en Tiempo1
10        tiempo.minuto = 15; // error: minuto tiene acceso privado en Tiempo1
11        tiempo.segundo = 30; // error: segundo tiene acceso privado en Tiempo1
12    } // fin de Main
13 } // fin de la clase PruebaAccesoMiembros

```

Lista de errores					
3 errores 0 advertencias 0 mensajes					
	Descripción	Archivo	Línea	Columna	Proyecto
1	'Tiempo1.hora' no es accesible debido a su nivel de protección	PruebaAccesoMiembros	9	14	fig09_03
2	'Tiempo1.minuto' no es accesible debido a su nivel de protección	PruebaAccesoMiembros	10	14	fig09_03
3	'Tiempo1.segundo' no es accesible debido a su nivel de protección	PruebaAccesoMiembros	11	14	fig09_03

Figura 9.3 | Los miembros privados de la clase `Tiempo1` no son accesibles.

Observe que los miembros de una clase (por ejemplo, los métodos y las variables de instancia) no necesitan declararse explícitamente como `private`. Si el miembro de una clase no se declara con un modificador de acceso, tiene acceso `private` de manera predeterminada. Nosotros siempre declaramos los miembros `private` en forma explícita.

9.4 Referencias a los miembros del objeto actual mediante this

Cada objeto puede acceder a una referencia a sí mismo mediante la palabra clave `this` (también conocida como *referencia*). Cuando se hace una llamada a un método no `static` para un objeto específico, el cuerpo del método utiliza en forma implícita la palabra clave `this` para hacer referencia a las variables de instancia y los demás métodos del objeto. Como verá en la figura 9.4, puede utilizar también la palabra clave `this` *explícitamente* en el cuerpo de un método no `static`. Las secciones 9.5 y 9.6 muestran dos usos más interesantes de la palabra clave `this`; la sección 9.10 explica por qué no puede usarse esta palabra clave en un método `static`.

Ahora demostraremos el uso implícito y explícito de la referencia `this` para permitir al método `Main` de la clase `PruebaThis` que muestre en pantalla los datos `private` de un objeto de la clase `TiempoSimple` (figura 9.4). Por cuestión de brevedad, declaramos dos clases en un archivo; la clase `PruebaThis` se declara en las líneas 5-12 y la clase `TiempoSimple` se declara en las líneas 15-48.

La clase `TiempoSimple` (líneas 15-48) declara tres variables de instancia `private`: `hora`, `minuto` y `segundo` (líneas 17-19). El constructor (líneas 24-29) recibe tres argumentos `int` para inicializar un objeto `TiempoSimple`. Observe que para el constructor utilizamos nombres de parámetros idénticos a los nombres de las variables de instancia de la clase (líneas 17-19). No recomendamos esta práctica, pero lo hicimos aquí para ocultar las variables de instancia correspondientes y así poder ilustrar el uso explícito de la referencia `this`. En la sección 7.11 vimos que si un método contiene una variable local con el mismo nombre que el de un campo, hará referencia a la variable local y no al campo. En este caso, la variable local oculta el campo en el alcance del método. No obstante, el método puede utilizar la referencia `this` para hacer referencia a la variable de instancia oculta de manera explícita, como se muestra en las líneas 26-28 para las variables de instancia ocultas de `TiempoSimple`.

El método `CrearString` (líneas 32-37) devuelve un objeto `string` creado por una instrucción que utiliza la referencia `this` en forma explícita e implícita. La línea 35 utiliza la referencia `this` en forma explícita para

```

1  // Fig. 9.4: PruebaThis.cs
2  // Uso implícito y explícito de this para hacer referencia a los miembros de un objeto.
3  using System;
4
5  public class PruebaThis
6  {
7      public static void Main( string[] args )
8      {
9          TiempoSimple tiempo = new TiempoSimple( 15, 30, 19 );
10         Console.WriteLine( tiempo.CrearString() );
11     } // fin de Main
12 } // fin de la clase PruebaThis
13
14 // clase TiempoSimple para demostrar la referencia "this"
15 public class TiempoSimple
16 {
17     private int hora; // 0-23
18     private int minuto; // 0-59
19     private int segundo; // 0-59
20
21     // si el constructor usa nombres de parámetros idénticos a los
22     // nombres de las variables de instancia, se requiere la referencia "this"
23     // para diferenciar entre ellos
24     public TiempoSimple( int hora, int minuto, int segundo )
25     {

```

Figura 9.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto. (Parte 1 de 2).


```

26      this.hora = hora; // establece la variable de instancia hora de este ("this")
      objeto
27      this.minuto = minuto; // establece el minuto de este ("this") objeto
28      this.segundo = segundo; // establece el segundo de este ("this") objeto
29  } // fin del constructor de TiempoSimple
30
31  // usa "this" en forma explícita e implícita para llamar a AStringUniversal
32  public string CrearString()
33  {
34      return string.Format( "{0,24}: {1}\n{2,24}: {3}",
35                          "this.AStringUniversal()", this.AStringUniversal(),
36                          "AStringUniversal()", AStringUniversal() );
37  } // fin del método CrearString
38
39  // convierte a string en formato de hora universal (HH:MM:SS)
40  public string AStringUniversal()
41  {
42      // "this" no se requiere aquí para acceder a las variables de instancia,
43      // porque el método no tiene variables locales con los mismos
44      // nombres que las variables de instancia
45      return string.Format( "{0:D2}:{1:D2}:{2:D2}",
46                          this.hora, this.minuto, this.segundo );
47  } // fin del método AStringUniversal
48  } // fin de la clase TiempoSimple

```

```

this.AStringUniversal(): 15:30:19
AStringUniversal(): 15:30:19

```

Figura 9.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto. (Parte 2 de 2).

llamar al método `AStringUniversal`. La línea 36 la utiliza en forma implícita para llamar al mismo método. Observe que ambas líneas realizan la misma tarea. Por lo general, los programadores no utilizan la referencia `this` en forma explícita para hacer referencia a otros métodos en el objeto actual. Además, observe que la línea 46 en el método `AStringUniversal` utiliza en forma explícita la referencia `this` para acceder a cada variable de instancia. Esto no es necesario aquí, ya que el método no tiene variables locales que oculten las variables de instancia de la clase.



Error común de programación 9.2

A menudo se produce un error lógico cuando un método contiene un parámetro o variable local con el mismo nombre que una variable de instancia de la clase. En tal caso, use la referencia `this` si desea acceder a la variable de instancia de la clase; de no ser así, se hará referencia al parámetro o variable local del método.



Tip de prevención de errores 9.1

Evite los nombres de los parámetros o variables locales que tengan conflicto con los nombres de los campos. Esto ayuda a evitar errores sutiles, difíciles de localizar.

La clase `PruebaThis` (figura 9.4, líneas 5-12) demuestra el uso de la clase `TiempoSimple`. La línea 9 crea una instancia de la clase `TiempoSimple` e invoca a su constructor. La línea 10 invoca al método `CrearString` del objeto y después muestra los resultados en pantalla.



Tip de rendimiento 9.1

Para conservar la memoria, C# mantiene sólo una copia de cada método por clase; todos los objetos de la clase invocan a este método. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase (es decir, las variables no `static`). Cada método de la clase utiliza en forma implícita la referencia `this` para determinar el objeto específico de la clase que se manipulará.

9.5 Indexadores

En el capítulo 4 se introdujeron las propiedades como una forma para acceder a los datos `private` de una clase en forma controlada, a través de los descriptores de acceso `get` y `set`. En ocasiones, una clase encapsula listas de datos como arreglos. Dicha clase puede utilizar la palabra clave `this` para definir miembros de la clase similares a las propiedades, a los cuales se les conoce como *indexadores*; éstos permiten el acceso indexado a las listas de elementos, al estilo de los arreglos. Con los arreglos “convencionales” de C#, el índice (o subíndice) debe ser un valor entero. Un beneficio de los indexadores es que se pueden definir subíndices enteros y no enteros. Por ejemplo, podríamos permitir que el código cliente manipule datos mediante el uso de objetos `string` como índices que representen los nombres o las descripciones de los elementos de datos. Al manipular elementos de arreglos “convencionales” en C#, el operador de acceso al elemento de un arreglo siempre devuelve un valor del mismo tipo; es decir, el tipo del arreglo. Los indexadores son más flexibles: pueden devolver cualquier tipo, incluso uno que sea distinto del de los datos subyacentes.

Aunque el operador de acceso a los elementos de un indexador se utiliza igual que un operador de acceso a los elementos de un arreglo, los indexadores se definen como las propiedades en una clase. A diferencia de las propiedades, para las cuales se puede elegir un nombre de propiedad apropiado, los indexadores deben definirse con la palabra clave `this`. Los indexadores tienen la siguiente forma general:

```
modificadorAcceso tipoRetorno this[ TipoIndice1 nombre1, TipoIndice2 nombre2, ... ]
{
    get
    {
        // utiliza nombre1, nombre2, ... aquí para obtener los datos
    }
    set
    {
        // usa nombre1, nombre2, ... aquí para establecer los datos
    }
}
```

Los parámetros *TipoIndice* que se especifican en los corchetes ([]) son accesibles para los descriptores de acceso `get` y `set`. Estos descriptores de acceso definen cómo se utilizará el índice (o índices) para extraer o modificar el miembro de datos apropiado. Al igual que con las propiedades, el descriptor de acceso `get` del indexador debe devolver (return) un valor de tipo *tipoRetorno* y el descriptor de acceso `set` puede utilizar el parámetro implícito `value` para hacer referencia al valor que debe asignarse al elemento.



Error común de programación 9.3

Declarar los indexadores como `static` es un error de sintaxis.

La aplicación de las figuras 9.5-9.6 contiene dos clases: la clase `Caja` representa un cuadro con una longitud, una anchura y una altura, y la clase `PruebaCaja` demuestra el uso de los indexadores de la clase `Caja`.

```
1 // Fig. 9.5: Caja.cs
2 // La definición de la clase Caja representa una caja con dimensiones de
3 // longitud, anchura y altura con indexadores.
4 public class Caja
5 {
6     private string[] nombres = { "longitud", "anchura", "altura" };
7     private double[] medidas = new double[ 3 ];
8
9     // constructor
10    public Caja( double longitud, double anchura, double altura )
11    {
```

Figura 9.5 | La definición de la clase `Caja` representa una caja con medidas de longitud, anchura y altura con indexadores. (Parte 1 de 2).

```

12     medidas[ 0 ] = longitud;
13     medidas[ 1 ] = anchura;
14     medidas[ 2 ] = altura;
15 }
16
17 // indexador para acceder a medidas por el número de índice entero
18 public double this[ int indice ]
19 {
20     get
21     {
22         // valida índice a obtener
23         if ( ( indice < 0 ) || ( indice >= medidas.Length ) )
24             return -1;
25         else
26             return medidas[ indice ];
27     } // fin de get
28     set
29     {
30         if ( indice >= 0 && indice < medidas.Length )
31             medidas[ indice ] = value;
32     } // fin de set
33 } // fin de indexador numérico
34
35 // indexador para acceder a medidas por sus nombres tipo string
36 public double this[ string nombre ]
37 {
38     get
39     {
40         // localiza elemento a obtener
41         int i = 0;
42         while ( ( i < nombres.Length ) &&
43             ( nombre.ToLower() != nombres[ i ] ) )
44             i++;
45
46         return ( i == nombres.Length ) ? -1 : medidas[ i ];
47     } // fin de get
48     set
49     {
50         // localiza elemento a establecer
51         int i = 0;
52         while ( ( i < nombres.Length ) &&
53             ( nombre.ToLower() != nombres[ i ] ) )
54             i++;
55
56         if ( i != nombres.Length )
57             medidas[ i ] = value;
58     } // fin de set
59 } // fin de indexador de string
60 } // fin de la clase Caja

```

Figura 9.5 | La definición de la clase Caja representa una caja con medidas de longitud, anchura y altura con indexadores. (Parte 2 de 2).

Los miembros de datos private de la clase Caja son el arreglo string llamado nombres (línea 6), que contiene los nombres (es decir, "longitud", "anchura" y "altura") para las medidas de una Caja, y el arreglo double llamado medidas (línea 7), que contiene el tamaño de cada medida. Cada elemento en el arreglo nombres corresponde a un elemento en el arreglo medidas (por ejemplo, medidas[2] contiene la altura de la Caja).

Caja define dos indexadores (líneas 18-33 y 36-59), cada uno de los cuales devuelve (return) un valor `double` que representa el tamaño de la medida especificada por el parámetro del indexador. Los indexadores pueden sobrecargarse de igual forma que los métodos. El primer indexador utiliza un índice `int` para manipular un elemento en el arreglo `medidas`. El segundo indexador utiliza un índice `string`, que representa el nombre de la medida para manipular un elemento en el arreglo `medidas`. Cada indexador devuelve `-1` si su descriptor de acceso `get` encuentra un subíndice inválido. El descriptor de acceso `set` de cada indexador asigna `value` al elemento apropiado de `medidas`, sólo si el índice es válido. Por lo general, un indexador lanza una excepción si recibe un índice inválido. En el capítulo 12, Manejo de excepciones, veremos cómo lanzar excepciones.

Observe que el indexador `string` utiliza una estructura `while` para buscar un `string` que concuerde en el arreglo `nombres` (líneas 42-44 y 52-54). Si hay una concordancia, el indexador manipula el elemento correspondiente en el arreglo `medidas` (líneas 46 y 57).

La clase `PruebaCaja` (figura 9.6) manipula los miembros de datos `private` de la clase `Caja`, a través de los indexadores de `Caja`. La variable local `caja` se declara en la línea 10 y se inicializa con una nueva instancia de la clase `Caja`. Utilizamos el constructor de `Caja` para inicializar `caja` con las medidas 30, 30 y 30. Las líneas 14-16 utilizan el indexador declarado con el parámetro `int` para obtener las tres medidas de `caja`, y las muestra en pantalla mediante `WriteLine`. La expresión `caja[ 0 ]` (línea 14) llama en forma implícita al descriptor de acceso `get` del indexador para obtener el valor de la variable de instancia `private medidas[ 0 ]` de `caja`. De manera similar, la asignación a `caja[ 0 ]` en la línea 20 llama en forma implícita al descriptor de acceso `set` en las líneas 28-32 de la figura 9.5. El descriptor de acceso `set` establece en forma implícita su parámetro `value` a 10 y después establece `medidas[ 0 ]` a `value` (10). Las líneas 24 y 28-30 en la figura 9.6 realizan acciones similares, usando el indexador sobrecargado con un parámetro `string` para manipular los mismos datos.

```

1  // Fig. 9.6: PruebaCaja.cs
2  // Los indexadores proporcionan acceso a los miembros de un objeto Caja.
3  using System;
4
5  public class PruebaCaja
6  {
7      public static void Main( string[] args )
8      {
9          // crea una caja
10         Caja caja = new Caja( 30, 30, 30 );
11
12         // muestra medidas con indexadores numéricos
13         Console.WriteLine( "Se creó una caja con las medidas:" );
14         Console.WriteLine( "caja[ 0 ] = {0}", caja[ 0 ] );
15         Console.WriteLine( "caja[ 1 ] = {0}", caja[ 1 ] );
16         Console.WriteLine( "caja[ 2 ] = {0}", caja[ 2 ] );
17
18         // establece una medida con el indexador numérico
19         Console.WriteLine( "\nSe estableció caja[ 0 ] a 10...\n" );
20         caja[ 0 ] = 10;
21
22         // establece una medida con el indexador string
23         Console.WriteLine( "Se estableció caja[ \"anchura\" ] a 20...\n" );
24         caja[ "anchura" ] = 20;
25
26         // muestra medidas con indexadores string
27         Console.WriteLine( "Ahora la caja tiene las medidas:" );
28         Console.WriteLine( "caja[ \"longitud\" ] = {0}", caja[ "longitud" ] );
29         Console.WriteLine( "caja[ \"anchura\" ] = {0}", caja[ "anchura" ] );
30         Console.WriteLine( "caja[ \"altura\" ] = {0}", caja[ "altura" ] );
31     } // fin del método Main
32 } // fin de la clase PruebaCaja

```

Figura 9.6 | Los indexadores proporcionan acceso a los miembros de un objeto. (Parte I de 2).

```

Se creó una caja con las medidas:
caja[ 0 ] = 30
caja[ 1 ] = 30
caja[ 2 ] = 30

Se estableció caja[ 0 ] a 10...

Se estableció caja[ "anchura" ] a 20...

Ahora la caja tiene las medidas:
caja[ "longitud" ] = 10
caja[ "anchura" ] = 20
caja[ "altura" ] = 30

```

Figura 9.6 | Los indexadores proporcionan acceso a los miembros de un objeto. (Parte 2 de 2).

9.6 Caso de estudio de la clase Tiempo: constructores sobrecargados

Como sabe, puede declarar su propio constructor para especificar cómo deben inicializarse los objetos de una clase. A continuación demostraremos una clase con varios *constructores sobrecargados*, que permiten a los objetos inicializarse de distintas formas. Para sobrecargar los constructores, sólo hay que proporcionar varias declaraciones del constructor con distintas firmas. En la sección 7.12 vimos que el compilador diferencia las firmas en base al número, tipos y orden de los parámetros en cada firma.

Clase Tiempo2 con constructores sobrecargados

De manera predeterminada, las variables de instancia hora, minuto y segundo de la clase Tiempo1 (figura 9.1) se inicializan a sus valores predeterminados de 0 (medianoche en formato de hora universal). La clase Tiempo1 no permite que sus clientes inicialicen la hora con valores específicos distintos de cero. La clase Tiempo2 (figura 9.7) contiene cinco constructores sobrecargados para inicializar de manera conveniente sus objetos en una variedad de formas. Los constructores aseguran que cada objeto Tiempo2 comience en un estado consistente. En esta aplicación, cuatro de los constructores invocan un quinto constructor, el cual a su vez llama al método EstablecerTiempo. El método EstablecerTiempo invoca los descriptores de acceso set de las propiedades Hora, Minuto y Segundo, que aseguran que el valor suministrado para hora se encuentre en el rango de 0 a 23, y que los valores para minuto y segundo se encuentren cada uno en el rango de 0 a 59. Si un valor está fuera de rango, se establece a 0 mediante la propiedad

```

1 // Fig. 9.7: Tiempo2.cs
2 // Declaración de la clase Tiempo2 con constructores sobrecargados.
3 public class Tiempo2
4 {
5     private int hora; // 0 - 23
6     private int minuto; // 0 - 59
7     private int segundo; // 0 - 59
8
9     // constructor de Tiempo2 sin argumentos: inicializa cada variable de instancia
10    // a cero; asegura que los objetos Tiempo2 empiecen en un estado consistente
11    public Tiempo2() : this( 0, 0, 0 ) { }
12
13    // constructor de Tiempo2: se suministra hora, minuto y segundo con valor predet. 0
14    public Tiempo2( int h ) : this( h, 0, 0 ) { }
15
16    // constructor de Tiempo2: se suministran hora y minuto, segundo con valor predet. 0
17    public Tiempo2( int h, int m ) : this( h, m, 0 ) { }
18
19    // constructor de Tiempo2: se suministran hora, minuto y segundo

```

Figura 9.7 | Declaración de la clase Tiempo2 con constructores sobrecargados. (Parte 1 de 3).

```

20 public Tiempo2( int h, int m, int s )
21 {
22     EstablecerTiempo( h, m, s ); // invoca a EstablecerTiempo para validar el tiempo
23 } // fin de constructor de Tiempo2 con tres argumentos
24
25 // constructor de Tiempo2: se suministra otro objeto Tiempo2
26 public Tiempo2( Tiempo2 tiempo )
27 : this( tiempo.Hora, tiempo.Minuto, tiempo.Segundo ) { }
28
29 // establece un nuevo valor de tiempo usando hora universal; asegura que
30 // los datos permanezcan consistentes al establecer los valores inválidos a cero
31 public void EstablecerTiempo( int h, int m, int s )
32 {
33     Hora = h; // establece la propiedad Hora
34     Minuto = m; // establece la propiedad Minuto
35     Segundo = s; // establece la propiedad Segundo
36 } // fin del método EstablecerTiempo
37
38 // Propiedades para obtener y establecer
39 // propiedad que obtiene (get) y establece (set) la hora
40 public int Hora
41 {
42     get
43     {
44         return hora;
45     } // fin de get
46     // hacer la escritura inaccesible fuera de la clase
47     private set
48     {
49         hora = ( ( value >= 0 && value < 24 ) ? value : 0 );
50     } // fin de set
51 } // fin de la propiedad Hora
52
53 // propiedad que obtiene (get) y establece (set) el minuto
54 public int Minuto
55 {
56     get
57     {
58         return minuto;
59     } // fin de get
60     // hacer la escritura inaccesible fuera de la clase
61     private set
62     {
63         minuto = ( ( value >= 0 && value < 60 ) ? value : 0 );
64     } // fin de set
65 } // fin de la propiedad Minuto
66
67 // propiedad que obtiene (get) y establece (set) el segundo
68 public int Segundo
69 {
70     get
71     {
72         return segundo;
73     } // fin de get
74     // hacer la escritura inaccesible fuera de la clase
75     private set
76     {
77         segundo = ( ( value >= 0 && value < 60 ) ? value : 0 );

```

Figura 9.7 | Declaración de la clase Tiempo2 con constructores sobrecargados. (Parte 2 de 3).

```

78     } // fin de set
79 } // fin de la propiedad Segundo
80
81 // convierte a string en formato de hora universal (HH:MM:SS)
82 public string AStringUniversal()
83 {
84     return string.Format(
85         "{0:D2}:{1:D2}:{2:D2}", Hora, Minuto, Segundo );
86 } // fin del método AStringUniversal
87
88 // convierte a string en formato de hora estándar (H:MM:SS AM o PM)
89 public override string ToString()
90 {
91     return string.Format( "{0}:{1:D2}:{2:D2} {3}",
92         ( ( Hora == 0 || Hora == 12 ) ? 12 : Hora % 12 ),
93         Minuto, Segundo, ( Hora < 12 ? "AM" : "PM" ) );
94 } // fin del método ToString
95 } // fin de la clase Tiempo2

```

Figura 9.7 | Declaración de la clase `Tiempo2` con constructores sobrecargados. (Parte 3 de 3).

correspondiente (una vez más se asegura que cada variable de instancia permanezca en un estado consistente). Para invocar el constructor apropiado, el compilador relaciona el número y los tipos de los argumentos especificados en la llamada al constructor con el número y los tipos de los parámetros especificados en la declaración de cada constructor. Observe que la clase `Tiempo2` también proporciona propiedades para cada variable de instancia.

Constructores de la clase `Tiempo2`

La línea 11 declara un *constructor sin parámetros*, es decir, que se invoca sin argumentos. Observe que este constructor tiene un cuerpo vacío, como se indica mediante el conjunto vacío de llaves después del encabezado. En vez de un cuerpo, presentamos un uso de la referencia `this` que se permite sólo en el encabezado del constructor. En la línea 11, el encabezado usual del constructor va seguido de un signo de dos puntos (`:`) y de la palabra clave `this`. La referencia `this` se utiliza en la sintaxis de llamada a un método (junto con los tres argumentos `int`) para invocar al constructor de `Tiempo2` que recibe tres argumentos `int` (líneas 20-23). El constructor sin parámetros pasa los valores de 0 para hora, minuto y segundo al constructor con tres parámetros `int`. El uso de la referencia `this` que se muestra aquí se denomina *inicializador de constructor*. Estos inicializadores son una forma popular de reutilizar el código de inicialización que proporciona uno de los constructores de la clase, en vez de definir código similar en el cuerpo de otro constructor. Utilizamos esta sintaxis en cuatro de los cinco constructores de `Tiempo2` para que la clase sea más fácil de mantener. Si necesitamos cambiar la forma en que se inicializan los objetos de la clase `Tiempo2`, sólo hay que modificar el constructor al que necesitan llamar los demás constructores de la clase. Incluso ese constructor podría no requerir de modificación; simplemente llama al método `TiempoSimple` para realizar la verdadera inicialización, por lo que es posible que los cambios que requiera la clase se localicen en este método.

La línea 14 declara un constructor de `Tiempo2` con un solo parámetro `int` que representa la hora, que se pasa con 0 para minuto y segundo al constructor de las líneas 20-23. La línea 17 declara un constructor `Tiempo2` que recibe dos parámetros `int`, que representan la hora y el minuto, que se pasan con un 0 para segundo al constructor de las líneas 20-23. Al igual que el constructor sin parámetros, cada uno de estos constructores invoca al constructor en las líneas 20-23 para minimizar la duplicación de código. Las líneas 20-23 declaran el constructor `Tiempo2` que recibe tres parámetros `int`, que representan la hora, el minuto y el segundo. Este constructor llama a `EstablecerTiempo` para inicializar las variables de instancia con valores consistentes. A su vez, `EstablecerTiempo` invoca a los descriptores de acceso `set` de las propiedades `Hora`, `Minuto` y `Segundo`.



Error común de programación 9.4

Un constructor puede llamar a los métodos de la clase. Tenga en cuenta que tal vez las variables de instancia no se encuentren aún en un estado consistente, ya que el constructor está en el proceso de inicializar el objeto. El uso de variables de instancia antes de inicializarlas en forma apropiada es un error lógico.

Las líneas 26-27 declaran un constructor de `Tiempo2` que recibe una referencia `Tiempo2` a otro objeto `Tiempo2`. En este caso, los valores del argumento `Tiempo2` se pasan al constructor de tres parámetros en las líneas 20-23 para inicializar `hora`, `minuto` y `segundo`. Observe que la línea 27 podría haber accedido en forma directa a las variables de instancia `hora`, `minuto` y `segundo` del argumento `tiempo` del constructor mediante las expresiones `tiempo.hora`, `tiempo.minuto` y `tiempo.segundo`, aun y cuando `hora`, `minuto` y `segundo` se declaran como variables `private` de la clase `Tiempo2`.



Observación de ingeniería de software 9.4

Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos del segundo objeto (incluyendo los que sean `private`).

Observaciones en relación con los métodos, propiedades y constructores de la clase `Tiempo2`

Observe que se accede a las propiedades de `Tiempo2` a través del cuerpo de la clase. En especial, el método `EstablecerTiempo` asigna valores a las propiedades `Hora`, `Minuto` y `Segundo` en las líneas 33-35, y los métodos `AStringUniversal` y `ToString` usan las propiedades `Hora`, `Minuto` y `Segundo` en la línea 85 y en las líneas 92 y 93, respectivamente. En cada caso, estos métodos podrían haber accedido a los datos `private` de la clase en forma directa, sin necesidad de usar las propiedades. Ahora, considere la acción de cambiar la representación del tiempo, de tres valores `int` (que requieren 12 bytes de memoria) a un solo valor `int` que represente el número total de segundos transcurridos a partir de medianoche (que requiere sólo 4 bytes de memoria). Si hacemos ese cambio, sólo tendrían que cambiar los cuerpos de los métodos que acceden directamente a los datos `private`; en especial, las propiedades individuales `Hora`, `Minuto` y `Segundo`. No habría necesidad de modificar los cuerpos de los métodos `EstablecerTiempo`, `AStringUniversal` o `ToString`, ya que no acceden a los datos `private` en forma directa. Si se diseña la clase de esta forma se reduce la probabilidad de que se produzcan errores de programación al momento de alterar la implementación de la clase.

De manera similar, cada constructor de `Tiempo2` podría escribirse de forma que incluya una copia de las instrucciones apropiadas del método `EstablecerTiempo`. Esto sería un poco más eficiente, ya que se eliminan la llamada extra al constructor y la llamada a `EstablecerTiempo`. No obstante, duplicar las instrucciones en varios métodos o constructores dificulta más el proceso de modificar la representación interna de datos de la clase, además de que aumenta la probabilidad de que haya errores. Si hacemos que los constructores de `Tiempo2` llamen al constructor con tres parámetros (o que incluso llamen a `EstablecerTiempo` directamente), cualquier modificación a la implementación de `EstablecerTiempo` sólo tendrá que hacerse una vez.



Observación de ingeniería de software 9.5

Al implementar un método de una clase, use las propiedades de la clase para acceder a sus datos `private`. Esto simplifica el mantenimiento del código y reduce la probabilidad de errores.

Observe además que la clase `Tiempo2` aprovecha los modificadores de acceso para asegurar que los clientes de la clase utilicen los métodos y propiedades apropiados para acceder a los datos `private`. En especial, las propiedades `Hora`, `Minuto` y `Segundo` declaran los descriptores de acceso `private set` (líneas 47, 61 y 75, respectivamente) para restringir el uso de los descriptores de acceso `set` a los miembros de la clase. Declaramos estos descriptores como `private` por las mismas razones que declaramos las variables de instancia `private`: para simplificar el mantenimiento del código y asegurar que los datos permanezcan en un estado consistente. Aunque los métodos en la clase `Tiempo2` aún tienen todas las ventajas de usar los descriptores de acceso `set` para realizar la validación, los clientes de la clase deben utilizar el método `EstablecerTiempo` para modificar estos datos. Los descriptores de acceso `get` de las propiedades `Hora`, `Minuto` y `Segundo` se declaran en forma implícita como `public`, ya que sus propiedades se declaran `public`; cuando no hay un modificador de acceso antes de un descriptor de acceso `get` o `set`, el descriptor hereda el modificador de acceso que va antes del nombre de la propiedad.

Uso de los constructores sobrecargados de la clase `Tiempo2`

La clase `PruebaTiempo2` (figura 9.8) crea seis objetos `Tiempo2` (líneas 9-14) para invocar a los constructores sobrecargados de `Tiempo2`. La línea 9 muestra que para invocar el constructor sin parámetros (línea 11 de la


```

1  // Fig. 9.8: PruebaTiempo2.cs
2  // Uso de constructores sobrecargados para inicializar objetos Tiempo2.
3  using System;
4
5  public class PruebaTiempo2
6  {
7      public static void Main( string[] args )
8      {
9          Tiempo2 t1 = new Tiempo2(); // 00:00:00
10         Tiempo2 t2 = new Tiempo2( 2 ); // 02:00:00
11         Tiempo2 t3 = new Tiempo2( 21, 34 ); // 21:34:00
12         Tiempo2 t4 = new Tiempo2( 12, 25, 42 ); // 12:25:42
13         Tiempo2 t5 = new Tiempo2( 27, 74, 99 ); // 00:00:00
14         Tiempo2 t6 = new Tiempo2( t4 ); // 12:25:42
15
16         Console.WriteLine( "Se construyó con:\n" );
17         Console.WriteLine( "t1: todos los argumentos con valor predeterminado" );
18         Console.WriteLine( "    {0}", t1.AStringUniversal() ); // 00:00:00
19         Console.WriteLine( "    {0}\n", t1.ToString() ); // 12:00:00 AM
20
21         Console.WriteLine(
22             "t2: se especificó hora; minuto y segundo con valores predeterminados" );
23         Console.WriteLine( "    {0}", t2.AStringUniversal() ); // 02:00:00
24         Console.WriteLine( "    {0}\n", t2.ToString() ); // 2:00:00 AM
25
26         Console.WriteLine(
27             "t3: se especificaron hora y minuto; segundo con valor predeterminado" );
28         Console.WriteLine( "    {0}", t3.AStringUniversal() ); // 21:34:00
29         Console.WriteLine( "    {0}\n", t3.ToString() ); // 9:34:00 PM
30
31         Console.WriteLine( "t4: se especificaron hora, minuto y segundo" );
32         Console.WriteLine( "    {0}", t4.AStringUniversal() ); // 12:25:42
33         Console.WriteLine( "    {0}\n", t4.ToString() ); // 12:25:42 PM
34
35         Console.WriteLine( "t5: se especificaron todos los valores inválidos" );
36         Console.WriteLine( "    {0}", t5.AStringUniversal() ); // 00:00:00
37         Console.WriteLine( "    {0}\n", t5.ToString() ); // 12:00:00 AM
38
39         Console.WriteLine( "t6: se especificó objeto t4 de Tiempo2" );
40         Console.WriteLine( "    {0}", t6.AStringUniversal() ); // 12:25:42
41         Console.WriteLine( "    {0}\n", t6.ToString() ); // 12:25:42 PM
42     } // fin de Main
43 } // fin de la clase PruebaTiempo2

```

Se construyó con:

t1: todos los argumentos con valor predeterminado
 00:00:00
 12:00:00 AM

t2: se especificó hora; minuto y segundo con valores predeterminados
 02:00:00
 2:00:00 AM

t3: se especificaron hora y minuto; segundo con valor predeterminado
 21:34:00
 9:34:00 PM

(continúa...)

Figura 9.8 | Uso de constructores sobrecargados para inicializar objetos Tiempo2. (Parte I de 2).

```

t4: se especificaron hora, minuto y segundo
    12:25:42
    12:25:42 PM

t5: se especificaron todos los valores inválidos
    00:00:00
    12:00:00 AM

t6: se especificó objeto t4 de Tiempo2
    12:25:42
    12:25:42 PM

```

Figura 9.8 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2`. (Parte 2 de 2).

figura 9.7) se coloca un conjunto vacío de paréntesis después del nombre de la clase, cuando se asigna un objeto `Tiempo2` mediante `new`. Las líneas 10-14 de la aplicación demuestran el paso de argumentos a los demás constructores de `Tiempo2`. Para invocar el constructor sobrecargado apropiado, C# relaciona el número y los tipos de los argumentos especificados en la llamada al constructor con el número y los tipos de los parámetros especificados en la declaración de cada constructor. La línea 10 invoca el constructor de la línea 14 de la figura 9.7. La línea 11 invoca el constructor de la línea 17 de la figura 9.7. Las líneas 12-13 invocan el constructor de las líneas 20-23 de la figura 9.7. La línea 14 invoca el constructor de las líneas 26-27 de la figura 9.7. La aplicación muestra en pantalla la representación `string` de cada objeto `Tiempo2` inicializado, para confirmar que cada uno de ellos se haya inicializado en forma apropiada.

9.7 Constructores predeterminados y sin parámetros

Toda clase debe tener cuando menos un constructor. En la sección 4.9 vimos que si no se proporcionan constructores en la declaración de una clase, el compilador crea uno predeterminado que no recibe argumentos cuando se le invoca. En la sección 10.4.1 aprenderá que el constructor predeterminado realiza de manera implícita una tarea especial. Toda clase debe tener cuando menos un constructor.

El compilador no creará un constructor predeterminado para una clase que declare cuando menos un constructor en forma explícita. En este caso, si desea poder invocar al constructor sin argumentos, debe declarar un constructor sin parámetros (como en la línea 11 de la figura 9.7). Al igual que un constructor predeterminado, un constructor sin parámetros se invoca con paréntesis vacíos. Observe que el constructor sin parámetros de `Tiempo2` inicializa en forma explícita un objeto `Tiempo2`; para ello pasa un 0 a cada parámetro del constructor de tres parámetros. Como 0 es el valor predeterminado para las variables de instancia `int`, el constructor sin parámetros en este ejemplo podría omitir el inicializador del constructor. En este caso, cada variable de instancia recibiría su valor predeterminado al momento de crear el objeto. Si omitiéramos el constructor sin parámetros, los clientes de esta clase no podrían crear un objeto `Tiempo2` con la expresión `new Tiempo2()`.



Error común de programación 9.5

Si una clase tiene constructores, pero ninguno de los constructores `public` son sin parámetros, y si una aplicación intenta llamar a un constructor sin parámetros para inicializar un objeto de esa clase, se produce un error de compilación. Se puede llamar a un constructor sin argumentos sólo cuando la clase no tiene constructores (en cuyo caso se llama al constructor predeterminado), o si la clase tiene un constructor `public` sin parámetros.



Error común de programación 9.6

Sólo los constructores pueden tener el mismo nombre que la clase. Declarar un método, propiedad o campo con el mismo nombre que la clase es un error de compilación.

9.8 Composición

Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como *composición* y algunas veces como *relación “tiene un”*. Por ejemplo, un objeto de la clase `RelojAlarma` necesita

saber la hora actual y la hora en la que se supone sonará su alarma, por lo que es razonable incluir dos referencias a objetos `Tiempo` como miembros del objeto `RelojAlarma`.



Observación de ingeniería de software 9.6

La composición es una forma de reutilización de software, en donde una clase tiene como miembros referencias a objetos de otras clases.

Nuestro ejemplo de composición contiene tres clases: `Fecha` (figura 9.9), `Empleado` (figura 9.10) y `PruebaEmpleado` (figura 9.11). La clase `Fecha` (figura 9.9) declara las variables de instancia `mes`, `día` y `año` (líneas 7-9) para representar una fecha. El constructor recibe tres parámetros `int`. La línea 15 invoca en forma implícita el descriptor de acceso `set` de la propiedad `Mes` (líneas 41-50) para validar el mes; un valor fuera de rango se establece en 1 para mantener un estado consistente. La línea 16 utiliza de manera similar la propiedad `Anio` para establecer el año; pero observe que el descriptor de acceso `set` de `Anio` (líneas 28-31) asume que el valor de `año` es correcto y no lo valida. La línea 17 utiliza la propiedad `Día` (líneas 54-78), la cual valida y asigna el valor de `día` con base en el `mes` y `año` actuales (mediante el uso de las propiedades `Mes` y `Anio` para poder obtener los valores de `mes` y `año`). Observe que el orden de inicialización es importante, ya que el descriptor de acceso `set` de la propiedad `Día` valida el valor para `día`, con base en la suposición de que `mes` y `año` son correctos. La línea 66 determina si el día es correcto, con base en el número de días en el `Mes` especificado. Si el día no es correcto, las líneas 69-70 determinan si el `Mes` es Febrero, el día es 29 y el `Anio` es bisiesto. En caso contrario, si el parámetro `value` no contiene un valor correcto para `día`, la línea 75 establece `día` en 1 para mantener la `Fecha` en un estado consistente. Observe que la línea 18 en el constructor muestra en pantalla la referencia `this` como un objeto `string`. Como `this` es una referencia al objeto `Fecha` actual, el método `ToString` del objeto (líneas 81-84) se llama en forma implícita para obtener la representación `string` del objeto.

La clase `Empleado` (figura 9.10) tiene las variables de instancia `primerNombre`, `apellido`, `fechaNacimiento` y `fechaContratacion`. Los miembros `fechaNacimiento` y `fechaContratacion` (líneas 7-8) son referencias

```

1  // Fig. 9.9: Fecha.cs
2  // Declaración de la clase Fecha.
3  using System;
4
5  public class Fecha
6  {
7      private int mes; // 1-12
8      private int dia; // 1-31 dependiendo del mes
9      private int anio; // cualquier año (se podría validar)
10
11     // constructor: usa la propiedad Mes para confirmar el valor apropiado para mes;
12     // usa la propiedad Dia para confirmar el valor apropiado para día
13     public Fecha( int elMes, int elDia, int elAnio )
14     {
15         Mes = elMes; // valida el mes
16         Anio = elAnio; // podría validar el año
17         Dia = elDia; // valida el día
18         Console.WriteLine( "Constructor de objeto Fecha para fecha {0}", this );
19     } // fin del constructor de Fecha
20
21     // propiedad que obtiene (get) y establece (set) el año
22     public int Anio
23     {
24         get
25         {
26             return anio;
27         } // fin de get

```

Figura 9.9 | Declaración de la clase `Fecha`. (Parte I de 2).

```

28     private set // hace la escritura inaccesible fuera de la clase
29     {
30         anio = value; // podría validar
31     } // fin de set
32 } // fin de la propiedad Anio
33
34 // propiedad que obtiene (get) y establece (set) el mes
35 public int Mes
36 {
37     get
38     {
39         return mes;
40     } // fin de get
41     private set // hace la escritura inaccesible fuera de la clase
42     {
43         if ( value > 0 && value <= 12 ) // valida el mes
44             mes = value;
45         else // mes es inválido
46         {
47             Console.WriteLine( "El mes inválido ({0}) se establece en 1.", value );
48             mes = 1; // mantiene el objeto en un estado consistente
49         } // fin de else
50     } // fin de set
51 } // fin de la propiedad Mes
52
53 // propiedad que obtiene (get) y establecer (sets) el dia
54 public int Dia
55 {
56     get
57     {
58         return dia;
59     } // fin de get
60     private set // hace la escritura inaccesible fuera de la clase
61     {
62         int[] diasPorMes =
63             { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
64
65         // revisa si dia está dentro del rango para mes
66         if ( value > 0 && value <= diasPorMes[ Mes ] )
67             dia = value;
68         // revisa si es año bisiesto
69         else if ( Mes == 2 && value == 29 &&
70             ( Anio % 400 == 0 || ( Anio % 4 == 0 && Anio % 100 != 0 ) ) )
71             dia = value;
72         else
73         {
74             Console.WriteLine( "Dia inválido ({0}) se establece en 1.", value );
75             dia = 1; // mantiene el objeto en un estado consistente
76         } // fin de else
77     } // fin de set
78 } // fin de la propiedad Dia
79
80 // devuelve un objeto string de la forma mes/dia/anio
81 public override string ToString()
82 {
83     return string.Format( "{0}/{1}/{2}", Mes, Dia, Anio );
84 } // fin del método ToString
85 } // fin de la clase Fecha

```

Figura 9.9 | Declaración de la clase Fecha. (Parte 2 de 2).

```

1  // Fig. 9.10: Empleado.cs
2  // La clase Empleado con referencias a otros objetos.
3  public class Empleado
4  {
5      private string primerNombre;
6      private string apellido;
7      private Fecha fechaNacimiento;
8      private Fecha fechaContratacion;
9
10     // constructor para inicializar nombre, fecha de nacimiento y de contratación
11     public Empleado( string nombre, string apell,
12                     Fecha fechaDeNacimiento, Fecha fechaDeContratacion )
13     {
14         primerNombre = nombre;
15         apellido = apell;
16         fechaNacimiento = fechaDeNacimiento;
17         fechaContratacion = fechaDeContratacion;
18     } // fin del constructor de Empleado
19
20     // convierte Empleado al formato string
21     public override string ToString()
22     {
23         return string.Format( "{0}, {1} Contratado: {2} Cumpleaños: {3}",
24                               apellido, primerNombre, fechaContratacion, fechaNacimiento );
25     } // fin del método ToString
26 } // fin de la clase Empleado

```

Figura 9.10 | La clase Empleado con referencias a otros objetos.

a objetos Fecha, que demuestran que una clase puede tener como variables de instancia referencias a objetos de otras clases. El constructor de Empleado (líneas 11-18) recibe cuatro parámetros: nombre, apell, fechaDeNacimiento y fechaDeContratacion. Los objetos referenciados por los parámetros fechaDeNacimiento y fechaDeContratacion se asignan a las variables de instancia fechaNacimiento y fechaContratacion del objeto Empleado, respectivamente. Cuando se hace una llamada al método ToString de la clase Empleado, devuelve un objeto string que contiene las representaciones string de los dos objetos Fecha. Cada uno de estos objetos string se obtiene mediante una llamada implícita al método ToString de la clase Fecha.

La clase PruebaEmpleado (figura 9.11) crea dos objetos Fecha (líneas 9-10) para representar la fecha de nacimiento y de contratación de un Empleado, respectivamente. La línea 11 crea un Empleado e inicializa sus variables de instancia pasando al constructor dos objetos string (que representan el nombre y el apellido del Empleado) y dos objetos Fecha (que representan la fecha de nacimiento y de contratación). La línea 13 invoca en forma implícita al método ToString de Empleado para mostrar en pantalla los valores de sus variables de instancia y demostrar que el objeto se inicializó en forma apropiada.

9.9 Recolección de basura y destructores

Todo objeto que creamos utiliza varios recursos del sistema como la memoria. En muchos lenguajes de programación, estos recursos del sistema se reservan para uso del objeto hasta que se liberan explícitamente. Si se pierden todas las referencias al objeto que administra el recurso antes de que éste se libere en forma explícita, la aplicación ya no podrá acceder al recurso para liberarlo. Esto se conoce como *fuga de recursos*.

Necesitamos una manera disciplinada de regresar los recursos al sistema cuando ya no se necesitan, con lo cual se evita la fuga de recursos. El Entorno en tiempo de ejecución (CLR) administra la memoria en forma automática, mediante el uso de un *recolector de basura* para reclamar la memoria ocupada por los objetos que ya no se utilizan, para poder usarla con otros objetos. Cuando ya no hay más referencias a un objeto, éste se convierte en *candidato para la destrucción*. Todo objeto tiene un miembro especial, llamado *destructor*, que el recolector de basura invoca para realizar las *tareas de preparación para la terminación* sobre un objeto, justo antes de que el recolector de basura reclame la memoria del objeto. Un destructor se declara de igual manera que un constructor

```

1 // Fig. 9.11: PruebaEmpleado.cs
2 // Demostración de la composición.
3 using System;
4
5 public class PruebaEmpleado
6 {
7     public static void Main( string[] args )
8     {
9         Fecha nacimiento = new Fecha( 7, 24, 1949 );
10        Fecha contratacion = new Fecha( 3, 12, 1988 );
11        Empleado empleado = new Empleado( "Roberto", "Cantú", nacimiento, contratacion );
12
13        Console.WriteLine( empleado );
14    } // fin de Main
15 } // fin de la clase PruebaEmpleado

```

Constructor de objeto Fecha para fecha 7/24/1949
 Constructor de objeto Fecha para fecha 3/12/1988
 Cantú, Roberto Contratado: 3/12/1988 Cumpleaños: 7/24/1949

Figura 9.11 | Demostración de la composición.

sin parámetros, sólo que su nombre es el nombre de la clase, precedido por el símbolo (~), y no tiene modificador de acceso en su encabezado. Una vez que el recolector de basura llama al destructor del objeto, éste se convierte en *candidato para la recolección de basura*. El recolector de basura puede reclamar la memoria que usa dicho objeto. Las *fugas de memoria*, que son comunes en otros lenguajes como C y C++ (debido a que en esos lenguajes la memoria no se reclama de manera automática), son menos probables en C# (pero aun así se puede presentar, de ciertas formas sutiles). También es posible presentar otros tipos de fugas de recursos. Por ejemplo, una aplicación podría abrir un archivo en disco para modificar el contenido de ese archivo. Si la aplicación no cierra el archivo, ninguna otra aplicación puede modificarlo (o ni siquiera utilizarlo), sino hasta que termine la aplicación que lo abrió.

Un problema con el recolector de basura es que no se garantiza que realice sus tareas en un tiempo específico. Por lo tanto, podría llamar al destructor en cualquier momento después de que el objeto se convierta en candidato para la destrucción, y puede reclamar la memoria en cualquier momento después de que se ejecute el destructor. De hecho, tal vez no se realice ninguna de las dos acciones antes de que termine una aplicación. En consecuencia, no se sabe si se va a llamar al destructor, ni cuándo lo llamará. Por esta razón, muchos programadores evitan el uso de los destructores. En la sección 9.10 demostramos una situación en la que utilizamos un destructor. También demostraremos algunos de los métodos **static** de la clase **GC** (en el espacio de nombres **System**), la cual nos permite ejercer cierto control sobre el recolector de basura y cuándo se hacen las llamadas a los destructores.



Observación de ingeniería de software 9.7

Una clase que utiliza recursos del sistema, como archivos en el disco, debe proporcionar un método para liberar los recursos en un momento dado. Muchas clases de la FCL proporcionan métodos **Close** o **Dispose** para este propósito.

9.10 Miembros de clase static

Todo objeto tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo una copia de cierta variable debe compartirse entre todos los objetos de una clase. En dichos casos se utiliza una *variable static*, que representa *información a nivel de clase*: todos los objetos de la clase comparten la misma pieza de datos. La declaración de una variable **static** comienza con la palabra clave **static**.

Veamos un ejemplo. Suponga que tenemos un videojuego con Marcianos y otras criaturas espaciales. Cada Marciano tiende a ser valiente y está dispuesto a atacar a otras criaturas espaciales cuando está consciente de que hay por lo menos otros cuatro Marcianos presentes. Si hay menos de cinco Marcianos presentes, cada Marciano se vuelve cobarde. Por ende, cada Marciano necesita conocer la *cuentaDeMarcianos*. Podríamos dotar a la clase Marciano con una variable de instancia llamada *cuentaDeMarcianos*. Si hacemos esto, cada Marciano tendrá

una copia separada de la variable de instancia, y cada vez que creamos un nuevo Marciano tendremos que actualizar la variable de instancia `cuentaDeMarcianos` en cada objeto Marciano. El problema es que se desperdicia espacio debido a las copias redundantes, se pierde tiempo al actualizar las copias separadas y está propenso a errores. Mejor declaramos a `cuentaDeMarcianos` como `static`, para que esté a nivel de toda la clase. Cada Marciano puede acceder a `cuentaDeMarcianos` como si fuera una variable de instancia de la clase Marciano, pero sólo se mantiene una copia de `static cuentaDeMarcianos`. Esto ahorra espacio. Al hacer que el constructor de Marciano incremente la variable `static cuentaDeMarcianos`, también ahorramos tiempo; sólo hay una copia, por lo que no tenemos que incrementar copias separadas de `cuentaDeMarcianos` para cada objeto Marciano.



Observación de ingeniería de software 9.8

Use una variable `static` cuando todos los objetos de una clase deban utilizar la misma copia de la variable.

El alcance de una variable `static` es el cuerpo de su clase. Para acceder a un miembro `public static` de una clase, se califica el nombre del miembro con el nombre de la clase y el operador punto (`.`), como en `Math.PI`. Sólo se puede acceder a los miembros `private static` de una clase a través de los métodos y propiedades de esa clase. En realidad, los miembros de clase `static` existen incluso cuando no existen objetos de la clase; están disponibles tan pronto como se carga la clase en memoria, en tiempo de ejecución. Para acceder a un miembro `private static` desde el exterior de su clase, se puede proporcionar un método o propiedad `public static`.



Error común de programación 9.7

Si se accede o se invoca a un miembro `static` mediante una referencia a este miembro, a través de una instancia de la clase, como un miembro no `static`, se produce un error de compilación.



Observación de ingeniería de software 9.9

Las variables y los métodos `static` existen, y pueden utilizarse, incluso aunque no existan instancias de objetos de esa clase.

Nuestra siguiente aplicación declara dos clases: `Empleado` (figura 9.12) y `PruebaEmpleado` (figura 9.13). La clase `Empleado` declara la variable `private static` llamada `Cuenta` (figura 9.12, línea 10), y la propiedad `public static` llamada `Cuenta` (líneas 52-58). Omitimos el descriptor de acceso `set` de la propiedad `Cuenta` para que ésta sea de sólo lectura; no queremos que los clientes de la clase puedan modificar `cuenta`. La variable `static cuenta` se inicializa a 0 en la línea 10. Si no se inicializa una variable `static`, el compilador asigna un valor predeterminado a la variable; en este caso 0, el valor predeterminado para el tipo `int`. La variable `cuenta` mantiene la cuenta del número de objetos de la clase `Empleado` que se encuentra actualmente en memoria. Esta cuenta incluye los objetos que ya son inaccesibles para la aplicación, pero el recolector de basura no ha invocado todavía a sus destructores.

```

1  // Fig. 9.12: Empleado.cs
2  // La variable static se utiliza para mantener la cuenta del número de
3  // objetos Empleado en memoria.
4  using System;
5
6  public class Empleado
7  {
8      private string primerNombre;
9      private string apellidoPaterno;
10     private static int cuenta = 0; // número de objetos en memoria
11
12     // inicializa empleado, suma 1 a cuenta static y
13     // muestra en pantalla un objeto string que indica que se llamó al constructor
14     public Empleado( string nombre, string apellido )
15     {

```

Figura 9.12 | La variable `static` se utiliza para mantener la cuenta del número de objetos `Empleado` en memoria. (Parte I de 2).


```

16     primerNombre = nombre;
17     apellidoPaterno = apellido;
18     cuenta++; // incrementa cuenta static de empleados
19     Console.WriteLine( "Constructor de Empleado: {0} {1}; cuenta = {2}",
20         PrimerNombre, ApellidoPaterno, Cuenta );
21 } // fin de constructor de Empleado
22
23 // resta 1 a cuenta static cuando el recolector de basura
24 // llama al destructor para limpiar el objeto;
25 // confirma que se llamó al destructor
26 ~Empleado()
27 {
28     cuenta--; // decrementa cuenta static de empleados
29     Console.WriteLine( "Destructor de Empleado: {0} {1}; cuenta = {2}",
30         PrimerNombre, ApellidoPaterno, Cuenta );
31 } // fin del destructor
32
33 // propiedad de sólo lectura que obtiene (get) el primer nombre
34 public string PrimerNombre
35 {
36     get
37     {
38         return primerNombre;
39     } // fin de get
40 } // fin de la propiedad PrimerNombre
41
42 // propiedad de sólo lectura que obtiene (get) el apellido
43 public string ApellidoPaterno
44 {
45     get
46     {
47         return apellidoPaterno;
48     } // fin de get
49 } // fin de la propiedad ApellidoPaterno
50
51 // propiedad de sólo lectura que obtiene (get) la cuenta de empleados
52 public static int Cuenta
53 {
54     get
55     {
56         return cuenta;
57     } // fin de get
58 } // fin de la propiedad Cuenta
59 } // fin de la clase Empleado

```

Figura 9.12 | La variable static se utiliza para mantener la cuenta del número de objetos Empleado en memoria. (Parte 2 de 2).

Cuando existen objetos Empleado, la cuenta de miembros puede usarse en cualquier método de un objeto Empleado; este ejemplo incrementa cuenta en el constructor (línea 18) y la decrementa en el destructor (línea 28). Cuando no existen objetos de la clase Empleado se puede hacer referencia de todas formas a la cuenta de miembros, pero sólo a través de una llamada a la propiedad public static Cuenta (líneas 52-58), como en Empleado.Cuenta, lo cual evalúa el número de objetos Empleado que se encuentran actualmente en memoria.

Observe que la clase Empleado tiene un destructor (líneas 26-31). Este destructor se incluye para decrementar la variable static cuenta y después muestra cuándo se ejecuta el recolector de basura en esta aplicación. A diferencia de los constructores y los métodos, ningún código escrito por el programador puede invocar al destructor en forma explícita. Sólo se puede invocar por el recolector de basura, por lo que no necesita un modificador de acceso; de hecho, si se incluye uno se produce un error de sintaxis.

El método Main de PruebaEmpleado (figura 9.13) crea instancias de dos objetos Empleado (líneas 14-15). Cuando se invoca al constructor de cada objeto Empleado, las líneas 16-17 de la figura 9.12 asignan el primer nombre y el apellido paterno de Empleado a las variables de instancia primerNombre y apellidoPaterno. Observe que estas dos instrucciones no crean copias de los argumentos string originales. En realidad, los objetos string en C# son inmutables: no pueden modificarse una vez creados. Por lo tanto, es seguro tener muchas referencias a un objeto string. Por lo general éste no es el caso para objetos de la mayoría de las otras clases en C#. Si los objetos string son inmutables, tal vez usted se pregunte por qué podemos utilizar los operadores + y += para concatenar objetos string. En realidad, las operaciones de concatenación de objetos string producen un nuevo objeto string que contiene los valores concatenados. Los objetos string originales no se modifican.

Cuando Main termina de utilizar los dos objetos Empleado, las referencias e1 y e2 se establecen a null en las líneas 29-30, por lo que ya no hacen referencia a los objetos que se instanciaron en las líneas 14-15. Los objetos se convierten en “candidatos para la recolección de basura”, ya que no hay más referencias a ellos en la aplicación.

```

1  // Fig. 9.13: PruebaEmpleado.cs
2  // Demostración de un miembro static.
3  using System;
4
5  public class PruebaEmpleado
6  {
7      public static void Main( string[] args )
8      {
9          // muestra que cuenta es 0 antes de crear Empleados
10         Console.WriteLine( "Empleados antes de instanciar: {0}",
11             Empleado.Cuenta );
12
13         // crea dos Empleados; cuenta debe convertirse en 2
14         Empleado e1 = new Empleado( "Susan", "Baker" );
15         Empleado e2 = new Empleado( "Bob", "Blue" );
16
17         // muestra que cuenta es 2 después de crear dos Empleados
18         Console.WriteLine( "\nEmpleados después de instanciar: {0}",
19             Empleado.Cuenta );
20
21         // obtiene los nombres de los Empleados
22         Console.WriteLine( "\nEmpleado 1: {0} {1}\nEmpleado 2: {2} {3}\n",
23             e1.PrimerNombre, e1.ApellidoPaterno,
24             e2.PrimerNombre, e2.ApellidoPaterno );
25
26         // en este ejemplo, sólo hay una referencia a cada Empleado,
27         // por lo que las siguientes instrucciones hacen que CLR marque cada
28         // objeto Empleado como candidato para la destrucción
29         e1 = null; // ya no se necesita el objeto e1
30         e2 = null; // ya no se necesita el objeto e2
31
32         GC.Collect(); // pide que se realice ahora la recolección de basura
33         // espera hasta que los destructores
34         // terminen de escribir a la consola
35         GC.WaitForPendingFinalizers();
36
37         // muestra la cuenta de Empleados después de llamar al recolector de basura y
38         // espera a que terminen todos los destructores
39         Console.WriteLine( "\nEmpleados después de la destrucción: {0}",
40             Empleado.Cuenta );
41     } // fin de Main
42 } // fin de la clase PruebaEmpleado

```

Figura 9.13 | Demostración de un miembro static. (Parte I de 2).

```

Empleados antes de instanciar: 0
Constructor de Empleado: Susan Baker; cuenta = 1
Constructor de Empleado: Bob Blue; cuenta = 2

Empleados después de instanciar: 2

Empleado 1: Susan Baker
Empleado 2: Bob Blue

Destructor de Empleado: Bob Blue; cuenta = 1
Destructor de Empleado: Susan Baker; cuenta = 0

Empleados después de la destrucción: 0

```

Figura 9.13 | Demostración de un miembro static. (Parte 2 de 2).

En un momento dado, el recolector de basura podría reclamar la memoria de estos objetos (o el sistema operativo reclamará la memoria cuando termine la aplicación). C# no garantiza cuándo, o incluso si el recolector de basura se ejecutará o no, por lo que en la línea 32, esta aplicación llama en forma explícita al recolector de basura mediante el método static `Collect` de la clase `GC`, para indicar que el recolector de basura debe hacer su “mejor esfuerzo” para reclamar los objetos inaccesibles. Sin embargo, esto sólo es su mejor esfuerzo; es posible que no se recolecte ningún objeto, o que sólo se recolecte un subconjunto de los objetos candidatos. Cuando el método `Collect` regresa, esto *no* indica que el recolector de basura haya terminado de buscar memoria para reclamarla. De hecho, el recolector de basura podría seguir ejecutándose. Por esta razón, llamamos al método static **`WaitForPendingFinalizers`** de la clase `GC`. Si el recolector de basura marca algunos objetos como candidatos para la destrucción, la llamada al método `WaitForPendingFinalizers` en la línea 35 detiene la ejecución del método `Main` hasta que se hayan ejecutado por completo los destructores de estos objetos.

En los resultados de ejemplo de la figura 9.13, el recolector de basura reclamó los objetos a los que `e1` y `e2` hacían referencia antes de que las líneas 39-40 mostraran en pantalla la cuenta actual de objetos `Empleado`. La última línea de salida indica que el número de objetos `Empleado` en memoria es 0 después de la llamada a `GC.Collect()`. Las líneas antepenúltima y penúltima de la salida muestran que el objeto `Empleado` para Bob Blue se destruyó antes del objeto `Empleado` para Susan Baker. Los resultados en su sistema podrían ser distintos, ya que no se garantiza que se ejecute el recolector de basura cuando se hace la llamada a `GC.Collect()`, ni se garantiza que los objetos se recolecten en un orden específico. De hecho, si omite la llamada a `WaitForPendingFinalizers`, es probable que se ejecuten las líneas 39-40 antes de que el recolector de basura tenga la oportunidad de llamar a los destructores.

[Nota: un método que se declara como static no puede acceder en forma directa a los miembros no static de la clase, ya que un método static puede llamarse incluso aunque no existan objetos de esa clase. Por la misma razón, no puede utilizarse la referencia `this` en un método static; la referencia `this` debe referirse a un objeto específico de la clase y, cuando se hace la llamada a un método static, tal vez no haya objetos de su clase en memoria. La referencia `this` se requiere para permitir que un método de una clase acceda a los miembros no static de la misma clase.]



Error común de programación 9.8

Si un método static llama al método de una instancia (no static) en la misma clase utilizando sólo el nombre del método, se produce un error de compilación. De manera similar, si un método static trata de acceder a una variable de instancia en la misma clase utilizando sólo el nombre de la variable, también se produce un error de compilación.



Error común de programación 9.9

Referirse a la referencia this en un método static es un error de sintaxis.

9.11 Variables de instancia `readonly`

El *principio del menor privilegio* es fundamental para la buena ingeniería de software. En el contexto de una aplicación, el principio establece que sólo se deben otorgar al código los privilegios y el nivel de acceso necesarios para realizar su tarea designada, pero no más. Veamos cómo se aplica este principio a las variables de instancia.

Algunas variables de instancia necesitan ser modificables y otras no. En la sección 8.4 presentamos la palabra clave `const` para declarar constantes, que deben inicializarse con un valor constante al momento de declararse. No obstante, suponga que queremos inicializar una constante que pertenece a un objeto en el constructor de ese objeto. C# cuenta con la palabra clave **`readonly`** para especificar que una variable de instancia de un objeto no es modificable, y que cualquier intento de modificarla una vez que se crea el objeto es un error. Por ejemplo,

```
private readonly int INCREMENTO;
```

declara la variable de instancia `readonly` llamada `INCREMENTO`, de tipo `int`. Al igual que las constantes, por convención, las variables `readonly` se declaran con su nombre en letras mayúsculas. Aunque las variables de instancia `readonly` pueden inicializarse al momento de declararse, esto no es requerido. Las variables de instancia `readonly` pueden inicializarse mediante cada uno de los constructores de la clase. El constructor puede asignar valores a una variable de instancia `readonly` varias veces; la variable deja de ser modificable una vez que el constructor completa su ejecución.



Observación de ingeniería de software 9.10

Declarar una variable de instancia como `readonly` ayuda a reforzar el principio del menor privilegio. Si una variable de instancia no debe modificarse una vez que se cree el objeto, declárela como `readonly` para evitar que se modifique.

Los miembros que se declaran como `const` deben recibir valores en tiempo de compilación. Por lo tanto, los miembros `const` sólo pueden inicializarse con otros valores constantes, como enteros, literales `string`, caracteres y otros miembros `const`. Los miembros constantes con valores que no puedan determinarse en tiempo de compilación deberán declararse con la palabra clave `readonly`, para que puedan inicializarse en tiempo de ejecución. Las variables que sean `readonly` pueden inicializarse con expresiones más complejas como un inicializador de arreglos o la llamada a un método que devuelva un valor o una referencia a un objeto.

Nuestro siguiente ejemplo contiene dos clases: `Incremento` (figura 9.14) y `PruebaIncremento` (figura 9.15). La primera contiene una variable de instancia `readonly` de tipo `int` llamada `INCREMENTO` (figura 9.14, línea 6). Observe que la variable `readonly` no se inicializa en su declaración, por lo que debe inicializarse mediante el constructor de la clase (líneas 10-13). Si la clase proporciona varios constructores, cada uno de ellos debe inicializar la variable `readonly`. Si un constructor no inicializa la variable `readonly`, ésta recibirá el mismo valor predeterminado que cualquier otra variable de instancia (0 para los tipos numéricos simples, `false` para los tipos `bool` y `null` para los tipos de referencias), y el compilador generará una advertencia. En la figura 9.14, el constructor recibe el parámetro `int` llamado `valorIncremento` y asigna su valor a `INCREMENTO` (línea 12). Si el constructor de la clase `Incremento` no inicializa `INCREMENTO` (si se omite la línea 12), el compilador genera la siguiente advertencia:

El campo 'Incremento.INCREMENTO' nunca se asigna y siempre tendrá el valor predeterminado 0

La clase de aplicación `PruebaIncremento` crea un objeto de la clase `Incremento` (figura 9.15, línea 9) y proporciona el valor 5 como argumento para el constructor, que se asigna a la variable `readonly` `INCREMENTO`. Las líneas 11 y 16 invocan en forma implícita el método `ToString` de la clase `Incremento`, el cual devuelve un objeto `string` con formato, que describe el valor de la variable de instancia `private total`.



Error común de programación 9.10

Tratar de modificar una variable de instancia `readonly` en cualquier parte que no sea su declaración o en los constructores del objeto, es un error de compilación.

```

1 // Fig. 9.14: Incremento.cs
2 // variable de instancia readonly en una clase.
3 public class Incremento
4 {
5     // variable de instancia readonly (sin inicializar)
6     private readonly int INCREMENTO;
7     private int total = 0; // total de todos los incrementos
8
9     // el constructor inicializa la variable de instancia readonly INCREMENTO
10    public Incremento( int valorIncremento )
11    {
12        INCREMENTO = valorIncremento; // inicializa variable readonly (una vez)
13    } // fin del constructor de Incremento
14
15    // suma INCREMENTO a total
16    public void SumarIncrementoATotal()
17    {
18        total += INCREMENTO;
19    } // fin del método SumarIncrementoATotal
20
21    // devuelve la representación string de los datos de un objeto Incremento
22    public override string ToString()
23    {
24        return string.Format( "total = {0}", total );
25    } // fin del método ToString
26 } // fin de la clase Incremento

```

Figura 9.14 | Variable de instancia readonly en una clase.

```

1 // Fig. 9.15: PruebaIncremento.cs
2 // variable de instancia readonly, inicializada mediante un argumento para su
  constructor.
3 using System;
4
5 public class PruebaIncremento
6 {
7     public static void Main( string[] args )
8     {
9         Incremento incrementador = new Incremento( 5 );
10
11         Console.WriteLine( "Antes de incrementar: {0}\n", incrementador );
12
13         for ( int i = 1; i <= 3; i++ )
14         {
15             incrementador.SumarIncrementoATotal();
16             Console.WriteLine( "Después de incrementar {0}: {1}", i, incrementador );
17         } // fin de for
18     } // fin de Main
19 } // fin de la clase PruebaIncremento

```

Antes de incrementar: total = 0

Después de incrementar 1: total = 5
 Después de incrementar 2: total = 10
 Después de incrementar 3: total = 15

Figura 9.15 | Variable de instancia readonly, inicializada mediante un argumento para su constructor.



Tip de prevención de errores 9.2

Los intentos por modificar una variable de instancia `readonly` se atrapan en tiempo de compilación, en vez de que produzcan errores en tiempo de ejecución. Siempre es preferible obtener errores en tiempo de compilación, si es posible, en vez de permitir que se escabullan hasta el tiempo de ejecución (en donde los estudios han encontrado que su reparación es por lo general muchas veces más costosa).



Observación de ingeniería de software 9.11

Si una variable de instancia `readonly` se inicializa con una constante sólo en su declaración, no es necesario tener una copia separada de la variable de instancia para todos los objetos de la clase. En vez de ello, la variable debe declararse `const`. Las constantes que se declaran con `const` son `static` de manera implícita, por lo que sólo habrá una copia para toda la clase.

9.12 Reutilización de software

Los programadores se concentran en elaborar nuevas clases y reutilizar las existentes. Existen muchas bibliotecas de clases, y otras se están desarrollando en diversas partes del mundo. Así, el software se construye a partir de componentes existentes, bien definidos, cuidadosamente probados, bien documentados, portables, optimizados para el mejor rendimiento, y ampliamente disponibles. Este tipo de reutilización de software agiliza el desarrollo de software poderoso y de alta calidad. El *desarrollo rápido de aplicaciones (RAD)* es de gran interés hoy en día.

Microsoft proporciona a los programadores de C# miles de clases en la Biblioteca de clases del .NET Framework, para ayudarlos a implementar aplicaciones en C#. El .NET Framework permite a los desarrolladores en C# trabajar para lograr una verdadera reutilización y un rápido desarrollo de aplicaciones. Los programadores en C# se pueden enfocar en la tarea en cuestión a la hora de desarrollar sus aplicaciones, y dejan los detalles de bajo nivel a las clases de la FCL. Por ejemplo, para escribir una aplicación que dibuje gráficos, un programador de la FCL no requiere conocer los gráficos en cada una de las plataformas computacionales en las que se ejecutará la aplicación. En vez de ello, el programador se puede concentrar en aprender las capacidades de gráficos de .NET (que son bastante sustanciales y aumentan en forma constante) y escribir una aplicación en C# que dibuje los gráficos, usando clases de la FCL como las que se encuentran en el espacio de nombres `System.Drawing`. Cuando la aplicación se ejecuta en cierta computadora, es responsabilidad de la CLR traducir los comandos de MSIL compilados del código C# en comandos que la computadora local pueda entender.

Las clases de la FCL permiten a los programadores en C# llevar las nuevas aplicaciones al mercado con más rapidez, mediante el uso de componentes preexistentes ya probados. Esto no sólo reduce el tiempo de desarrollo, sino que también mejora la habilidad del programador para depurar y dar mantenimiento a las aplicaciones. Para aprovechar las diversas herramientas de C#, es esencial que los programadores se familiaricen con la variedad de clases en el .NET Framework. Hay muchos recursos basados en Web en msdn2.microsoft.com (msdn.microsoft.com/library/spa/ en español) para ayudarle con esta tarea. El recurso principal para aprender acerca de la FCL es la Referencia de .NET Framework en la biblioteca MSDN, que puede encontrar en

msdn.microsoft.com/library/spa/default.asp?url=/library/SPA/netstart/html/sdk_netstart.asp

Además, msdn2.microsoft.com (msdn.microsoft.com/library/spa/ en español) proporciona muchos recursos más, incluyendo tutoriales, artículos y sitios específicos para temas individuales de C#.



Buena práctica de programación 9.1

Evite reinventar la rueda. Estudie las herramientas de la FCL. Si la FCL contiene una clase que cumpla con los requerimientos de su aplicación, utilícela en lugar de crear su propia clase.

Para comprender todo el potencial de la reutilización de software, necesitamos mejorar los esquemas de clasificación, los esquemas de licenciamiento, los mecanismos de protección que evitan la corrupción de copias maestras de las clases, los esquemas de descripción que utilizan los desarrolladores de sistemas para determinar si las clases existentes cumplen con sus necesidades, los mecanismos de exploración que determinan qué clases hay disponibles y qué tan cerca están de cumplir con los requerimientos de los desarrolladores de software, etc. Muchos problemas interesantes de investigación y desarrollo se han resuelto, y hay muchos más por resolver. Es probable que estos problemas se resuelvan, debido a que el valor potencial del aumento en la reutilización de software es enorme.

9.13 Abstracción de datos y encapsulamiento

Por lo general, las clases ocultan a sus clientes los detalles de su implementación. A esto se le conoce como *ocultamiento de información*. Como ejemplo, consideremos la estructura de datos tipo pila que presentamos en la sección 7.6. Recuerde que una pila es una estructura de datos tipo “último en entrar, primero en salir” (UEPS): el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.

Las pilas pueden implementarse mediante arreglos y otras estructuras de datos, como listas enlazadas (en el capítulo 24, Estructuras de datos, y en el capítulo 26, Colecciones, hablaremos sobre las pilas y las listas enlazadas). Un cliente de una clase tipo pila no necesita preocuparse por la implementación de la pila. El cliente sólo sabe que cuando se colocan elementos de datos en la pila, éstos se recuperan en orden del último en entrar, primero en salir. El cliente sólo tiene que preocuparse por la funcionalidad que ofrece una pila, no por cómo se implementa esa funcionalidad. A este concepto se le conoce como *abstracción de datos*. Aunque los programadores podrían conocer los detalles de la implementación de una clase, no necesitan escribir código que dependa de estos detalles. Esto permite que una clase específica (como una que implemente a una pila y sus operaciones *push* y *pop*) se sustituya por otra versión, sin afectar al resto del sistema. Mientras que los servicios `public` de la clase no cambian (es decir, que todos los métodos originales sigan teniendo el mismo nombre, tipo de valor de retorno y lista de parámetros en la nueva declaración de la clase), el resto del sistema no se verá afectado.

La mayoría de los lenguajes de programación enfatiza las acciones. En estos lenguajes, los datos existen para dar soporte a las acciones que deben realizar las aplicaciones. Los datos son “menos interesantes” que las acciones; son “crudos”. Sólo existen unos cuantos tipos simples y es difícil para los programadores crear sus propios tipos. C# y el estilo de programación orientado a objetos elevan la importancia de los datos. Las principales actividades de la programación orientada a objetos en C# son la creación de tipos (por ejemplo, clases) y la expresión de las interacciones entre los objetos de esos tipos. Para crear lenguajes que enfatizan los datos, la comunidad de los lenguajes de programación necesitaba formalizar ciertas nociones sobre los datos. La formalización que consideramos en este libro es la de los *tipos de datos abstractos (ADTs)*, que mejora el proceso de desarrollo de aplicaciones.

Considere el tipo simple `int`, que la mayoría de las personas asociaría con un entero en matemáticas. En realidad, un `int` es una representación abstracta de un entero. A diferencia de los enteros matemáticos, los `int` de las computadoras tienen un tamaño fijo. Por ejemplo, el tipo simple `int` en C# está limitado al rango de $-2,147,486,648$ a $+2,147,483,647$. Si el resultado de un cálculo queda fuera de este rango se produce un error, y la computadora responde de cierta manera. Por ejemplo, podría producir “silenciosamente” un resultado incorrecto, como un valor demasiado grande como para alojarlo en una variable `int`; a esto se le conoce como *desbordamiento aritmético*. También se podría lanzar una excepción, conocida como `OverflowException` (en la sección 12.8 veremos las dos formas de tratar con el desbordamiento aritmético). Los enteros matemáticos no tienen este problema. Por lo tanto, el `int` de computadora es sólo una aproximación del entero real. Lo mismo se aplica para el tipo `double` y los demás tipos simples.

Hemos dado por sentada la noción de `int` hasta este punto, pero ahora la consideraremos desde una nueva perspectiva. Los tipos como `int`, `double` y `char` son ejemplos de los tipos de datos abstractos. Son representaciones de conceptos del mundo real, hasta un cierto nivel satisfactorio de precisión dentro de un sistema computacional.

En realidad un ADT captura dos nociones: una *representación de datos* y las *operaciones* que pueden realizarse sobre esos datos. Por ejemplo, en C# un `int` contiene un valor entero (datos) y proporciona las operaciones de suma, resta, multiplicación, división y residuo; la división entre cero no está definida. Los programadores en C# utilizan clases para implementar los tipos de datos abstractos.



Observación de ingeniería de software 9.12

Los programadores crean tipos a través del mecanismo de las clases. Pueden diseñarse nuevos tipos para que sean tan convenientes de usar como los tipos simples. Esto identifica a C# como un lenguaje extensible. Aunque este lenguaje es fácil de extender a través de nuevos tipos, el programador no puede alterar el lenguaje base en sí.

Otro tipo de datos abstracto que veremos es una *cola*, similar a una “fila de espera”. Los sistemas computacionales utilizan muchas colas de manera interna. Una cola ofrece un comportamiento bien definido a sus clientes: los clientes colocan elementos en una cola, uno a la vez, a través de una operación *enqueue* (agregar a la cola), después los recuperan uno a la vez a través de una operación *dequeue* (retirar de la cola). Una cola devuelve los elementos en el orden *primero en entrar, primero en salir (PEPS)*, lo cual significa que el primer elemento que se inserta en una

cola es el primero que se extrae. De manera conceptual, una cola puede volverse infinitamente larga, pero las colas reales son finitas.

La cola oculta una representación de datos interna que lleva el registro de los elementos que esperan en la fila en un momento dado, y ofrece operaciones a sus clientes (*enqueue* y *dequeue*). Los clientes no se preocupan por la implementación de la cola; sólo se basan en que operará como es de esperarse. Cuando un cliente agrega un elemento a la cola, ésta debe aceptarlo y colocarlo en algún tipo de estructura de datos PEPS. De manera similar, cuando el cliente desea el siguiente elemento de la parte frontal de la cola, ésta debe extraerlo de su representación de datos interna y entregarlo en orden PEPS (es decir, el elemento que haya estado más tiempo en la cola es el que debe devolver la siguiente operación *dequeue*).

El ADT tipo cola garantiza la integridad de su estructura de datos interna. Los clientes no pueden manipular esta estructura de datos en forma directa; sólo el ADT tipo cola tiene acceso a sus datos internos. Los clientes sólo pueden realizar operaciones que se permitan sobre la representación de los datos; el ADT rechaza las operaciones que no proporcione su interfaz pública. En el capítulo 24, Estructuras de datos, hablaremos sobre las pilas y las colas con mayor detalle.

9.14 Caso de estudio de la clase **Tiempo**: creación de bibliotecas de clases

En casi todos los ejemplos de este libro hemos visto que las clases de bibliotecas preexistentes, como la Biblioteca de clases del .NET Framework, pueden importarse en una aplicación en C#. Cada clase en la FCL pertenece a un espacio de nombres que contiene un grupo de clases relacionadas. A medida que las aplicaciones se vuelven más complejas, los espacios de nombres nos permiten administrar la complejidad de los componentes de una aplicación. Las bibliotecas de clases y los espacios de nombres también facilitan la reutilización de software, al permitir que las aplicaciones agreguen clases de otros espacios de nombres (como lo hemos hecho en la mayoría de los ejemplos). En esta sección veremos cómo crear nuestras propias bibliotecas de clases.

Pasos para declarar y utilizar una clase reutilizable

Antes de poder usar una clase en varias aplicaciones, ésta debe colocarse en una biblioteca de clases para que sea reutilizable. La figura 9.16 muestra cómo especificar el espacio de nombres en el que debe colocarse una clase en la biblioteca. La figura 9.19 muestra cómo utilizar nuestra biblioteca de clases en una aplicación. Los pasos para crear una biblioteca reutilizable son:

1. Declare una clase `public`. Si la clase no es `public`, sólo la podrán usar otras clases en el mismo ensamblado.
2. Seleccione un nombre para el espacio de nombres y agregue una *declaración namespace* al archivo de código fuente para la declaración de la clase reutilizable.
3. Compile la clase en una biblioteca de clases.
4. Agregue una referencia a la biblioteca de clases en una aplicación.
5. Especifique una directiva `using` para el espacio de nombres de la clase reutilizable, y utilice la clase.

Paso 1: crear una clase `public`

Para el *paso 1* utilizamos la clase `public Tiempo1` que declaramos en la figura 9.1. No se han hecho modificaciones a la implementación de la clase, por lo que no hablaremos otra vez aquí sobre sus detalles de implementación.

Paso 2: agregar la declaración namespace

Para el *paso 2* agregamos una declaración `namespace` a la figura 9.1. La nueva versión se muestra en la figura 9.16. La línea 3 declara un espacio de nombres llamado `Capitulo09`. Al colocar la clase `Tiempo1` dentro de la declaración `namespace` se indica que esa clase forma parte del espacio de nombres especificado. El nombre del espacio de nombres es parte del nombre completamente calificado de la clase, por lo que el nombre de la clase `Tiempo1` es en realidad `Capitulo09.Tiempo1`. Puede utilizar este nombre completamente calificado en sus aplicaciones, o puede escribir una directiva `using` (como veremos en breve) y utilizar su *nombre simple* (el nombre descalificado de la clase: `Tiempo1`) en la aplicación. Si otro espacio de nombres también contiene una clase `Tiempo1`, pueden

```

1 // Fig. 9.16: Tiempo1.cs
2 // Declaración de la clase Tiempo1 en un espacio de nombres.
3 namespace Capitulo09
4 {
5     public class Tiempo1
6     {
7         private int hora; // 0 - 23
8         private int minuto; // 0 - 59
9         private int segundo; // 0 - 59
10
11         // establece un nuevo valor de tiempo, usando la hora universal; asegura que
12         // los datos permanezcan consistentes al establecer los valores inválidos a cero
13         public void EstablecerTiempo( int h, int m, int s )
14         {
15             hora = ( ( h >= 0 && h < 24 ) ? h : 0 ); // valida hora
16             minuto = ( ( m >= 0 && m < 60 ) ? m : 0 ); // valida minuto
17             segundo = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida segundo
18         } // fin del método EstablecerTiempo
19
20         // convierte a string en formato de hora universal (HH:MM:SS)
21         public string AStringUniversal()
22         {
23             return string.Format( "{0:D2}:{1:D2}:{2:D2}",
24                 hora, minuto, segundo );
25         } // fin del método AStringUniversal
26
27         // convierte a string en formato de hora estándar (H:MM:SS AM o PM)
28         public override string ToString()
29         {
30             return string.Format( "{0}:{1:D2}:{2:D2} {3}",
31                 ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ),
32                 minuto, segundo, ( hora < 12 ? "AM" : "PM" ) );
33         } // fin del método ToString
34     } // fin de la clase Tiempo1
35 } // fin del espacio de nombres Capitulo09

```

Figura 9.16 | Declaración de la clase Tiempo1 en un espacio de nombres.

usarse los nombres completamente calificados de las clases para diferenciarlas en la aplicación y evitar un *conflicto de nombres* (lo que también se conoce como *colisión de nombres*).

Sólo las declaraciones namespace, las directivas using, los comentarios y los atributos de C# (que utilizaremos en el capítulo 18) pueden aparecer fuera de las llaves de una declaración de tipo (por ejemplo, clases y enumeraciones). Sólo las clases que se declaren como public podrán reutilizarse por los clientes de la biblioteca de clases. Por lo general, las clases no public se colocan en una biblioteca para dar soporte a las clases public reutilizables de esa biblioteca.

Paso 3: compilar la biblioteca de clases

El *paso 3* es compilar la clase en una biblioteca de clases. Para crear una biblioteca de clases en Visual C# Express, debemos crear un nuevo proyecto haciendo clic en el menú **Archivo**, después seleccionamos **Nuevo proyecto...** y elegimos **Biblioteca de clases** de la lista de plantillas, como se muestra en la figura 9.17. Después agregamos el código de la figura 9.16 en el nuevo proyecto (puede copiar el código de los ejemplos del libro o escribirlo usted mismo). En los proyectos que ha creado hasta ahora, el compilador de C# creaba un archivo .exe ejecutable, que contenía la aplicación. Al compilar un proyecto **Biblioteca de clases**, el compilador crea un *archivo .dll*, al cual se le conoce como *biblioteca de vínculos dinámicos*: un tipo de ensamblado al que podemos hacer referencias desde otras aplicaciones.

Paso 4: agregar una referencia a la biblioteca de clases

Una vez que la clase se compila y se guarda en el archivo de biblioteca de clases, se puede hacer referencia a la biblioteca desde cualquier aplicación; para ello hay que indicar al IDE de Visual C# Express en dónde puede

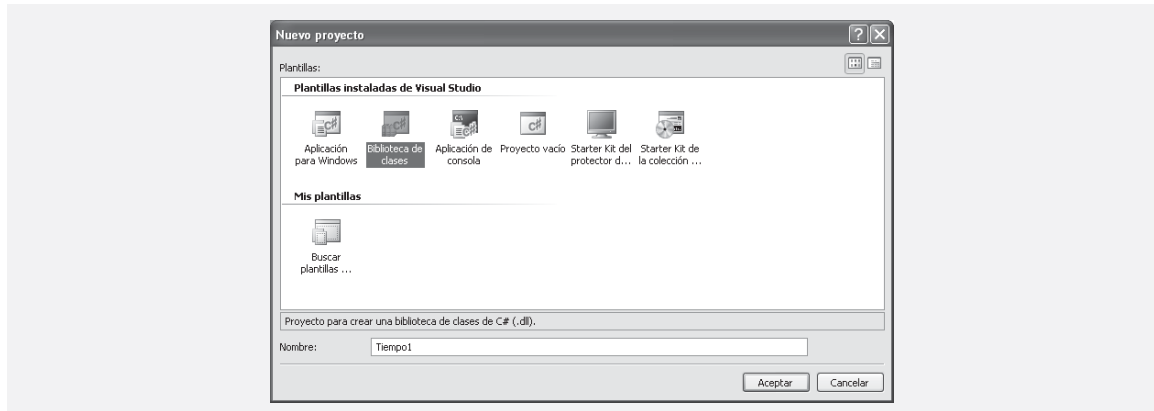


Figura 9.17 | Creación de un proyecto **Biblioteca de clases**.

encontrar el archivo de biblioteca (*paso 4*). Cree un nuevo proyecto (vacío) y haga clic derecho sobre el nombre del proyecto en la ventana **Explorador de soluciones**. Seleccione **Agregar referencia...** del menú contextual que aparezca. El cuadro de diálogo que aparece a continuación contiene una lista de bibliotecas de clases del .NET Framework. Algunas bibliotecas de clases, como la que contiene el espacio de nombres `System`, son tan comunes que se agregan a su aplicación de manera implícita. Las que se encuentran en esta lista no se agregan así.

En el cuadro de diálogo **Agregar referencia...**, haga clic en la ficha **Examinar**. En la sección 3.3 vimos que cuando se crea una aplicación, Visual C# 2005 coloca el archivo `.exe` en la carpeta `bin\Release` del directorio de su aplicación. Cuando usted crea una biblioteca de clases, Visual C# coloca el archivo `.dll` en el mismo lugar. En la ficha **Examinar** puede navegar hasta el directorio que contiene el archivo de biblioteca de clases que usted creó en el *paso 3*, como se muestra en la figura 9.18. Seleccione el archivo `.dll` y haga clic en **Aceptar**.

Paso 5: usar la clase desde una aplicación

Agregue un nuevo archivo de código a su aplicación e introduzca el código para la clase `PruebaEspacioNombresTiempo1` (figura 9.19). Ahora que ha agregado una referencia a su biblioteca de clases en esta aplicación, `PruebaEspacioNombresTiempo1` puede utilizar su clase `Tiempo1` (*paso 5*) sin tener que agregar el archivo de código fuente `Tiempo1.cs` al proyecto.

En la figura 9.19, la directiva `using` en la línea 3 especifica que deseamos utilizar la(s) clase(s) de espacio de nombres `Capitulo09` en este archivo. La clase `PruebaEspacioNombres` es el espacio de nombres global de esta aplicación, ya que el archivo de la clase no contiene una declaración `namespace`. Como las dos clases se encuentran en espacios de nombres distintos, la directiva `using` en la línea 3 permite a la clase `PruebaEspacioNombres-Tiempo1` utilizar la clase `Tiempo1`, como si estuviera en el mismo espacio de nombres.

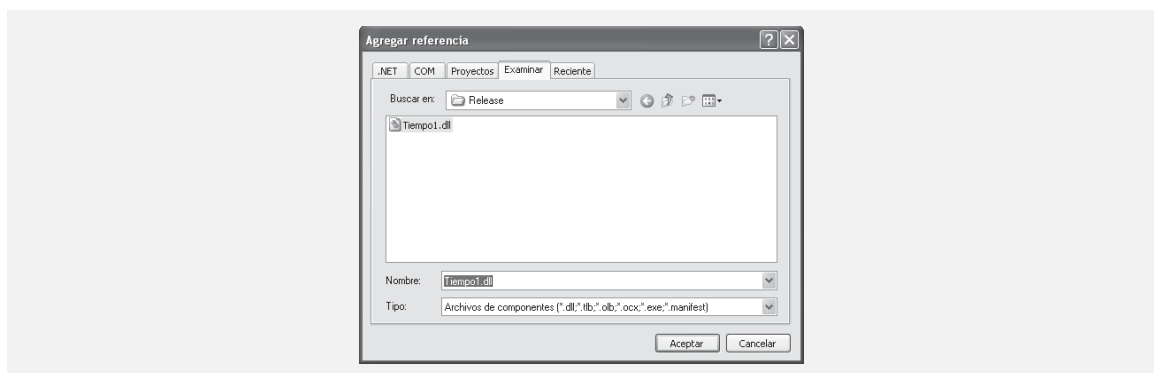


Figura 9.18 | Agregar una referencia.

En la sección 4.4 vimos que podíamos omitir la dirección `using` en la línea 4, si nos referíamos siempre a la clase `Console` por su nombre de clase completamente calificado, `System.Console`. De manera similar, podríamos omitir la directiva `using` en la línea 3 para el espacio de nombres `Capitulo09` si cambiáramos la declaración de `Tiempo1` en la línea 11 de la figura 9.19 para que utilizara el nombre completamente calificado de la clase `Tiempo1`, como se muestra a continuación:

```
Capitulo09.Tiempo1 tiempo = new Capitulo09.Tiempo1();
```

9.15 Acceso interno

Las clases pueden declararse con sólo dos modificadores de acceso: `public` e `internal`. Si no hay un modificador de acceso en la declaración de la clase, el valor predeterminado es el *acceso interno*. Esto permite que la clase

```

1  // Fig. 9.19: PruebaEspacioNombresTiempo1.cs
2  // Uso del objeto Tiempo1 en una aplicación.
3  using Capitulo09;
4  using System;
5
6  public class PruebaEspacioNombresTiempo1
7  {
8      public static void Main( string[] args )
9      {
10         // crea e inicializa un objeto Tiempo1
11         Tiempo1 tiempo = new Tiempo1(); // llama al constructor de Tiempo1
12
13         // imprime en pantalla representaciones string del tiempo
14         Console.Write( "La hora universal inicial es: " );
15         Console.WriteLine( tiempo.AStringUniversal() );
16         Console.Write( "La hora estándar inicial es: " );
17         Console.WriteLine( tiempo.ToString() );
18         Console.WriteLine(); // imprime una línea en blanco
19
20         // cambia tiempo e imprime en pantalla tiempo actualizado
21         tiempo.EstablecerTiempo( 13, 27, 6 );
22         Console.Write( "La hora universal después de EstablecerTiempo es: " );
23         Console.WriteLine( tiempo.AStringUniversal() );
24         Console.Write( "La hora estándar después de EstablecerTiempo es: " );
25         Console.WriteLine( tiempo.ToString() );
26         Console.WriteLine(); // imprime una línea en blanco
27
28         // establece tiempo con valores inválidos; imprime el tiempo actualizado
29         tiempo.EstablecerTiempo( 99, 99, 99 );
30         Console.WriteLine( "Después de probar configuraciones inválidas:" );
31         Console.Write( "Hora universal: " );
32         Console.WriteLine( tiempo.AStringUniversal() );
33         Console.Write( "Hora estándar: " );
34         Console.WriteLine( tiempo.ToString() );
35     } // fin de Main
36 } // fin de la clase PruebaEspacioNombresTiempo1

```

```

La hora universal inicial es: 00:00:00
La hora estándar inicial es: 12:00:00 AM

La hora universal después de EstablecerTiempo es: 13:27:06
La hora estándar después de EstablecerTiempo es: 1:27:06 PM

Después de probar configuraciones inválidas:
Hora universal: 00:00:00
Hora estándar: 12:00:00 AM

```

Figura 9.19 | Uso de un objeto `Tiempo1` en una aplicación.

pueda utilizarse por todo el código en el mismo ensamblado que la clase, pero no por el código en otros ensamblados. Dentro del mismo ensamblado que la clase, esto equivale al acceso `public`. No obstante, si se hace referencia a una biblioteca de clases desde una aplicación, las clases `internal` de la biblioteca estarán inaccesibles desde el código de la aplicación. De manera similar, los métodos, las variables de instancia y otros miembros de una clase que se declara como `internal` son accesibles para todo el código compilado en el mismo ensamblado, pero no para el código en otros ensamblados.

La aplicación de la figura 9.20 demuestra el acceso `internal`. Esta aplicación contiene dos clases en un archivo de código fuente: la clase de aplicación `PruebaAccesoInternal` (líneas 6-22) y la clase `DatosInternal` (líneas 25-43).

En la declaración de la clase `DatosInternal`, las líneas 27-28 declaran las variables de instancia `numero` y `mensaje` con el modificador de acceso `internal`; la clase `DatosInternal` tiene el acceso `internal` de manera predeterminada, por lo que no hay necesidad de un modificador de acceso. El método `static Main` de `PruebaAccesoInternal` crea una instancia de la clase `DatosInternal` (línea 10) para demostrar cómo se modifican las variables de instancia de `DatosInternal` de manera directa (como se muestra en las líneas 16-17). Dentro del mismo ensamblado, el acceso `internal` es equivalente al acceso `public`. Los resultados se pueden ver en la ventana de resultados. Si compilamos esta clase en un archivo `.dll` de biblioteca de clases y la referenciamos desde una nueva aplicación, esa aplicación tendrá acceso a la clase `public` `PruebaAccesoInternal`, pero no a la clase `internal` `DatosInternal` o a sus miembros `internal`.

```

1  // Fig. 9.20: PruebaAccesoInternal.cs
2  // Los miembros que se declaran como internal en una clase, son accesibles para
3  // otras clases en el mismo ensamblado.
4  using System;
5
6  public class PruebaAccesoInternal
7  {
8      public static void Main(string[] args)
9      {
10         DatosInternal datosInternal = new DatosInternal();
11
12         // imprime representación string de datosInternal
13         Console.WriteLine( "Después de instanciar:\n{0}", datosInternal );
14
15         // modifica los datos de acceso internal en datosInternal
16         datosInternal.numero = 77;
17         datosInternal.mensaje = "Adiós";
18
19         // imprime representación string de datosInternal
20         Console.WriteLine( "\nDespués de modificar valores:\n{0}", datosInternal );
21     } // fin de Main
22 } // fin de la clase PruebaAccesoInternal
23
24 // clase con variables de instancia con acceso internal
25 class DatosInternal
26 {
27     internal int numero; // variable de instancia con acceso internal
28     internal string mensaje; // variable de instancia con acceso internal
29
30     // constructor
31     public DatosInternal()
32     {
33         numero = 0;
34         mensaje = "Hola";
35     } // fin del constructor de DatosInternal
36

```

Figura 9.20 | Los miembros que se declaren como `internal` en una clase son accesibles para otras clases en el mismo ensamblado. (Parte I de 2).

```

37 // devuelve representación string del objeto DatosInternal
38 public override string ToString()
39 {
40     return string.Format(
41         "numero: {0}; mensaje: {1}", numero, mensaje);
42     } // fin del método ToString
43 } // fin de la clase DatosInternal

```

Después de instanciar:
numero: 0; mensaje: Hola

Después de modificar valores:
numero: 77; mensaje: Adiós

Figura 9.20 | Los miembros que se declaren como `internal` en una clase son accesibles para otras clases en el mismo ensamblado. (Parte 2 de 2).

9.16 Vista de clases y Examinador de objetos

Ahora que hemos introducido los conceptos clave de la programación orientada a objetos, le presentaremos dos características que proporciona Visual Studio para facilitar el diseño de las aplicaciones orientadas a objetos: *Vista de clases* y *Examinador de objetos*.

Uso de la ventana Vista de clases

La ventana **Vista de clases** muestra los campos y los métodos para todas las clases en un proyecto. Para acceder a esta característica, seleccione **Vista de clases** del menú **Ver**. La figura 9.21 muestra la **Vista de clases** para el proyecto **Tiempo1** de la figura 9.1 (clase **Tiempo1**) y la figura 9.2 (clase **PruebaTiempo1**). La vista sigue una estructura jerárquica, en donde posiciona el nombre del proyecto (**Tiempo1**) como la raíz e incluye una serie de nodos que representan las clases, variables y métodos en el proyecto. Si aparece un signo más (+) a la izquierda de un nodo, ese nodo se puede expandir para mostrar otros. Si aparece un signo menos (-) a la izquierda de un nodo, ese nodo se puede contraer. De acuerdo con la **Vista de clases** el proyecto **Tiempo1** contiene la clase **Tiempo1** y la clase **PruebaTiempo1** como hijos. Cuando se selecciona la clase **Tiempo1**, aparecen sus miembros en la mitad inferior de la ventana. La clase **Tiempo1** contiene los métodos **EstablecerTiempo**, **ToString** y **AStringUniversal** (los cuales se indican mediante cuadros color púrpura) y las variables de instancia **hora**, **minuto** y **segundo** (que se indican mediante cuadros color azul). Los iconos de candado, que se encuentran a la izquierda de los iconos de cuadros color azul para las variables de instancia, especifican que estas variables son `private`. La clase **PruebaTiempo1** contiene el método **Main**. Observe que tanto la clase **Tiempo1** como la clase **PruebaTiempo1** contienen

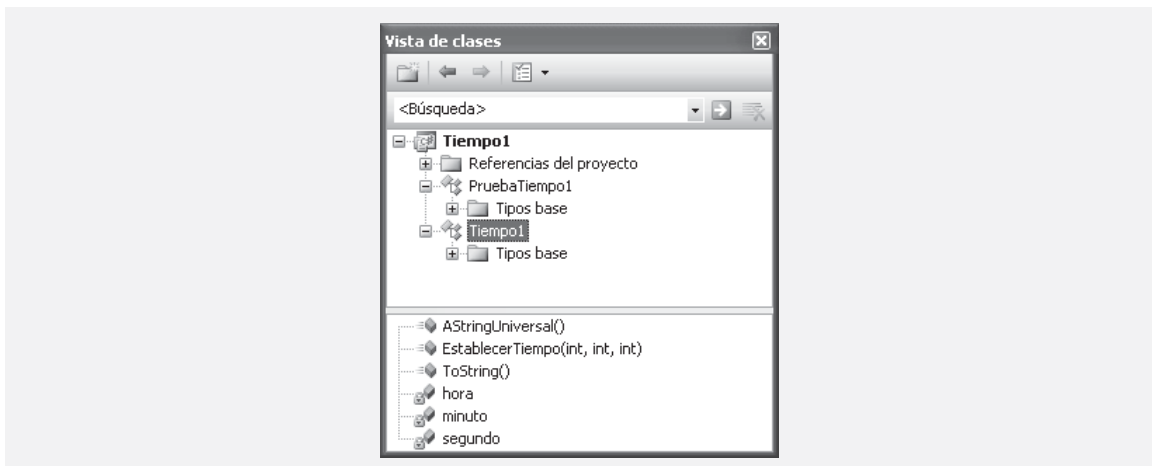


Figura 9.21 | La **Vista de clases** de la clase **Tiempo1** (figura 9.1) y de la clase **PruebaTiempo1** (figura 9.2).

el nodo **Tipos base**. Si expande este nodo, podrá ver la clase `Object` en cada caso, ya que cada clase hereda de la clase `System.Object` (que veremos en el capítulo 10).

Uso del Examinador de objetos

El **Examinador de objetos** de Visual C# Express lista todas las clases en la biblioteca de C#. Puede utilizar el **Examinador de objetos** para aprender acerca de la funcionalidad que ofrece una clase específica. Para abrir el **Examinador de objetos**, seleccione **Otras ventanas** del menú **Ver** y haga clic en **Examinador de objetos**. La figura 9.22 ilustra el **Examinador de objetos** cuando el usuario navega a la clase `Math` en el espacio de nombres `System`, en el ensamblado `mscorlib` (Microsoft Core Library). [Nota: tenga cuidado de no confundir el espacio de nombres `System` con el ensamblado llamado `System`. El ensamblado `System` describe a otros miembros del espacio de nombres `System`, pero la clase `System.Math` está en `mscorlib`.] El **Examinador de objetos** lista todos los métodos que proporciona la clase `Math` en el marco superior derecho; esto le ofrece un “acceso instantáneo” a la información relacionada con la funcionalidad de varios objetos. Si hace clic en el nombre de un miembro en el marco superior derecho, aparece una descripción de ese miembro en el marco inferior derecho. Observe también que el **Examinador de objetos** lista en el marco izquierdo a todas las clases de la FCL. El **Examinador de objetos** puede ser un mecanismo rápido para aprender acerca de una clase o del método de una clase. Recuerde que también puede ver la descripción completa de una clase o de un método en la documentación en línea disponible a través del menú **Ayuda** en Visual C# Express.

9.17 (Opcional) Caso de estudio de ingeniería de software: inicio de la programación de las clases del sistema ATM

En las secciones del Caso de estudio de ingeniería de software de los capítulos 1 y del 3 al 8, introducimos los fundamentos de la orientación a objetos y desarrollamos un diseño orientado a objetos para nuestro sistema ATM. En los capítulos del 4 al 7 presentamos la programación orientada a objetos en C#. En el capítulo 8 vimos un análisis más detallado acerca de los detalles de la programación con clases. Ahora comenzaremos a implementar nuestro diseño orientado a objetos; para ello convertiremos los diagramas de clases en código de C#. En la sección final del Caso de estudio de ingeniería de software (sección 11.9) modificaremos el código para incorporar los conceptos orientados a objetos de herencia y polimorfismo. En el apéndice J presentamos la implementación completa del código en C#.

Visibilidad

Ahora vamos a aplicar modificadores de acceso a los miembros de nuestras clases. En el capítulo 4 presentamos los modificadores de acceso `public` y `private`. Los modificadores de acceso determinan la *visibilidad*, o accesibilidad, de los atributos y operaciones de un objeto para otros objetos. Antes de empezar a implementar nuestro diseño, debemos considerar cuáles atributos y métodos de nuestras clases deben ser `public` y cuáles deben ser `private`.

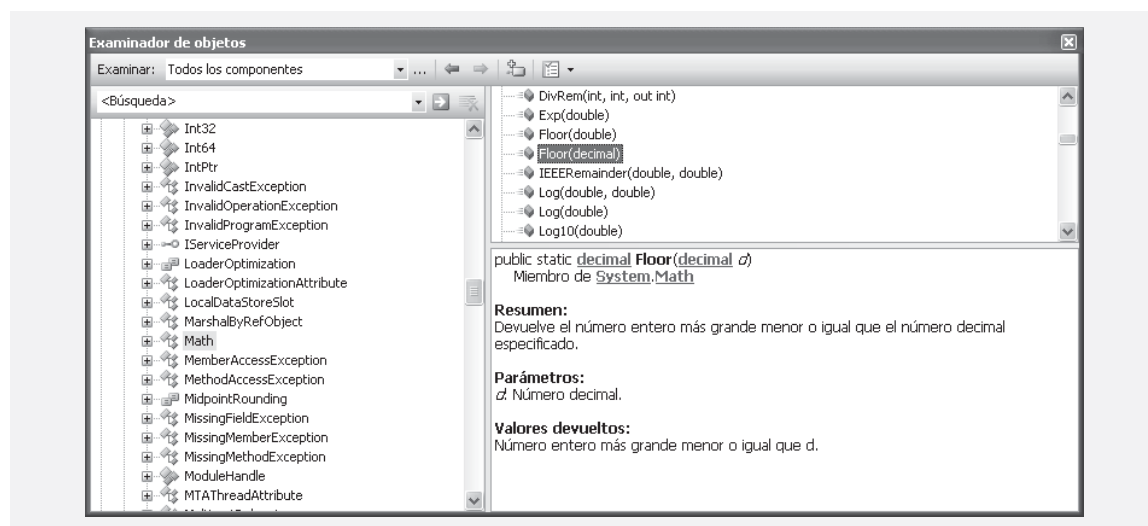


Figura 9.22 | Examinador de objetos para la clase `Math`.

En el capítulo 4 observamos que por lo general los atributos deben ser `private` y que los métodos invocados por los clientes de una clase deben ser `public`. No obstante, los métodos que se llaman sólo por otros métodos de la clase como “funciones utilitarias” deben ser `private`. UML emplea *marcadores de visibilidad* para modelar la visibilidad de los atributos y las operaciones. La visibilidad pública se indica mediante la colocación de un signo más (+) antes de una operación o atributo; un signo menos (–) indica una visibilidad privada. La figura 9.23 muestra nuestro diagrama de clases actualizado, en el cual se incluyen los marcadores de visibilidad. [Nota: no incluimos parámetros de operación en la figura 9.23. Esto es perfectamente normal. Agregar los marcadores de visibilidad no afecta a los parámetros que ya están modelados en los diagramas de clases de las figuras 7.20 a 7.24.]

Navegabilidad

Antes de empezar a implementar nuestro diseño en C#, presentaremos una notación adicional de UML. El diagrama de clases de la figura 9.24 refina aún más las relaciones entre las clases del sistema ATM, al agregar flechas de navegabilidad a las líneas de asociación. Las *flechas de navegabilidad* (representadas como flechas con puntas delgadas en el diagrama de clases) indican en qué dirección puede recorrerse una asociación, y se basan en las colaboraciones modeladas en los diagramas de comunicación y de secuencia (vea la sección 8.14). Al implementar un sistema diseñado mediante el uso de UML, los programadores utilizan flechas de navegabilidad para ayudar a determinar cuáles objetos necesitan referencias a otros objetos. Por ejemplo, la flecha de navegabilidad que apunta de la clase ATM a la clase BaseDatosBanco indica que podemos navegar de una a la otra, con lo cual se permite a la clase ATM invocar a las operaciones de BaseDatosBanco. No obstante, como la figura 9.24 no

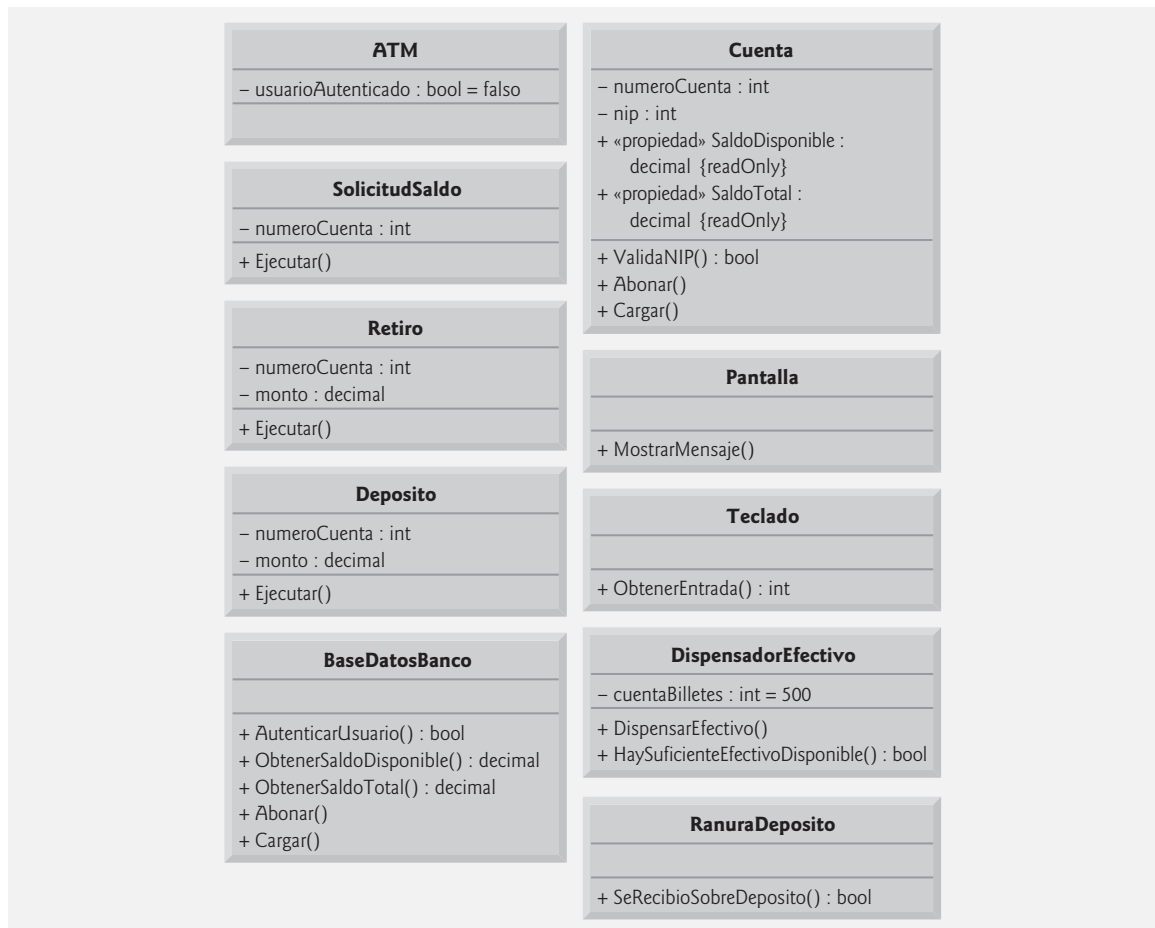


Figura 9.23 | Diagrama de clases con marcadores de visibilidad.

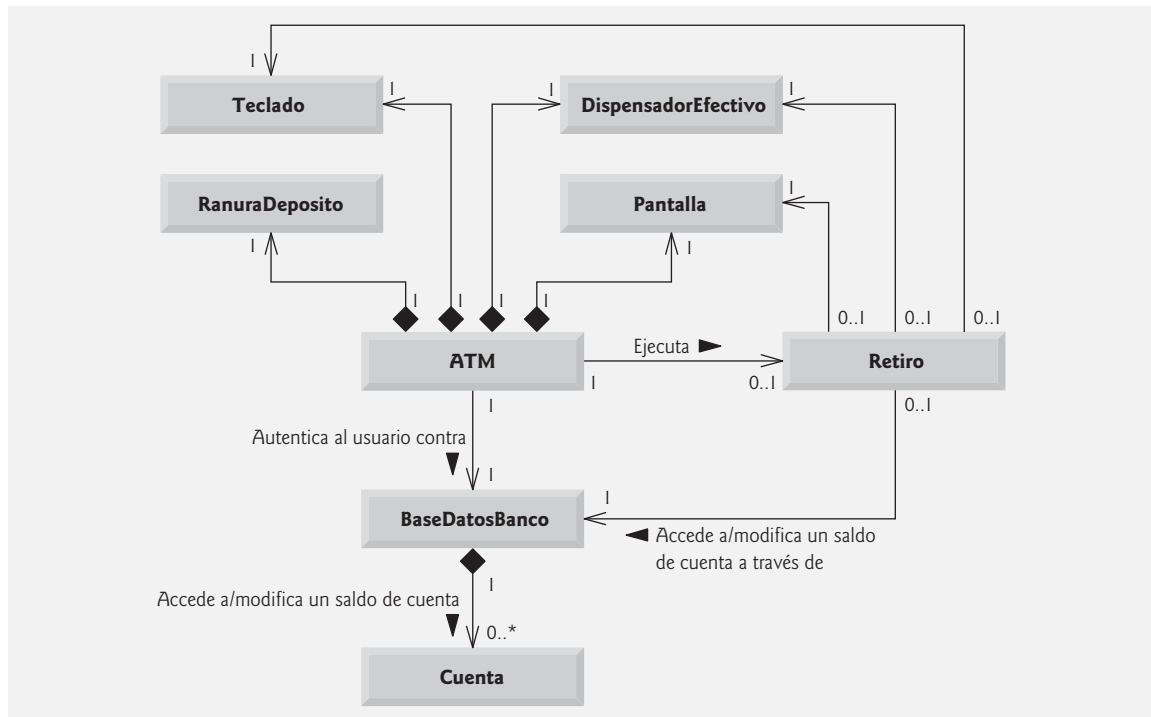


Figura 9.24 | Diagrama de clases con flechas de navegabilidad.

contiene una flecha de navegabilidad que apunte de la clase `BaseDatosBanco` a la clase `ATM`, la clase `BaseDatosBanco` no puede acceder a las operaciones de la clase `ATM`. Observe que las asociaciones en un diagrama de clases que tiene flechas de navegabilidad en ambos extremos, o que no tiene ninguna flecha de navegabilidad, indica una **navegabilidad bidireccional**: la navegación puede proceder en cualquier dirección a lo largo de la asociación.

Al igual que el diagrama de clases de la figura 4.24, el de la figura 9.24 omite las clases `SolicitudSaldo` y `Deposito` para simplificarlo. La navegabilidad de las asociaciones en las que participan estas dos clases se asemeja mucho a la navegabilidad de las asociaciones de la clase `Retiro`. En la sección 4.11 vimos que `SolicitudSaldo` tiene una asociación con la clase `Pantalla`. Podemos navegar de la clase `SolicitudSaldo` a la clase `Pantalla` a lo largo de esta asociación, pero no podemos navegar de la clase `Pantalla` a la clase `SolicitudSaldo`. Por ende, si modeláramos la clase `SolicitudSaldo` en la figura 9.24, colocaríamos una flecha de navegabilidad en el extremo de la clase `Pantalla` de esta asociación. Recuerde también que la clase `Deposito` se asocia con las clases `Pantalla`, `Teclado` y `RanuraDeposito`. Podemos navegar de la clase `Deposito` a cada una de estas clases, pero no al revés. Por lo tanto, podríamos colocar flechas de navegabilidad en los extremos de las clases `Pantalla`, `Teclado` y `RanuraDeposito` de estas asociaciones. [Nota: modelaremos estas clases y asociaciones adicionales en nuestro diagrama de clases final en la sección 11.9, una vez que hayamos simplificado la estructura de nuestro sistema al incorporar el concepto orientado a objetos de la herencia.]

Implementación del sistema ATM a partir de su diseño de UML

Ahora estamos listos para empezar a implementar el sistema ATM. Primero convertiremos las clases de los diagramas de las figuras 9.23 y 9.24 en código de C#. Este código representará el “esqueleto” del sistema. En el capítulo 11 modificaremos el código para incorporar el concepto orientado a objetos de la herencia. En el apéndice J, Código del caso de estudio del ATM, presentaremos el código de C# completo y funcional que implementa nuestro diseño orientado a objetos.

Como ejemplo, empezaremos a desarrollar el código para la clase `Retiro` a partir de nuestro diseño de la clase `Retiro` en la figura 9.23. Utilizamos esta figura para determinar los atributos y operaciones de la clase. Usamos el modelo de UML en la figura 9.24 para determinar las asociaciones entre las clases. Seguiremos estos cuatro lineamientos para cada clase:

1. Use el nombre que se localiza en el primer compartimiento de una clase en un diagrama de clases, para declarar la clase como `public` con un constructor sin parámetros vacío; incluimos este constructor tan sólo como un receptáculo para recordarnos que la mayoría de las clases necesitará uno o más constructores. En el apéndice J, en donde completamos una versión funcional de esta clase, agregaremos todos los argumentos y el código necesarios al cuerpo del constructor. Al principio, la clase `Retiro` produce el código de la figura 9.25. [Nota: si encontramos que las variables de instancia de la clase sólo requieren la inicialización predeterminada, eliminaremos el constructor sin parámetros vacío, ya que es innecesario.]
2. Use los atributos que se localizan en el segundo compartimiento de la clase para declarar las variables de instancia. Los atributos `private numeroCuenta` y `monto` de la clase `Retiro` producen el código de la figura 9.26.
3. Use las asociaciones descritas en el diagrama de clases para declarar las referencias a otros objetos. De acuerdo con la figura 9.24, `Retiro` puede acceder a un objeto de la clase `Pantalla`, a un objeto de la clase `Teclado`, a un objeto de la clase `DispensadorEfectivo` y a un objeto de la clase `BaseDatosBanco`. La clase `Retiro` debe mantener referencias a estos objetos y enviarles mensajes, por lo que las líneas 10-13 de la figura 9.27 declaran las referencias apropiadas como variables de instancia `private`. En la implementación de la clase `Retiro` en el apéndice J, un constructor inicializa estas variables de instancia con referencias a los objetos actuales.
4. Use las operaciones que se localizan en el tercer compartimiento de la figura 9.23 para declarar las armazones de los métodos. Si todavía no hemos especificado un tipo de valor de retorno para una operación, declaramos el método con el tipo de retorno `void`. Consulte los diagramas de clases de las figuras 7.20 a 7.24 para declarar cualquier parámetro necesario. Al agregar la operación `public Ejecutar` (que tiene una lista de parámetros vacía) en la clase `Retiro` se produce el código en las líneas 23-26 de la figura 9.28. [Nota: codificaremos los cuerpos de los métodos cuando implementemos el sistema ATM completo en el apéndice J.]

```

1 // Fig. 9.25: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM
3 public class Retiro
4 {
5     // constructor sin parámetros
6     public Retiro()
7     {
8         // código del cuerpo del constructor
9     } // fin del constructor
10 } // fin de la clase Retiro

```

Figura 9.25 | Código de C# inicial para la clase `Retiro`, con base en las figuras 9.23 y 9.24.

```

1 // Fig. 9.26: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM
3 public class Retiro
4 {
5     // atributos
6     private int numeroCuenta; // cuenta de la que se van a retirar los fondos
7     private decimal monto; // monto que se va a retirar de la cuenta
8
9     // constructor sin parámetros
10    public Retiro()
11    {
12        // código del cuerpo del constructor
13    } // fin del constructor
14 } // fin de la clase Retiro

```

Figura 9.26 | C# incorpora variables `private` para la clase `Retiro`, con base en las figuras 9.23 y 9.24.


```

1 // Fig. 9.27: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM
3 public class Retiro
4 {
5     // atributos
6     private int numeroCuenta; // cuenta de la que se van a retirar los fondos
7     private decimal monto; // monto a retirar
8
9     // referencias a los objetos asociados
10    private Pantalla pantalla; // pantalla del ATM
11    private Teclado teclado; // teclado del ATM
12    private DispensadorEfectivo dispensadorEfectivo; // dispensador de efectivo del ATM
13    private BaseDatosBanco baseDatosBanco; // base de datos de información de las cuentas
14
15    // constructor sin parámetros
16    public Retiro()
17    {
18        // código del cuerpo del constructor
19    } // fin del constructor
20 } // fin de la clase Retiro

```

Figura 9.27 | Código de C# que incorpora manejadores de referencias `private` para las asociaciones de la clase `Retiro`, con base en las figuras 9.23 y 9.24.

```

1 // Fig. 9.28: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM
3 public class Retiro
4 {
5     // atributos
6     private int numeroCuenta; // cuenta de la que se van a retirar los fondos
7     private decimal monto; // monto a retirar
8
9     // referencias a los objetos asociados
10    private Pantalla pantalla; // pantalla del ATM
11    private Teclado teclado; // teclado del ATM
12    private DispensadorEfectivo dispensadorEfectivo; // dispensador de efectivo del ATM
13    private BaseDatosBanco baseDatosBanco; // base de datos de información de las cuentas
14
15    // constructor sin parámetros
16    public Retiro()
17    {
18        // código del cuerpo del constructor
19    } // fin del constructor
20
21    // operaciones
22    // realiza una transacción
23    public void Ejecutar()
24    {
25        // código del cuerpo del método Ejecutar
26    } // fin del método Ejecutar
27 } // fin de la clase Retiro

```

Figura 9.28 | Código de C# que incorpora el método `Ejecutar` en la clase `Retiro`, con base en las figuras 9.23 y 9.24.



Observación de ingeniería de software 9.13

Muchas herramientas de modelado de UML pueden convertir los diseños basados en UML en código de C#, con lo que se agiliza en forma considerable el proceso de implementación. Para obtener más información acerca de estos generadores “automáticos” de código, consulte los recursos Web que se listan al final de la sección 3.10.

Esto concluye nuestra discusión acerca de los fundamentos de la generación de archivos de clases a partir de diagramas de UML. En la sección final del Caso de estudio de ingeniería de software (sección 11.9), demostraremos cómo modificar el código en la figura 9.28 para incorporar los conceptos orientados a objetos de la herencia y el polimorfismo, que presentaremos en los capítulos 10 y 11, respectivamente.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 9.1** Indique si el siguiente enunciado es *verdadero* o *falso*, y si es *falso*, explique por qué: si un atributo de una clase se marca con un signo menos (-) en un diagrama de clases, el atributo no es directamente accesible fuera de la clase.
- 9.2** En la figura 9.24, la asociación entre los objetos ATM y Pantalla indica:
- a) que podemos navegar de la Pantalla al ATM.
 - b) que podemos navegar del ATM a la Pantalla.
 - c) a y b; la asociación es bidireccional.
 - d) Ninguna de las anteriores.
- 9.3** Escriba código en C# para empezar a implementar el diseño para la clase Cuenta.

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 9.1** Verdadero. El signo negativo (-) indica una visibilidad privada.
- 9.2** b.
- 9.3** El diseño para la clase Cuenta produce el código en la figura 9.29. Observe que incluimos las variables de instancia `private` llamadas `saldoDisponible` y `saldoTotal` para almacenar los datos que manipularán las propiedades `SaldoDisponible` y `SaldoTotal`, y los métodos `Abonar` y `Cargar`.

```

1  // Fig. 9.29: Cuenta.cs
2  // La clase Cuenta representa una cuenta bancaria.
3  public class Cuenta
4  {
5      private int numeroCuenta; // número de cuenta
6      private int nip; // NIP para la autenticación
7      private decimal saldoDisponible; // monto de retiro disponible
8      private decimal saldoTotal; // fondos disponibles + depósito pendiente
9
10     // constructor sin parámetros
11     public Cuenta()
12     {
13         // código del cuerpo del constructor
14     } // fin del constructor
15
16     // valida NIP del usuario
17     public bool ValidarNIP()
18     {
19         // código del cuerpo del método ValidarNIP
20     } // fin del método ValidarNIP
21
22     // propiedad de sólo lectura que obtiene el saldo disponible
23     public decimal SaldoDisponible
24     {
25         get
26         {
27             // código del cuerpo del descriptor de acceso get de SaldoDisponible
28         } // fin de get
29     } // fin de la propiedad SaldoDisponible
30

```

Figura 9.29 | Código de C# para la clase Cuenta, con base en las figuras 9.23 y 9.24. (Parte 1 de 2).

```

31 // propiedad de sólo lectura que obtiene el saldo total
32 public decimal SaldoTotal
33 {
34     get
35     {
36         // código del cuerpo del descriptor de acceso get de SaldoTotal
37     } // fin de get
38 } // fin de la propiedad SaldoTotal
39
40 // abona a la cuenta
41 public void Abonar()
42 {
43     // código del cuerpo del método Abonar
44 } // fin del método Abonar
45
46 // carga a la cuenta
47 public void Cargar()
48 {
49     // código del cuerpo del método Cargar
50 } // fin del método Cargar
51 } // fin de la clase Cuenta

```

Figura 9.29 | Código de C# para la clase Cuenta, con base en las figuras 9.23 y 9.24. (Parte 2 de 2).

9.18 Conclusión

En este capítulo hablamos sobre conceptos adicionales de las clases. El caso de estudio de la clase `Tiempo` presentó una declaración de clase completa que consiste de datos `private`, constructores `public` sobrecargados para flexibilidad en la inicialización, propiedades para manipular los datos y métodos de la clase que devuelven representaciones `string` de un objeto `Tiempo` en dos formatos distintos. Aprendió que toda clase puede declarar un método `ToString` que devuelve una representación `string` de un objeto de la clase, y que este método puede invocarse en forma implícita cuando se imprime en pantalla un objeto de una clase en la forma de un objeto `string`.

Aprendió que la referencia `this` se utiliza en forma implícita en los métodos no `static` de una clase para acceder a las variables de instancia de la clase y a otros métodos no `static`. Vio usos explícitos de la referencia `this` para acceder a los miembros de la clase (incluyendo los campos ocultos) y aprendió a utilizar la palabra clave `this` en un constructor para llamar a otro constructor de la clase. También aprendió a declarar indexadores con la palabra clave `this`, con lo cual se puede acceder a los datos de un objeto en forma muy parecida a la manera en que se accede a los elementos de un arreglo.

Vio que una clase puede tener referencias a los objetos de otras clases como miembros; un concepto conocido como composición. Aprendió acerca de la capacidad de recolección de basura de C# y cómo reclama la memoria de los objetos que ya no se utilizan. Explicamos la motivación para las variables `static` en una clase, y le demostramos cómo declarar y utilizar variables y métodos `static` en sus propias clases. También aprendió a declarar e inicializar variables `readonly`.

Le mostramos cómo crear una biblioteca de clases para reutilizar componentes, y cómo utilizar las clases de la biblioteca en una aplicación. Aprendió que las clases que se declaran sin un modificador de acceso reciben un acceso `internal` de manera predeterminada. Vio que las clases en un ensamblado pueden acceder a los miembros con acceso `internal` de las otras clases dentro del mismo ensamblado. También le mostramos cómo utilizar las ventanas **Vista de clases** y **Examinador de objetos** de Visual Studio para navegar por las clases de la FCL y sus propias aplicaciones, para descubrir información acerca de esas clases.

En el siguiente capítulo aprenderá otra tecnología clave de programación orientada a objetos: la herencia. En ese capítulo verá que todas las clases en C# se relacionan en forma directa o indirecta con la clase `Object`. También empezará a comprender cómo la herencia le permite crear aplicaciones más poderosas en menos tiempo.

10

Programación orientada a objetos: herencia

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Cómo la herencia promueve la reutilización de código.
- Los conceptos de las clases bases y las clases derivadas.
- Crear una clase derivada que herede los atributos y comportamientos de una clase base.
- Utilizar el modificador de acceso `protected` para otorgar a los métodos de la clase derivada el acceso a los miembros de la clase base.
- Cómo acceder a los miembros de la clase base con `base`.
- Cómo se utilizan los constructores en las jerarquías de herencias.
- Los métodos de la clase `object`, la clase base directa o indirecta de todas las clases.

No digas que conoces a otro por completo, sino hasta que hayas dividido una herencia con él.

—Johann Kasper Lavater

Este método es definir como el número de una clase, la clase de todas las clases similares a la clase dada.

—Bertrand Russell

Es bueno heredar una biblioteca, pero es mejor coleccionar una.

—Augustine Birrell

Preserva la autoridad base de los libros de otros.

—William Shakespeare

- 10.1 Introducción
- 10.2 Clases base y clases derivadas
- 10.3 Miembros `protected`
- 10.4 Relación entre las clases base y las clases derivadas
 - 10.4.1 Creación y uso de una clase `EmpleadoPorComision`
 - 10.4.2 Creación de una clase `EmpleadoBaseMasComision` sin usar la herencia
 - 10.4.3 Creación de una jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision`
 - 10.4.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected`
 - 10.4.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `private`
- 10.5 Los constructores en las clases derivadas
- 10.6 Ingeniería de software mediante la herencia
- 10.7 La clase `object`
- 10.8 Conclusión

10.1 Introducción

En este capítulo continuaremos nuestra discusión acerca de la programación orientada a objetos (POO), analizaremos una de sus características principales: la *herencia*, que es una forma de reutilización de software, en la cual para crear una nueva clase se absorben los miembros de una clase existente y se mejoran con capacidades nuevas o modificadas. Con la herencia, los programadores ahorran tiempo durante el desarrollo de aplicaciones, al reutilizar software de alta calidad, probado y depurado. Esto también incrementa la probabilidad de que el sistema se implemente en forma efectiva.

Al crear una clase, en lugar de declarar miembros completamente nuevos, puede hacer que la nueva clase herede los miembros de una ya existente. La clase existente se llama *clase base*, y la clase nueva es la *clase derivada*. Esta última puede convertirse en la clase base para clases derivadas en el futuro.

Por lo general, una clase derivada agrega sus propios campos y métodos. Por lo tanto, es más específica que su clase base y representa un grupo más especializado de objetos. Es común que la clase derivada exhiba los comportamientos de su clase base y comportamientos adicionales, específicos para la clase derivada.

Cuando la clase base hereda, en forma explícita, a partir de una clase derivada se conoce como *clase base directa*. Por otro lado, una *clase base indirecta* es la que se encuentra por encima de la clase base directa en la *jerarquía de clases*, que define las relaciones de herencia entre las clases. La jerarquía de clases empieza con la clase `object` (que es el alias en C# para `System.Object` en la Biblioteca de clases del .NET Framework), de la cual *toda* clase se *extiende* (o “hereda de”), ya sea en forma directa o indirecta. La sección 10.7 lista los métodos de la clase `object`, que todas las demás clases heredan. En el caso de la *herencia simple*, una clase se deriva de una clase base directa. A diferencia de C++, C# no soporta la herencia múltiple (que ocurre cuando una clase se deriva de más de una clase base directa). En el capítulo 11, Polimorfismo, interfaces y sobrecarga de operadores, le explicaremos cómo puede utilizar las interfaces para que se dé cuenta de los muchos de los beneficios de la herencia múltiple, y al mismo tiempo evite los problemas asociados.

La experiencia en la creación de sistemas de software indica qué cantidades considerables de código tratan con casos especiales muy relacionados. Cuando los programadores se preocupan por los casos especiales, los detalles pueden oscurecer el panorama en general. Con la programación orientada a objetos, los programadores pueden (cuando sea apropiado) enfocarse en las características comunes entre los objetos en el sistema, en vez de los casos especiales.

Hay una diferencia entre la *relación “es un”* y la *relación “tiene un”*. “Es un” representa la herencia. En una relación del tipo “es un”, un objeto de una clase derivada también puede tratarse como un objeto de su clase base. Por ejemplo, un auto *es un* vehículo, y un camión *es un* vehículo. En contraste, “tiene un” representa la composición (vea el capítulo 9). En una relación del tipo “tiene un”, un objeto contiene como miembros refe-

rencias a otros objetos. Por ejemplo, un auto *tiene un* volante, y un objeto auto *tiene una* referencia a un objeto volante.

Las nuevas clases pueden heredar de las clases en *bibliotecas de clases*. Las organizaciones desarrollan sus propias bibliotecas de clases y pueden aprovechar las otras bibliotecas disponibles en todo el mundo. Algún día, es probable que la mayoría del software se construya a partir de *componentes reutilizables estándar*, así como se construyen hoy en día los automóviles y la mayoría del hardware de computadora. Esto facilitará el desarrollo de software más poderoso, abundante y económico.

10.2 Clases base y clases derivadas

A menudo, un objeto de una clase *es un* objeto de otra clase también. Por ejemplo, en geometría, un rectángulo *es un* cuadrilátero (así como los cuadrados, los paralelogramos y los trapezoides). Por ende, se puede decir que la clase Rectángulo hereda de la clase Cuadrilátero. En este contexto, la Cuadrilátero es una clase base y Rectángulo es una clase derivada. Un rectángulo *es un* tipo específico de cuadrilátero, pero es incorrecto decir que todo cuadrilátero *es un* rectángulo; el cuadrilátero podría ser un paralelogramo o cualquier otra figura. La figura 10.1 lista varios ejemplos simples de las clases base y las clases derivadas; observe que las primeras tienden a ser “más generales”; y las segundas, “más específicas”.

Como todo objeto de una clase derivada *es un* objeto de su clase base, y una clase base puede tener muchas clases derivadas, el conjunto de objetos representados por una clase base es por lo general mayor que el conjunto de objetos representado por cualquiera de sus clases derivadas. Por ejemplo, la clase base Vehículo representa a todos los vehículos: autos, camiones, barcos, bicicletas, etc. En contraste, la clase derivada Auto representa un subconjunto más pequeño y específico de vehículos.

Las relaciones de herencia forman estructuras jerárquicas similares a los árboles (figuras 10.2 y 10.3). Una clase base existe en una relación jerárquica con sus clases derivadas. Cuando varias clases participan en relaciones de herencia, se “afilian” con otras. Una clase se convierte ya sea en una clase base, suministrando miembros a las otras clases, o en una clase derivada, heredando sus miembros de otra clase. En algunos casos, una clase es tanto base como derivada.

Desarrollaremos una jerarquía de clases simple, también llamada *jerarquía de herencia* (figura 10.2). El diagrama de clases de UML de la figura 10.2 muestra una comunidad universitaria que tiene muchos tipos de miembros, incluyendo empleados, estudiantes y ex alumnos. Los empleados pueden ser docentes o administrativos. Los miembros docentes pueden ser administradores (como los decanos y los directores de departamento) o maestros. Observe que la jerarquía podría contener muchas otras clases. Por ejemplo, los estudiantes pueden ser graduados o no graduados. Los estudiantes no graduados pueden ser de primer año, de segundo, de tercero o de cuarto año.

Cada flecha con una punta triangular hueca en el diagrama de jerarquía representa una relación “*es un*”. Al seguir las flechas en esta jerarquía de clases podemos decir, por ejemplo, que “un Empleado *es un* MiembroDeLaComunidad” y que “un Maestro *es un* miembro Docente”. MiembroDeLaComunidad es la clase base directa de Empleado, Estudiante y ExAlumno, y es una clase base indirecta de todas las demás clases en el diagrama. Empezando desde la parte inferior del diagrama, usted puede seguir las flechas y aplicar la relación “*es un*” hasta la clase base superior. Por ejemplo, un Administrador *es un* miembro Docente, *es un* Empleado y *es un* MiembroDeLaComunidad.

Clase base	Clases derivadas
Estudiante	EstudianteGraduado, EstudianteNoGraduado
Figura	Círculo, Triángulo, Rectángulo
Prestamo	PrestamoAuto, PrestamoMejoraHogar, PrestamoHipotecario
Empleado	Docente, Administrativo, TrabajadorPorHoras, TrabajadorPorComision
CuentaBanco	CuentaCheques, CuentaAhorros

Figura 10.1 | Ejemplos de herencia.

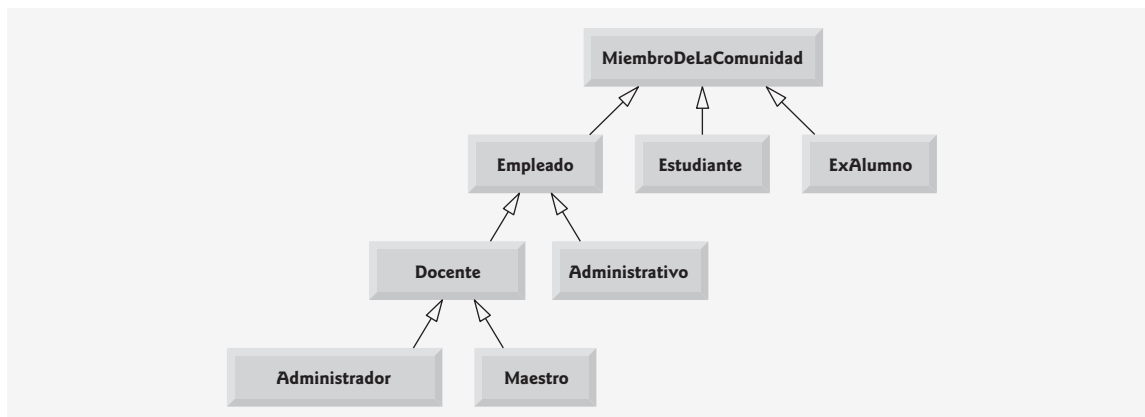


Figura 10.2 | Diagrama de clases de UML que muestra una jerarquía de herencia para cada *MiembroDeLaComunidad* de una universidad.

Ahora considere la jerarquía de herencia para *Figura* en la figura 10.3. Esta jerarquía comienza con la clase base *Figura*, que se extiende mediante las clases derivadas *FiguraBidimensional* y *FiguraTridimensional* (un objeto *Figura* puede ser objeto *Bidimensional* u objeto *Tridimensional*). El tercer nivel de esta jerarquía contiene algunos objetos *FiguraBidimensional* y *FiguraTridimensional* específicos. Al igual que en la figura 10.2, podemos seguir las flechas desde la parte inferior del diagrama de clases hasta la clase base superior en esta jerarquía de clases, para identificar varias relaciones del tipo *es un*. Por ejemplo, un *Triangulo* *es una* *FiguraBidimensional* y *es una* *Figura*, mientras que una *Esfera* *es una* *FiguraTridimensional* y *es una* *Figura*. Observe que esta jerarquía podría contener muchas otras clases. Por ejemplo, las elipses y los trapezoides son objetos de la clase *FiguraBidimensional*.

No todas las relaciones de clases son relaciones de herencia. En el capítulo 9 hablamos sobre la relación “*tiene un*”, en la cual las clases tienen miembros que son referencias a objetos de otras clases. Dichas relaciones crean clases mediante la composición de clases existentes. Por ejemplo, dadas las clases *Empleado*, *FechaNacimiento* y *NumeroTelefonico*, sería impropio decir que un *Empleado* *es una* *FechaNacimiento*, o que un *Empleado* *es un* *NumeroTelefonico*. No obstante, un *Empleado* *tiene una* *FechaNacimiento* y *tiene un* *NumeroTelefonico*.

Es posible tratar a los objetos de una clase base y a los de una clase derivada de manera similar; sus características comunes se expresan en los miembros de la clase base. Los objetos de todas las clases que extiendan a una clase base común pueden tratarse como objetos de esa clase base (es decir, dichos objetos tienen una relación del tipo “*es un*” con la clase base). No obstante, los objetos de la clase base no pueden tratarse como objetos de sus clases derivadas. Por ejemplo, todos los autos son vehículos, pero no todos los vehículos son autos (los otros vehículos podrían ser camiones, aviones o bicicletas, por ejemplo). En este capítulo y en el capítulo 11, Polimorfismo, interfaces y sobrecarga de operadores, consideraremos muchos ejemplos de las relaciones “*es un*”.

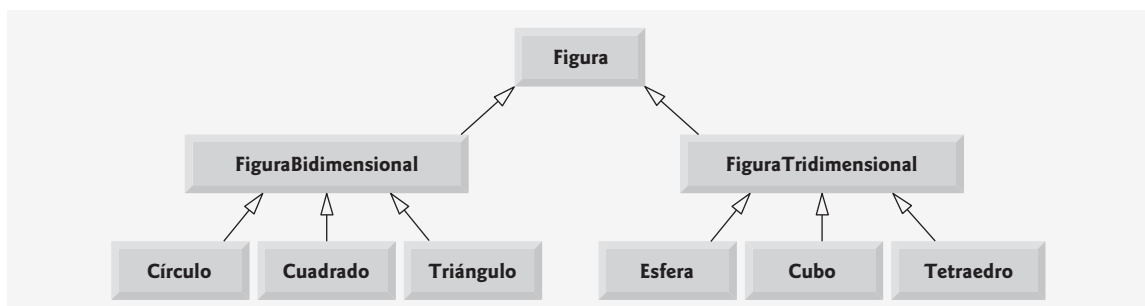


Figura 10.3 | Diagrama de clases que muestra una jerarquía de herencia para objetos *Figura*.

Uno de los problemas con la herencia es que una clase derivada puede heredar métodos que no necesita, o no debe tener. Aun cuando el método de una clase base sea apropiado para una clase derivada, esa clase derivada necesita por lo general una versión personalizada del método. En dichos casos, la clase derivada puede *redefinir* el método de la clase base con una implementación apropiada, como veremos con frecuencia en los ejemplos de código del capítulo.

10.3 Miembros protected

En el capítulo 9 vimos los modificadores de acceso `public`, `private` e `internal`. Los miembros `public` de una clase son accesibles en cualquier parte en donde la aplicación tenga una referencia a un objeto de esa clase, o a una de sus clases derivadas. Los miembros `private` de una clase son accesibles sólo dentro de la misma clase. Los miembros `private` de una clase base son heredados por sus clases derivadas, pero no son directamente accesibles por los métodos y las propiedades de la clase derivada. En esta sección presentaremos el modificador de acceso `protected`. El uso del acceso `protected` ofrece un nivel intermedio de acceso entre `public` y `private`. Se puede acceder a los miembros `protected` de una clase base a través de los miembros de esa clase base y de los miembros de sus clases derivadas (los miembros de una clase también pueden declararse como **`protected internal`**. Es posible acceder a los miembros `protected internal` de una clase base a través de los miembros de esa clase base, por los miembros de sus clases derivadas y por cualquier clase dentro del mismo ensamblado).

Todos los miembros no `private` de la clase base retienen su modificador de acceso original cuando se convierten en miembros de la clase derivada (es decir, los miembros `public` de la clase base se convierten en miembros `public` de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros `protected` de la clase derivada).

Los métodos de una clase derivada pueden hacer referencia a los miembros `public` y `protected` heredados de la clase base con sólo usar los nombres de los miembros. Cuando un método de la clase derivada redefine a un método de la clase base, se puede acceder a la versión de la clase base del método desde la clase derivada si se antepone al nombre del método de la clase base la palabra clave `base` y el operador punto (`.`). En la sección 10.4 hablaremos sobre cómo acceder a los miembros redefinidos de la clase base.



Observación de ingeniería de software 10.1

Los métodos de una clase derivada no pueden acceder en forma directa a los miembros `private` de la clase base. Una clase derivada puede modificar el estado de los campos `private` de la clase base sólo a través de los métodos y propiedades no `private` que se proporcionan en la clase base.



Observación de ingeniería de software 10.2

Declarar campos `private` en una clase base le ayuda a probar, depurar y modificar sistemas en forma correcta. Si una clase derivada puede acceder a los campos `private` de su clase base, las clases que hereden de esa clase base podrían acceder también a los campos. Esto propagaría el acceso a los que deberían ser campos `private`, y se perderían los beneficios del ocultamiento de la información.

10.4 Relación entre las clases base y las clases derivadas

En esta sección usaremos una jerarquía de herencia que contiene tipos de empleados en la aplicación de nómina de una compañía, para hablar sobre la relación entre una clase base y sus clases derivadas. En esta compañía, a los empleados por comisión (que se representan como objetos de una clase base) se les paga un porcentaje de sus ventas, mientras que los empleados por comisión con salario base (que se representan como objetos de una clase derivada) reciben un salario base, más un porcentaje de sus ventas.

Dividiremos nuestra discusión sobre la relación entre los empleados por comisión y los empleados por comisión con salario base en cinco ejemplos. El primero declara la clase `EmpleadoPorComision`, que hereda directamente de la clase `object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión y el monto de ventas en bruto (es decir, total).

El segundo ejemplo declara la clase `EmpleadoBaseMasComision`, que también hereda directamente de la clase `object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión, el monto de ventas en bruto y el salario base. Para crear esta última clase,

escribiremos cada línea de código que ésta requiera; pronto veremos que es mucho más eficiente crear esta clase haciendo que herede de la clase `EmpleadoPorComision`.

El tercer ejemplo declara una clase `EmpleadoBaseMasComision2` separada, la cual extiende a la clase `EmpleadoPorComision` (es decir, un `EmpleadoBaseMasComision2` es un `EmpleadoPorComision` que también tiene un salario base). Demostraremos que los métodos de la clase base deben declararse explícitamente como `virtual`, si van a ser redefinidos por los métodos en las clases derivadas. La clase `EmpleadoBaseMasComision2` trata de acceder a los miembros `private` de la clase `EmpleadoPorComision`; esto produce errores de compilación, ya que una clase derivada no puede acceder a las variables de instancia `private` de su clase base.

El cuarto ejemplo muestra que si las variables de instancia de la clase base `EmpleadoPorComision` se declaran como `protected`, una clase `EmpleadoBaseMasComision3` que extiende a la clase `EmpleadoPorComision2` puede acceder a los datos de manera directa. Para este fin, declaramos la clase `EmpleadoPorComision2` con variables de instancia `protected`. Todas las clases `EmpleadoBasePorComision` contienen una funcionalidad idéntica, pero le mostraremos que la clase `EmpleadoBaseMasComision3` es más fácil de crear y manipular.

Una vez que hablemos sobre la conveniencia de utilizar variables de instancia `protected`, crearemos el quinto ejemplo, que establece las variables de instancia de `EmpleadoPorComision` de vuelta a `private` en la clase `EmpleadoPorComision3`, para hacer cumplir la buena ingeniería de software. Después le mostraremos cómo una clase `EmpleadoBaseMasComision4` separada, que extiende a la clase `EmpleadoPorComision3`, puede utilizar los métodos públicos de `EmpleadoPorComision3` para manipular las variables de instancia `private` de `EmpleadoPorComision3`.

10.4.1 Creación y uso de una clase `EmpleadoPorComision`

Comenzaremos por declarar la clase `EmpleadoPorComision` (figura 10.4). La línea 3 empieza la declaración de la clase. El signo de dos puntos (`:`) seguido por el nombre de clase `object` al final del encabezado de la declaración indica que la clase `EmpleadoPorComision` extiende a la clase `object` (`System.Object` en la FCL). Los programadores de C# utilizan la herencia para crear clases a partir de clases existentes. De hecho, todas las clases en C# (excepto `object`) extienden a una clase existente. Como la clase `EmpleadoPorComision` extiende la clase `object`, la clase `EmpleadoPorComision` hereda los métodos de la clase `object`; la clase `object` no tiene campos. Cada clase en C# hereda en forma directa o indirecta los métodos de `object`. Si una clase no especifica que hereda de otra clase, la nueva clase hereda de `object` en forma implícita. Por esta razón, es común que los programadores no incluyan `“: object”` en su código; en nuestro ejemplo lo haremos por fines demostrativos.

```

1  // Fig. 10.4: EmpleadoPorComision.cs
2  // La clase EmpleadoPorComision representa a un empleado por comisión.
3  public class EmpleadoPorComision : object
4  {
5      private string primerNombre;
6      private string apellidoPaterno;
7      private string numeroSeguroSocial;
8      private decimal ventasBrutas; // ventas semanales totales
9      private decimal tarifaComision; // porcentaje de comisión
10
11     // constructor con cinco parámetros
12     public EmpleadoPorComision( string nombre, string apellido, string nss,
13         decimal ventas, decimal tarifa )
14     {
15         // la llamada implícita al constructor del objeto ocurre aquí
16         primerNombre = nombre;
17         apellidoPaterno = apellido;
18         numeroSeguroSocial = nss;
19         VentasBrutas = ventas; // valida las ventas brutas mediante una propiedad
20         TarifaComision = tarifa; // valida la tarifa de comisión mediante una propiedad
21     } // fin del constructor de EmpleadoPorComision con cinco parámetros

```

Figura 10.4 | La clase `EmpleadoPorComision` representa a un empleado por comisión. (Parte I de 3).

```

22
23 // propiedad de sólo lectura que obtiene el primer nombre del empleado por comisión
24 public string PrimerNombre
25 {
26     get
27     {
28         return primerNombre;
29     } // fin de get
30 } // fin de la propiedad PrimerNombre
31
32 // propiedad de sólo lectura que obtiene el apellido paterno del empleado por
    comisión
33 public string ApellidoPaterno
34 {
35     get
36     {
37         return apellidoPaterno;
38     } // fin de get
39 } // fin de la propiedad ApellidoPaterno
40
41 // propiedad de sólo lectura que obtiene el
42 // número de seguro social del empleado por comisión
43 public string NumeroSeguroSocial
44 {
45     get
46     {
47         return numeroSeguroSocial;
48     } // fin de get
49 } // fin de la propiedad NumeroSeguroSocial
50
51 // propiedad que obtiene y establece las ventas brutas del empleado por comisión
52 public decimal VentasBrutas
53 {
54     get
55     {
56         return ventasBrutas;
57     } // fin de get
58     set
59     {
60         ventasBrutas = ( value < 0 ) ? 0 : value;
61     } // fin de set
62 } // fin de la propiedad VentasBrutas
63
64 // propiedad que obtiene y establece la tarifa de comisión del empleado por comisión
65 public decimal TarifaComision
66 {
67     get
68     {
69         return tarifaComision;
70     } // fin de get
71     set
72     {
73         tarifaComision = ( value > 0 && value < 1 ) ? value : 0;
74     } // fin de set
75 } // fin de la propiedad TarifaComision
76
77 // calcula el salario del empleado por comisión
78 public decimal Ingresos()

```

Figura 10.4 | La clase `EmpleadoPorComision` representa a un empleado por comisión. (Parte 2 de 3).

```

79     {
80         return tarifaComision * ventasBrutas;
81     } // fin del método Ingresos
82
83     // devuelve representación string del objeto EmpleadoPorComision
84     public override string ToString()
85     {
86         return string.Format(
87             "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}",
88             "empleado por comisión", PrimerNombre, ApellidoPaterno,
89             "número de seguro social", NumeroSeguroSocial,
90             "ventas brutas", VentasBrutas, "tarifa de comisión", TarifaComision );
91     } // fin del método ToString
92 } // fin de la clase EmpleadoPorComision

```

Figura 10.4 | La clase `EmpleadoPorComision` representa a un empleado por comisión. (Parte 3 de 3).



Observación de ingeniería de software 10.3

El compilador establece la clase base de una clase a `object` cuando la declaración de la clase no extiende explícitamente una clase base.

Los servicios `public` de la clase `EmpleadoPorComision` incluyen un constructor (líneas 12-21) y los métodos `Ingresos` (líneas 78-81) y `ToString` (líneas 84-91). Las líneas 24-75 declaran propiedades `public` para manipular las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la clase (las cuales se declaran en las líneas 5-9). La clase `EmpleadoPorComision` declara cada una de sus variables de instancia como `private`, por lo que los objetos de otras clases no pueden acceder directamente a estas variables. Declarar las variables de instancia como `private` y proporcionar propiedades `public` para manipular y validar estas variables ayuda a cumplir con la buena ingeniería de software. Por ejemplo, los descriptores de acceso `set` de las propiedades `VentasBrutas` y `TarifaComision` validan sus argumentos antes de asignar los valores a las variables de instancia `ventasBrutas` y `tarifaComision`, en forma respectiva.

Los constructores no se heredan, por lo que la clase `EmpleadoPorComision` no hereda el constructor de la clase `object`. Sin embargo, el constructor de la clase `EmpleadoPorComision` llama al constructor de la clase `object` de manera implícita. De hecho, la primer tarea del constructor de cualquier clase derivada es llamar al constructor de su clase base directa, ya sea en forma explícita o implícita (si no se especifica una llamada al constructor), para asegurar que las variables de instancia heredadas de la clase base se inicialicen en forma apropiada. En la sección 10.4.3 hablaremos sobre la sintaxis para llamar al constructor de una clase base en forma explícita. Si el código no incluye una llamada explícita al constructor de la clase base, el compilador genera una llamada implícita al constructor predeterminado o sin parámetros de la clase base. El comentario en la línea 15 de la figura 10.4 indica en dónde se hace la llamada implícita al constructor predeterminado de la clase base `object` (usted no necesita escribir el código para esta llamada). El constructor predeterminado (vacío) de la clase `object` no hace nada. Observe que aun si una clase no tiene constructores, el constructor predeterminado que declara el compilador de manera implícita para la clase llamará al constructor predeterminado o sin parámetros de la clase base. La clase `object` es la única clase que no tiene una clase base.

Una vez que se realiza la llamada implícita al constructor de `object`, las líneas 16-20 del constructor de `EmpleadoPorComision` asignan valores a las variables de instancia de la clase. Observe que no validamos los valores de los argumentos `nombre`, `apellido` y `nss` antes de asignarlos a las variables de instancia correspondientes. Sin duda podríamos validar el nombre y el apellido; tal vez asegurarnos de que tengan una longitud razonable. De manera similar, podría validarse un número de seguro social, para garantizar que contenga nueve dígitos, con o sin guiones cortos (por ejemplo, 123-45-6789 o 123456789).

El método `Ingresos` (líneas 78-81) calcula los ingresos de un `EmpleadoPorComision`. La línea 80 multiplica la `tarifaComision` por las `ventasBrutas` y devuelve el resultado.

El método `ToString` (líneas 84-91) es especial: es uno de los métodos que hereda cualquier clase de manera directa o indirecta de la clase `object`, que es la raíz de la jerarquía de clases de C#. La sección 10.7 muestra un

resumen de los métodos de la clase `object`. El método `ToString` devuelve un `string` que representa a un objeto. Una aplicación llama a este método de manera implícita cada vez que un objeto debe convertirse en una representación `string`, como en el método `Write` de `Console` o en el método `string.Format`, en el que se utiliza un elemento de formato. El método `ToString` de la clase `object` devuelve un `string` que incluye el nombre de la clase del objeto. En esencia, es un receptáculo que puede redefinirse (y por lo general así es) por una clase derivada para especificar una representación `string` apropiada de los datos en un objeto de la clase derivada. El método `ToString` de la clase `EmpleadoPorComision` redefine al método `ToString` de la clase `object`. Al invocarse, el método `ToString` de `EmpleadoPorComision` usa el método `string.Format` para devolver un `string` que contiene información acerca del `EmpleadoPorComision`. Utilizamos el especificador de formato `C` para dar formato a `ventasBrutas` como cantidad monetaria, y el especificador de formato `F2` para dar formato a `tarifaComision` con dos dígitos de precisión a la derecha del punto decimal. Para redefinir a un método de una clase base, una clase derivada debe declarar al método con la palabra clave **`override`** y con la misma firma (nombre del método, número de parámetros y tipos de los parámetros) y tipo de valor de retorno que el método de la clase base; el método `ToString` de `object` no recibe parámetros y devuelve el tipo `string`, por lo que `EmpleadoPorComision` declara a `ToString` sin parámetros y con el tipo de valor de retorno `string`.



Error común de programación 10.1

Es un error de compilación redefinir a un método con un modificador de acceso distinto. Tenga en cuenta que si se redefine un método con un modificador de acceso más restrictivo se romperá la relación "es un". Si un método `public` pudiera redefinirse como `protected` o `private`, los objetos de la clase derivada no podrían responder a las mismas llamadas a métodos que los objetos de la clase base. Una vez que se declara un método en una clase base, el método debe tener el mismo modificador de acceso para todas las clases derivadas en forma directa o indirecta de esa clase.

La figura 10.5 prueba la clase `EmpleadoPorComision`. Las líneas 10-11 crean una instancia de un objeto `EmpleadoPorComision` e invocan a su constructor (líneas 12-21 de la figura 10.4) para inicializarlo con "Sue" como el primer nombre, "Jones" como el apellido, "222-22-2222" como el número de seguro social, 10000.00M como el monto de ventas brutas y .06M como la tarifa de comisión. Adjuntamos el sufijo `M` al monto de ventas brutas y a la tarifa de comisión para indicar que deben interpretarse como literales `decimal`, y no como `double`. Las líneas 16-25 utilizan las propiedades de `EmpleadoPorComision` para extraer los valores de las variables de instancia del objeto e imprimirlas en pantalla. Las líneas 26-27 imprimen en pantalla el monto calculado por el método `Ingresos`. Las líneas 29-30 invocan a los descriptores de acceso `set` de las propiedades `VentasBrutas` y `TarifaComision` del objeto para modificar los valores de las variables de instancia `ventasBrutas` y `tarifaComision`. Las líneas 32-33 imprimen en pantalla la representación `string` del `EmpleadoPorComision` actualizado. Observe que cuando se imprime un objeto en pantalla utilizando un elemento de formato, se invoca de manera implícita al método `ToString` del objeto para obtener su representación `string`. La línea 34 imprime en pantalla los ingresos otra vez.

```

1 // Fig. 10.5: PruebaEmpleadoPorComision.cs
2 // Prueba de la clase EmpleadoPorComision.
3 using System;
4
5 public class PruebaEmpleadoPorComision
6 {
7     public static void Main( string[] args )
8     {
9         // crea instancia de objeto EmpleadoPorComision
10        EmpleadoPorComision empleado = new EmpleadoPorComision( "Sue",
11        "Jones", "222-22-2222", 10000.00M, .06M );
12
13        // muestra datos del empleado por comisión
14        Console.WriteLine(

```

Figura 10.5 | Prueba de la clase `EmpleadoPorComision`. (Parte 1 de 2).

```

15         "Información del empleado obtenida por las propiedades y los métodos: \n" );
16     Console.WriteLine( "{0} {1}", "El primer nombre es",
17         empleado.PrimerNombre );
18     Console.WriteLine( "{0} {1}", "El apellido es",
19         empleado.ApellidoPaterno );
20     Console.WriteLine( "{0} {1}", "El número de seguro social es",
21         empleado.NumeroSeguroSocial );
22     Console.WriteLine( "{0} {1:C}", "Las ventas brutas son",
23         empleado.VentasBrutas );
24     Console.WriteLine( "{0} {1:F2}", "La tarifa de comisión es",
25         empleado.TarifaComision );
26     Console.WriteLine( "{0} {1:C}", "Los ingresos son",
27         empleado.Ingresos() );
28
29     empleado.VentasBrutas = 5000.00M; // establece las ventas brutas
30     empleado.TarifaComision = .1M; // establece la tarifa de comisión
31
32     Console.WriteLine( "\n{0}:\n\n{1}",
33         "Se actualizó la información del empleado obtenida por ToString", empleado);
34     Console.WriteLine( "ingresos: {0:C}", empleado.Ingresos() );
35 } // fin de Main
36 } // fin de la clase PruebaEmpleadoPorComision

```

Información del empleado obtenida por las propiedades y los métodos:

El primer nombre es Sue
 El apellido es Jones
 El número de seguro social es 222-22-2222
 Las ventas brutas son \$10,000.00
 La tarifa de comisión es 0.06
 Los ingresos son: \$600.00

Se actualizó la información del empleado obtenida por ToString:

empleado por comisión: Sue Jones
 número de seguro social: 222-22-2222
 ventas brutas: \$5,000.00
 tarifa de comisión: 0.10
 ingresos: \$500.00

Figura 10.5 | Prueba de la clase `EmpleadoPorComision`. (Parte 2 de 2).

10.4.2 Creación de una clase `EmpleadoBaseMasComision` sin usar la herencia

Ahora hablaremos sobre la segunda parte de nuestra introducción a la herencia, mediante la declaración y prueba de la clase (completamente nueva e independiente) `EmpleadoBaseMasComision` (figura 10.6), la cual contiene los siguientes datos: primer nombre, apellido paterno, número de seguro social, monto de ventas brutas, tarifa de comisión y salario base. Los servicios `public` de la clase `EmpleadoBaseMasComision` incluyen un constructor (líneas 14-24), y los métodos `Ingresos` (líneas 99-102) y `ToString` (líneas 105-113). Las líneas 28-96 declaran propiedades `public` para las variables de instancia `private` `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas`, `tarifaComision` y `salarioBase` para la clase (las cuales se declaran en las líneas 6-11). Estas variables, propiedades y métodos encapsulan todas las características necesarias de un empleado por comisión con sueldo base. Observe la similitud entre esta clase y la clase `EmpleadoPorComision` (figura 10.4); en este ejemplo, no explotamos todavía esa similitud.

Observe que la clase `EmpleadoBaseMasComision` no especifica que extiende a `object` con la sintaxis “: `object`” en la línea 4, por lo que se deduce que la clase extiende a `object` en forma implícita. Observe además que, al igual que el constructor de la clase `EmpleadoPorComision` (línea 12-21 de la figura 10.4), el constructor de la

clase `EmpleadoBaseMasComision` invoca al constructor predeterminado de la clase `object` en forma implícita, como se indica en el comentario de la línea 17 de la figura 10.6.

El método `Ingresos` de la clase `EmpleadoBaseMasComision` (líneas 99-102) calcula los ingresos de un empleado por comisión con salario base. La línea 101 devuelve el resultado de la suma del salario base del empleado con el producto de la tarifa de comisión y las ventas brutas del empleado.

La clase `EmpleadoBaseMasComision` redefine el método `ToString` de `object` para que devuelva un `string` que contiene la información del `EmpleadoBaseMasComision` (líneas 105-113). Una vez más, utilizamos el especificador de formato `C` para dar formato a las ventas brutas y al salario base como cantidades monetarias, y el especificador de formato `F2` para dar formato a la tarifa de comisión con dos dígitos de precisión a la derecha del punto decimal (línea 108).

```

1  // Fig. 10.6: EmpleadoBaseMasComision.cs
2  // La clase EmpleadoBaseMasComision representa a un empleado que recibe
3  // un salario base, además de una comisión.
4  public class EmpleadoBaseMasComision
5  {
6      private string primerNombre;
7      private string apellidoPaterno;
8      private string numeroSeguroSocial;
9      private decimal ventasBrutas; // ventas semanales totales
10     private decimal tarifaComision; // porcentaje de comisión
11     private decimal salarioBase; // salario base por semana
12
13     // constructor con seis parámetros
14     public EmpleadoBaseMasComision( string nombre, string apellido,
15         string nss, decimal ventas, decimal tarifa, decimal salario )
16     {
17         // la llamada implícita al constructor del objeto se realiza aquí
18         primerNombre = nombre;
19         apellidoPaterno = apellido;
20         numeroSeguroSocial = nss;
21         VentasBrutas = ventas; // valida las ventas brutas a través de una propiedad
22         TarifaComision = tarifa; // valida la tarifa de comisión a través de una
                                // propiedad
23         SalarioBase = salario; // valida el salario base a través de una propiedad
24     } // fin del constructor de EmpleadoBaseMasComision con seis parámetros
25
26     // propiedad de sólo lectura que obtiene
27     // el primer nombre del empleado por comisión con salario base
28     public string PrimerNombre
29     {
30         get
31         {
32             return primerNombre;
33         } // fin de get
34     } // fin de la propiedad PrimerNombre
35
36     // propiedad de sólo lectura que obtiene
37     // el apellido paterno del empleado por comisión con salario base
38     public string ApellidoPaterno
39     {
40         get
41         {
42             return apellidoPaterno;

```

Figura 10.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión. (Parte I de 3).

```

43     } // fin de get
44 } // fin de la propiedad ApellidoPaterno
45
46 // propiedad de sólo lectura que obtiene
47 // el número de seguro social del empleado por comisión con salario base
48 public string NumeroSeguroSocial
49 {
50     get
51     {
52         return numeroSeguroSocial;
53     } // fin de get
54 } // fin de la propiedad NumeroSeguroSocial
55
56 // propiedad que obtiene y establece las
57 // ventas brutas del empleado por comisión con salario base
58 public decimal VentasBrutas
59 {
60     get
61     {
62         return ventasBrutas;
63     } // fin de get
64     set
65     {
66         ventasBrutas = ( value < 0 ) ? 0 : value;
67     } // fin de set
68 } // fin de la propiedad VentasBrutas
69
70 // propiedad que obtiene y establece la
71 // tarifa por comisión del empleado por comisión con salario base
72 public decimal TarifaComision
73 {
74     get
75     {
76         return tarifaComision;
77     } // fin de get
78     set
79     {
80         tarifaComision = ( value > 0 && value < 1 ) ? value : 0;
81     } // fin de set
82 } // fin de la propiedad TarifaComision
83
84 // propiedad que obtiene y establece
85 // el salario base del empleado por comisión con salario base
86 public decimal SalarioBase
87 {
88     get
89     {
90         return salarioBase;
91     } // fin de get
92     set
93     {
94         salarioBase = ( value < 0 ) ? 0 : value;
95     } // fin de set
96 } // fin de la propiedad SalarioBase
97
98 // calcula los ingresos
99 public decimal Ingresos()

```

Figura 10.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión. (Parte 2 de 3).


```

100     {
101         return SalarioBase + ( TarifaComision * VentasBrutas );
102     } // fin del método Ingresos
103
104     // devuelve representación string de EmpleadoBaseMasComision
105     public override string ToString()
106     {
107         return string.Format(
108             "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}",
109             "empleado por comisión con salario base", PrimerNombre, ApellidoPaterno,
110             "número de seguro social", NumeroSeguroSocial,
111             "ventas brutas", VentasBrutas, "tarifa de comisión", TarifaComision,
112             "salario base", SalarioBase );
113     } // fin del método ToString
114 } // fin de la clase EmpleadoBaseMasComision

```

Figura 10.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión. (Parte 3 de 3).

La figura 10.7 prueba la clase `EmpleadoBaseMasComision`. Las líneas 10-12 crean una instancia de un objeto `EmpleadoBaseMasComision` y pasan los parámetros "Bob", "Lewis", "333-33-3333", 5000.00M, .04M y 300.00M al constructor como el primer nombre, apellido paterno, número de seguro social, ventas brutas, tarifa de comisión y salario base, respectivamente. Las líneas 17-30 utilizan las propiedades y métodos de `EmpleadoBaseMasComision` para extraer los valores de las variables de instancia del objeto y calcular los ingresos para imprimirlos en pantalla. La línea 32 invoca a la propiedad `SalarioBase` del objeto para modificar el salario base. El descriptor de acceso `set` de la propiedad `SalarioBase` (figura 10.6, líneas 92-95) asegura que no se le asigne a la variable `SalarioBase` un valor negativo, ya que el salario base de un empleado no puede ser negativo. Las líneas 34-35 de la figura 10.7 invocan en forma implícita al método `ToString` del objeto, para obtener su representación string.

```

1  // Fig. 10.7: PruebaEmpleadoBaseMasComision.cs
2  // Prueba de la clase EmpleadoBaseMasComision.
3  using System;
4
5  public class PruebaEmpleadoBaseMasComision
6  {
7      public static void Main( string[] args )
8      {
9          // crea instancia de un objeto EmpleadoBaseMasComision
10         EmpleadoBaseMasComision empleado =
11             new EmpleadoBaseMasComision( "Bob", "Lewis",
12             "333-33-3333", 5000.00M, .04M, 300.00M );
13
14         // muestra en pantalla los datos del empleado por comisión con salario base
15         Console.WriteLine(
16             "Información del empleado obtenida por las propiedades y los métodos: \n" );
17         Console.WriteLine( "{0} {1}", "El primer nombre es",
18             empleado.PrimerNombre );
19         Console.WriteLine( "{0} {1}", "El apellido paterno es",
20             empleado.ApellidoPaterno );
21         Console.WriteLine( "{0} {1}", "El número de seguro social es",
22             empleado.NumeroSeguroSocial );
23         Console.WriteLine( "{0} {1:C}", "Las ventas brutas son",
24             empleado.VentasBrutas );

```

Figura 10.7 | Prueba de la clase `EmpleadoBaseMasComision`. (Parte 1 de 2).


```

25     Console.WriteLine( "{0} {1:F2}", "La tarifa de comisión es",
26         empleado.TarifaComision );
27     Console.WriteLine( "{0} {1:C}", "Los ingresos son",
28         empleado.Ingresos() );
29     Console.WriteLine( "{0} {1:C}", "El salario base es",
30         empleado.SalarioBase );
31
32     empleado.SalarioBase = 1000.00M; // establece el salario base
33
34     Console.WriteLine( "\n{0}:\n\n{1}",
35         "Información actualizada del empleado, obtenida mediante ToString", empleado );
36     Console.WriteLine( "ingresos: {0:C}", empleado.Ingresos() );
37 } // fin de Main
38 } // fin de la clase PruebaEmpleadoBaseMasComision

```

Información del empleado obtenida por las propiedades y los métodos:

El primer nombre es Bob
 El apellido paterno es Lewis
 El número de seguro social es 333-33-3333
 Las ventas brutas son \$5,000.00
 La tarifa de comisión es 0.04
 Los ingresos son \$500.00
 El salario base es \$300.00

Información actualizada del empleado, obtenida mediante ToString:

empleado por comisión con salario base: Bob Lewis
 número de seguro social: 333-33-3333
 ventas brutas: \$5,000.00
 tarifa de comisión: 0.04
 salario base: \$1,000.00
 ingresos: \$1,200.00

Figura 10.7 | Prueba de la clase `EmpleadoBaseMasComision`. (Parte 2 de 2).

Observe que la mayor parte del código para la clase `EmpleadoBaseMasComision` (figura 10.6) es similar, si no es que idéntico, al código para la clase `EmpleadoPorComision` (figura 10.4). Por ejemplo, en la clase `EmpleadoBaseMasComision`, las variables de instancia `private primerNombre` y `apellidoPaterno`, y las propiedades `PrimerNombre` y `ApellidoPaterno` son idénticas a las de la clase `EmpleadoPorComision`. Las clases `EmpleadoPorComision` y `EmpleadoBasePorComision` también contienen las variables de instancia `private numeroSeguroSocial`, `tarifaComision` y `ventasBrutas`, así como propiedades para manipular estas variables. Además, el constructor de `EmpleadoBasePorComision` es casi idéntico al de la clase `EmpleadoPorComision`, sólo que el constructor de `EmpleadoBaseMasComision` también establece el `salarioBase`. Las demás adiciones a la clase `EmpleadoBaseMasComision` son: la variable de instancia `private salarioBase` y la propiedad `SalarioBase`. El método `ToString` de la clase `EmpleadoBaseMasComision` es casi idéntico al de la clase `EmpleadoPorComision`, excepto que el método `ToString` de `EmpleadoBasePorComision` también da formato al valor de la variable de instancia `salarioBase` como cantidad monetaria.

Literalmente hablando, copiamos todo el código de la clase `EmpleadoPorComision` y lo pegamos en la clase `EmpleadoBaseMasComision`, después modificamos esta clase para incluir un salario base, y los métodos y propiedades que manipulan ese salario base. A menudo, este método de “copiar y pegar” está propenso a errores y consume mucho tiempo. Peor aún, se pueden esparcir muchas copias físicas del mismo código a lo largo de un sistema, con lo que el mantenimiento del código se convierte en una pesadilla. ¿Existe alguna manera de “absorber” los miembros de una clase, para que formen parte de otras clases sin tener que copiar el código? En los siguientes ejemplos responderemos a esta pregunta, utilizando un método más elegante para crear clases: la herencia.



Tip de prevención de errores 10.1

Copiar y pegar código de una clase a otra puede esparcir los errores a través de varios archivos de código fuente. Para evitar la duplicación de código (y posiblemente los errores) en situaciones en las que se desea que una clase “absorba” los miembros de otra, usamos la herencia en vez del método de “copiar y pegar”.



Observación de ingeniería de software 10.4

Con la herencia, los miembros comunes de todas las clases en la jerarquía se declaran en una clase base. Cuando se requieren modificaciones para estas características comunes, sólo hay que realizar los cambios en la clase base; así las clases derivadas heredan los cambios. Sin la herencia, habría que modificar todos los archivos de código fuente que contengan una copia del código en cuestión.

10.4.3 Creación de una jerarquía de herencia EmpleadoPorComision-EmpleadoBaseMasComision

Ahora declararemos la clase `EmpleadoBaseMasComision2` (figura 10.8), que extiende a la clase `EmpleadoPorComision` (figura 10.4). Un objeto `EmpleadoBaseMasComision` es un `EmpleadoPorComision` (ya que la herencia traspasa las capacidades de la clase `EmpleadoPorComision`), pero la clase `EmpleadoBaseMasComision2` también tiene la variable de instancia `salarioBase` (figura 10.8, línea 5). El signo de dos puntos (`:`) en la línea 3 de la declaración de la clase indica la herencia. Como clase derivada, `EmpleadoBaseMasComision2` hereda los miembros de la clase `EmpleadoPorComision` y puede acceder a los miembros que no son `private`. El constructor de la clase `EmpleadoPorComision` no se hereda. Por lo tanto, los servicios `public` de `EmpleadoBaseMasComision2` incluyen su constructor (líneas 9-14), los métodos y las propiedades `public` heredadas de la clase `EmpleadoPorComision`, la propiedad `SalarioBase` (líneas 18-28), el método `Ingresos` (líneas 31-35) y el método `ToString` (líneas 38-47).

```

1  // Fig. 10.8: EmpleadoBaseMasComision2.cs
2  // EmpleadoBaseMasComision2 hereda de la clase EmpleadoPorComision.
3  public class EmpleadoBaseMasComision2 : EmpleadoPorComision
4  {
5      private decimal salarioBase; // salario base por semana
6
7      // constructor de la clase derivada con seis parámetros
8      // con una llamada al constructor de la clase base EmpleadoPorComision
9      public EmpleadoBaseMasComision2( string nombre, string apellido,
10         string nss, decimal ventas, decimal tarifa, decimal salario )
11         : base( nombre, apellido, nss, ventas, tarifa )
12     {
13         SalarioBase = salario; // valida el salario base a través de una propiedad
14     } // fin del constructor de EmpleadoBaseMasComision2 con seis parámetros
15
16     // propiedad que obtiene y establece
17     // el salario base del empleado por comisión con salario base
18     public decimal SalarioBase
19     {
20         get
21         {
22             return salarioBase;
23         } // fin de get
24         set
25         {
26             salarioBase = ( value < 0 ) ? 0 : value;
27         } // fin de set
28     } // fin de la propiedad SalarioBase

```

Figura 10.8 | `EmpleadoBaseMasComision2` hereda de la clase `EmpleadoPorComision`. (Parte I de 2).

```

29
30 // calcula los ingresos
31 public override decimal Ingresos()
32 {
33     // no se permite: tarifaComision y ventasBrutas son private en la clase base
34     return salarioBase + ( tarifaComision * ventasBrutas );
35 } // fin del método Ingresos
36
37 // devuelve representación string de EmpleadoBaseMasComision2
38 public override string ToString()
39 {
40     // no se permite: trata de acceder a los miembros private de la clase base
41     return string.Format(
42         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}",
43         "empleado por comisión con salario base", primerNombre, apellidoPaterno,
44         "número de seguro social", numeroSeguroSocial,
45         "ventas brutas", ventasBrutas, "tarifa de comisión", tarifaComision,
46         "salario base", salarioBase );
47 } // fin del método ToString
48 } // fin de la clase EmpleadoBaseMasComision2

```



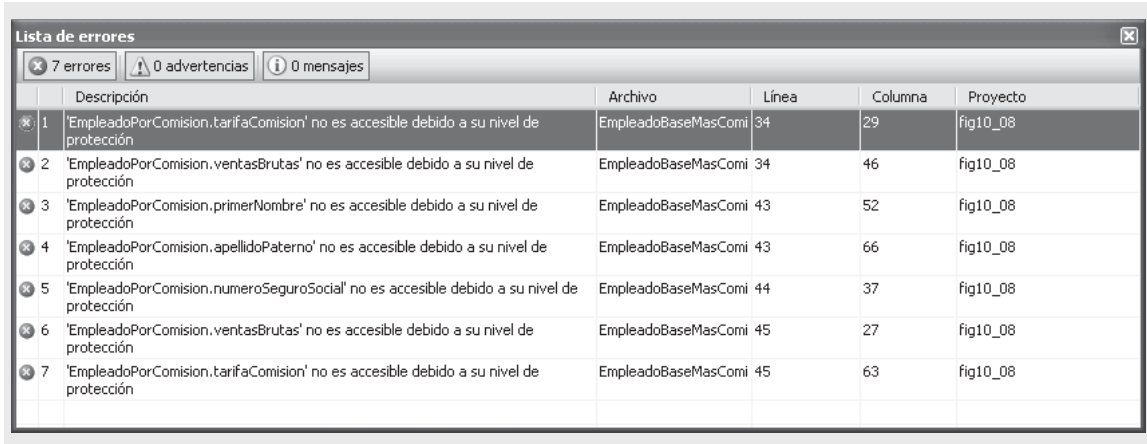
Figura 10.8 | EmpleadoBaseMasComision2 hereda de la clase EmpleadoPorComision. (Parte 2 de 2).

Cada constructor de clase derivada debe llamar en forma implícita o explícita al constructor de su clase base, para asegurar que las variables de instancia heredadas de la clase base se inicialicen en forma apropiada. El constructor de `EmpleadoBaseMasComision2` con seis parámetros llama en forma explícita al constructor de la clase `EmpleadoPorComision` con cinco parámetros, para inicializar la porción correspondiente a la clase base de un objeto `EmpleadoBaseMasComision2` (es decir, las variables `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`). La línea 11 en el encabezado del constructor de `EmpleadoBaseMasComision2` con seis parámetros invoca al constructor de `EmpleadoPorComision` con cinco parámetros (declarado en las líneas 12-21 de la figura 10.4) mediante el uso de un inicializador de constructor. En la sección 9.6 utilizamos inicializadores de constructores con la palabra clave `this` para llamar a los constructores sobrecargados en la misma clase. En la línea 11 de la figura 10.8, usamos un inicializador de constructor con la palabra clave `base` para invocar al constructor base. Los argumentos `nombre`, `apellido`, `nss`, `ventas` y `tarifa` se utilizan para inicializar a los miembros `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la clase base, respectivamente. Si el constructor de `EmpleadoBaseMasComision2` no invocara al constructor de `EmpleadoPorComision` de manera explícita, C# trataría de invocar al constructor predeterminado o sin parámetros de la clase `EmpleadoPorComision`; pero como la clase no tiene un constructor así, el compilador generaría un error. Cuando una clase base contiene un constructor sin parámetros, podemos usar `base()` en el inicializador del constructor para llamar a ese constructor de manera explícita, pero esto se hace muy raras veces.



Error común de programación 10.2

Si el constructor de una clase derivada llama a uno de los constructores de su clase base con argumentos que no concuerdan con el número y el tipo de los parámetros especificados en una de las declaraciones del constructor de la clase base, se produce un error de compilación.



	Descripción	Archivo	Línea	Columna	Proyecto
✖ 1	'EmpleadoPorComision.tarifaComision' no es accesible debido a su nivel de protección	EmpleadoBaseMasComi	34	29	fig10_08
✖ 2	'EmpleadoPorComision.ventasBrutas' no es accesible debido a su nivel de protección	EmpleadoBaseMasComi	34	46	fig10_08
✖ 3	'EmpleadoPorComision.primerNombre' no es accesible debido a su nivel de protección	EmpleadoBaseMasComi	43	52	fig10_08
✖ 4	'EmpleadoPorComision.apellidoPaterno' no es accesible debido a su nivel de protección	EmpleadoBaseMasComi	43	66	fig10_08
✖ 5	'EmpleadoPorComision.numeroSeguroSocial' no es accesible debido a su nivel de protección	EmpleadoBaseMasComi	44	37	fig10_08
✖ 6	'EmpleadoPorComision.ventasBrutas' no es accesible debido a su nivel de protección	EmpleadoBaseMasComi	45	27	fig10_08
✖ 7	'EmpleadoPorComision.tarifaComision' no es accesible debido a su nivel de protección	EmpleadoBaseMasComi	45	63	fig10_08

Figura 10.9 | Errores de compilación generados por `EmpleadoBaseMasComision2` (figura 10.8), después de declarar el método `Ingresos` en la figura 10.4 con la palabra clave `virtual`.

Las líneas 31-35 de la figura 10.8 declaran el método `Ingresos` usando la palabra clave `override` para redefinir el método de `EmpleadoPorComision`, como hicimos con el método `ToString` en ejemplos anteriores. La línea 31 produce un error de compilación, que indica que no podemos redefinir el método `Ingresos` de la clase, ya que no está “marcado como `virtual`, `abstract` u `override`” en forma explícita. Las palabras clave **`virtual`** y **`abstract`** indican que un método de la clase base puede redefinirse en clases derivadas (como aprenderá en la sección 11.4, los métodos `abstract` son implícitamente `virtuales`). El modificador `override` declara que el método de una clase derivada redefine a un método `virtual` o `abstract` de la clase base. Este modificador también declara en forma implícita el método de la clase derivada como `virtual` y le permite ser redefinido en las clases derivadas, en niveles más abajo en la jerarquía de herencia.

Si agregamos la palabra clave `virtual` a la declaración del método `Ingresos` en la figura 10.4 y compilamos nuevamente el programa, aparecerán otros errores de compilación. Como se muestra en la figura 10.9, el compilador genera errores adicionales para la línea 34 de la figura 10.8, ya que las variables de instancia `tarifaComision` y `ventasBrutas` de la clase base `EmpleadoPorComision` son `private`; no se permite que los métodos de la clase derivada `EmpleadoBaseMasComision2` accedan a las variables de instancia `private` de la clase base `EmpleadoPorComision`. Observe que utilizamos texto en negritas en la figura 10.8 para indicar el código erróneo. El compilador genera errores adicionales en las líneas 43-45 del método `ToString` de `EmpleadoBaseMasComision2` por la misma razón. Los errores en `EmpleadoBaseMasComision2` podrían haberse prevenido mediante el uso de las propiedades `public` heredadas de la clase `EmpleadoPorComision`. Por ejemplo, la línea 34 podría haber invocado a los descriptores de acceso `get` de las propiedades `TarifaComision` y `VentasBrutas` para acceder a las variables de instancia `private` `tarifaComision` y `ventasBrutas` de `EmpleadoPorComision`, respectivamente. Las líneas 43-45 también podrían haber usado las propiedades apropiadas para extraer los valores de las variables de instancia de la clase base.

10.4.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected`

Para permitir que la clase `EmpleadoBaseMasComision` acceda directamente a las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la clase base, podemos declarar esos miembros como `protected` en la clase base. Como vimos en la sección 10.3, los miembros `protected` de una clase base *se* heredan por todas las clases derivadas de esa clase base. La clase `EmpleadoPorComision2` (figura 10.10) es una modificación de la clase `EmpleadoPorComision` (figura 10.4), que declara las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `protected`, en vez de `private` (figura 10.10, líneas 5-9). Como vimos en la sección 10.4.3, también declaramos el método `Ingresos` como `virtual` en la línea 78, de manera que `EmpleadoBaseMasComision` pueda redefinir el método. Además del cambio en el nombre de la clase (y por ende el cambio en el nombre del

```

1 // Fig. 10.10: EmpleadoPorComision2.cs
2 // EmpleadoPorComision2 con variables de instancia protected.
3 public class EmpleadoPorComision2
4 {
5     protected string primerNombre;
6     protected string apellidoPaterno;
7     protected string numeroSeguroSocial;
8     protected decimal ventasBrutas; // ventas semanales totales
9     protected decimal tarifaComision; // porcentaje de comisión
10
11     // constructor con cinco parámetros
12     public EmpleadoPorComision2( string nombre, string apellido, string nss,
13         decimal ventas, decimal tarifa )
14     {
15         // la llamada implícita al constructor del objeto se realiza aquí
16         primerNombre = nombre;
17         apellidoPaterno = apellido;
18         numeroSeguroSocial = nss;
19         VentasBrutas = ventas; // valida las ventas brutas a través de una propiedad
20         TarifaComision = tarifa; // valida la tarifa de comisión a través de una
            propiedad
21     } // fin del constructor de EmpleadoPorComision2 con cinco parámetros
22
23     // propiedad de sólo lectura que obtiene el primer nombre del empleado por comisión
24     public string PrimerNombre
25     {
26         get
27         {
28             return primerNombre;
29         } // fin de get
30     } // fin de la propiedad PrimerNombre
31
32     // propiedad de sólo lectura que obtiene el apellido paterno del empleado por
        comisión
33     public string ApellidoPaterno
34     {
35         get
36         {
37             return apellidoPaterno;
38         } // fin de get
39     } // fin de la propiedad ApellidoPaterno
40
41     // propiedad de sólo lectura que obtiene
42     // el número de seguro social del empleado por comisión
43     public string NumeroSeguroSocial
44     {
45         get
46         {
47             return numeroSeguroSocial;
48         } // fin de get
49     } // fin de la propiedad NumeroSeguroSocial
50
51     // propiedad que obtiene y establece las ventas brutas del empleado por comisión
52     public decimal VentasBrutas
53     {
54         get
55         {
56             return ventasBrutas;

```

Figura 10.10 | EmpleadoPorComision2 con las variables de instancia protected. (Parte 1 de 2).

```

57     } // fin de get
58     set
59     {
60         ventasBrutas = ( value < 0 ) ? 0 : value;
61     } // fin de set
62 } // fin de la propiedad VentasBrutas
63
64 // propiedad que obtiene y establece la tarifa de comisión del empleado por comisión
65 public decimal TarifaComision
66 {
67     get
68     {
69         return tarifaComision;
70     } // fin de get
71     set
72     {
73         tarifaComision = ( value > 0 && value < 1 ) ? value : 0;
74     } // fin de set
75 } // fin de la propiedad TarifaComision
76
77 // calcula el sueldo del empleado por comisión
78 public virtual decimal Ingresos()
79 {
80     return tarifaComision * ventasBrutas;
81 } // fin del método Ingresos
82
83 // devuelve representación string del objeto EmpleadoPorComision
84 public override string ToString()
85 {
86     return string.Format(
87         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}",
88         "empleado por comisión", primerNombre, apellidoPaterno,
89         "número de seguro social", numeroSeguroSocial,
90         "ventas brutas", ventasBrutas, "tarifa de comisión", tarifaComision );
91     } // fin del método ToString
92 } // fin de la clase EmpleadoPorComision2

```

Figura 10.10 | EmpleadoPorComision2 con las variables de instancia protected. (Parte 2 de 2).

constructor) a EmpleadoPorComision2, el resto de la declaración de la clase en la figura 10.10 es idéntico al de la figura 10.4.

Podríamos haber declarado las variables de instancia primerNombre, apellidoPaterno, numeroSeguroSocial, ventasBrutas y tarifaComision de la clase base EmpleadoPorComision2 como public para permitir que la clase derivada EmpleadoBaseMasComision2 pueda acceder a las variables de instancia de la clase base. No obstante, declarar variables de instancia public es una mala ingeniería de software, ya que permite el acceso sin restricciones a las variables de instancia, lo que incrementa considerablemente la probabilidad de errores. Con las variables de instancia protected, la clase derivada obtiene acceso a las variables de instancia, pero las clases que no se derivan de la clase base no pueden acceder a sus variables en forma directa.

La clase EmpleadoBaseMasComision3 (figura 10.11) es una modificación de la clase EmpleadoBaseMasComision2 (figura 10.8), que extiende a EmpleadoPorComision2 (línea 4) en vez de la clase EmpleadoPorComision. Los objetos de la clase EmpleadoBaseMasComision3 heredan las variables de instancia protected primerNombre, apellidoPaterno, numeroSeguroSocial, ventasBrutas y tarifaComision de EmpleadoPorComision2; ahora todas estas variables son miembros protected de EmpleadoBaseMasComision3. Como resultado, el compilador no genera errores al compilar la línea 34 del método Ingresos y las líneas 42-44 del método ToString. Si otra clase extiende a EmpleadoBasePorComision3, la nueva clase derivada también hereda los miembros protected.

```

1 // Fig. 10.11: EmpleadoBaseMasComision3.cs
2 // EmpleadoBaseMasComision3 hereda de EmpleadoPorComision2 y tiene
3 // acceso a los miembros protected de EmpleadoPorComision2.
4 public class EmpleadoBaseMasComision3 : EmpleadoPorComision2
5 {
6     private decimal salarioBase; // salario base por semana
7
8     // constructor de la clase derivada con seis parámetros
9     // con una llamada al constructor de la clase bases EmpleadoPorComision
10    public EmpleadoBaseMasComision3( string nombre, string apellido,
11        string nss, decimal ventas, decimal tarifa, decimal salario )
12        : base( nombre, apellido, nss, ventas, tarifa )
13    {
14        SalarioBase = salario; // valida el salario base a través de una propiedad
15    } // fin del constructor de EmpleadoBaseMasComision3 con seis parámetros
16
17    // propiedad que obtiene y establece
18    // el salario base del empleado por comisión con salario base
19    public decimal SalarioBase
20    {
21        get
22        {
23            return salarioBase;
24        } // fin de get
25        set
26        {
27            salarioBase = ( value < 0 ) ? 0 : value;
28        } // fin de set
29    } // fin de la propiedad SalarioBase
30
31    // calcula los ingresos
32    public override decimal Ingresos()
33    {
34        return salarioBase + ( tarifaComision * ventasBrutas );
35    } // fin del método Ingresos
36
37    // devuelve representación string de EmpleadoBaseMasComision3
38    public override string ToString()
39    {
40        return string.Format(
41            "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}",
42            "empleado por comisión con salario base", primerNombre, apellidoPaterno,
43            "número de seguro social", numeroSeguroSocial,
44            "ventas brutas", ventasBrutas, "tarifa de comisión", tarifaComision,
45            "salario base", salarioBase );
46    } // fin del método ToString
47 } // fin de la clase EmpleadoBaseMasComision3

```

Figura 10.11 | EmpleadoBaseMasComision3 hereda de EmpleadoPorComision2 y tiene acceso a los miembros protected de EmpleadoPorComision2.

La clase EmpleadoBaseMasComision3 no hereda el constructor de la clase EmpleadoPorComision2. No obstante, el constructor de la clase EmpleadoBaseMasComision3 con seis parámetros (líneas 10-15) llama al constructor de la clase EmpleadoPorComision2 con cinco parámetros, mediante un inicializador de constructor. El constructor de EmpleadoBaseMasComision3 con seis parámetros debe llamar en forma explícita al constructor de la clase EmpleadoPorComision2, ya que EmpleadoPorComision2 no proporciona un constructor sin parámetros que pueda invocarse en forma implícita.

La figura 10.12 utiliza un objeto `EmpleadoBaseMasComision3` para realizar las mismas tareas que realizó la figura 10.7 con un objeto `EmpleadoBaseMasComision` (figura 10.6). Observe que los resultados de las dos aplicaciones son idénticos. Aunque declaramos la clase `EmpleadoBaseMasComision` sin utilizar la herencia, y declaramos la clase `EmpleadoBaseMasComision3` utilizando la herencia, ambas clases proporcionan la misma funcionalidad. El código fuente para la clase `EmpleadoBaseMasComision3`, que tiene 47 líneas, es mucho más corto que el de la clase `EmpleadoBaseMasComision`, que tiene 114 líneas, debido a que la clase `EmpleadoBaseMasComision3` hereda la mayor parte de su funcionalidad de `EmpleadoPorComision2`, mientras que la clase `EmpleadoBaseMasComision` sólo hereda la funcionalidad de la clase `object`. Además, ahora sólo hay una copia de la funcionalidad del empleado por comisión declarada en la clase `EmpleadoPorComision2`. Esto hace que el código sea más fácil de mantener, modificar y depurar, ya que el código relacionado con un empleado por comisión sólo existe en la clase `EmpleadoPorComision2`.

En este ejemplo, declaramos las variables de instancia de la clase base como `protected`, para que la clase derivada pudiera heredarlas. Al heredar variables de instancia `protected` se incrementa un poco el rendimiento, ya que podemos acceder directamente a las variables en la clase derivada, sin incurrir en la sobrecarga de invocar a los descriptores de acceso `set` o `get` de la propiedad correspondiente. No obstante, en la mayoría de los casos es mejor utilizar variables de instancia `private`, para cumplir con la ingeniería de software apropiada, y dejar las cuestiones relacionadas con la optimización al compilador. Su código será más fácil de mantener, modificar y depurar.

El uso de variables de instancia `protected` crea varios problemas potenciales. En primer lugar, el objeto de la clase derivada puede establecer el valor de una variable heredada directamente sin utilizar el descriptor de acceso `set` de la propiedad. Por lo tanto, un objeto de la clase derivada puede asignar un valor inválido a la variable, con lo cual el objeto queda en un estado inconsistente. Por ejemplo, si declaramos la variable de instancia `ventasBrutas` de `EmpleadoPorComision3` como `protected`, un objeto de una clase derivada (por ejemplo, `EmpleadoBaseMasComision`) podría entonces asignar un valor negativo a `ventasBrutas`.

```

1  // Fig. 10.12: PruebaEmpleadoBaseMasComision3.cs
2  // Prueba de la clase EmpleadoBaseMasComision3.
3  using System;
4
5  public class PruebaEmpleadoBaseMasComision3
6  {
7      public static void Main( string[] args )
8      {
9          // crea instancia de un objeto EmpleadoBaseMasComision3
10         EmpleadoBaseMasComision3 empleadoBaseMasComision =
11             new EmpleadoBaseMasComision3( "Bob", "Lewis",
12                 "333-33-3333", 5000.00M, .04M, 300.00M );
13
14         // muestra en pantalla los datos del empleado por comisión con salario base
15         Console.WriteLine(
16             "Información del empleado obtenida por las propiedades y los métodos: \n" );
17         Console.WriteLine( "{0} {1}", "El primer nombre es",
18             empleadoBaseMasComision.PrimerNombre );
19         Console.WriteLine( "{0} {1}", "El apellido paterno es",
20             empleadoBaseMasComision.ApellidoPaterno );
21         Console.WriteLine( "{0} {1}", "El número de seguro social es",
22             empleadoBaseMasComision.NumeroSeguroSocial );
23         Console.WriteLine( "{0} {1:C}", "Las ventas brutas son",
24             empleadoBaseMasComision.VentasBrutas );
25         Console.WriteLine( "{0} {1:F2}", "La tarifa de comisión es",
26             empleadoBaseMasComision.TarifaComision );
27         Console.WriteLine( "{0} {1:C}", "Los ingresos son",
28             empleadoBaseMasComision.Ingresos() );

```

Figura 10.12 | Prueba de la clase `EmpleadoBaseMasComision3`. (Parte I de 2).


```

29     Console.WriteLine( "{0} {1:C}", "El salario base es",
30         empleadoBaseMasComision.SalarioBase );
31
32     empleadoBaseMasComision.SalarioBase = 1000.00M; // establece el salario base
33
34     Console.WriteLine( "\n{0}:\n\n{1}",
35         "Información actualizada del empleado, obtenida por ToString",
36         empleadoBaseMasComision );
37     Console.WriteLine( "ingresos: {0:C}",
38         empleadoBaseMasComision.Ingresos() );
39 } // fin de Main
40 } // fin de la clase PruebaEmpleadoBaseMasComision3

```

Información del empleado obtenida por las propiedades y los métodos:

El primer nombre es Bob
 El apellido paterno es Lewis
 El número de seguro social es 333-33-3333
 Las ventas brutas son \$5,000.00
 La tarifa de comisión es 0.04
 Los ingresos son \$500.00
 El salario base es \$300.00

Información actualizada del empleado, obtenida por ToString:

empleado por comisión con salario base: Bob Lewis
 número de seguro social: 333-33-3333
 ventas brutas: \$5,000.00
 tarifa de comisión: 0.04
 salario base: \$1,000.00
 ingresos: \$1,200.00

Figura 10.12 | Prueba de la clase `EmpleadoBaseMasComision3`. (Parte 2 de 2).

El segundo problema con el uso de variables de instancia `protected` es que hay más probabilidad de que los métodos de la clase derivada se escriban de manera que dependan de la implementación de datos de la clase base. En la práctica, las clases derivadas sólo deben depender de los servicios de la clase base (es decir, métodos y propiedades que no sean `private`) y no en la implementación de datos de la clase base. Si hay variables de instancia `protected` en la clase base, tal vez necesitemos modificar todas las clases derivadas de la clase base si cambia la implementación de ésta. Por ejemplo, si por alguna razón tuviéramos que cambiar los nombres de las variables de instancia `primerNombre` y `apellidoPaterno` por `nombre` y `apellido`, entonces tendríamos que hacerlo para todas las ocurrencias en las que una clase derivada haga referencia directa a las variables de instancia `primerNombre` y `apellidoPaterno` de la clase base. En tal caso, se dice que el software es *frágil* o *quebradizo*, ya que un pequeño cambio en la clase base puede “quebrar” la implementación de la clase derivada. Lo óptimo es poder modificar la implementación de la clase base sin dejar de proporcionar los mismos servicios a las clases derivadas. Desde luego que, si cambian los servicios de la clase base, debemos reimplementar nuestras clases derivadas.



Observación de ingeniería de software 10.5

Al declarar variables de instancia `private` (a diferencia de `protected`) en la clase base se permite que la implementación de la clase base para estas variables de instancia cambie sin afectar la implementación de la clase derivada.

10.4.5 La jerarquía de herencia `EmpleadoPorComision`–

`EmpleadoBaseMasComision` mediante el uso de variables de instancia `private`

Ahora vamos a reexaminar nuestra jerarquía una vez más, pero ahora utilizaremos las mejores prácticas de ingeniería de software. La clase `EmpleadoPorComision3` (figura 10.13) declara las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `private`

```

1  // Fig. 10.13: EmpleadoPorComision3.cs
2  // La clase EmpleadoPorComision3 representa a un empleado por comisión.
3  public class EmpleadoPorComision3
4  {
5      private string primerNombre;
6      private string apellidoPaterno;
7      private string numeroSeguroSocial;
8      private decimal ventasBrutas; // ventas semanales totales
9      private decimal tarifaComision; // porcentaje de comisión
10
11     // constructor con cinco parámetros
12     public EmpleadoPorComision3( string nombre, string apellido, string nss,
13         decimal ventas, decimal tarifa )
14     {
15         // la llamada implícita al constructor de object se realiza aquí
16         primerNombre = nombre;
17         apellidoPaterno = apellido;
18         numeroSeguroSocial = nss;
19         VentasBrutas = ventas; // valida las ventas brutas a través de una propiedad
20         TarifaComision = tarifa; // valida la tarifa de comisión a través de una propiedad
21     } // fin del constructor de EmpleadoPorComision3 con cinco parámetros
22
23     // propiedad de sólo lectura que obtiene el primer nombre del empleado por comisión
24     public string PrimerNombre
25     {
26         get
27         {
28             return primerNombre;
29         } // fin de get
30     } // fin de la propiedad PrimerNombre
31
32     // propiedad de sólo lectura que obtiene el apellido paterno del empleado por comisión
33     public string ApellidoPaterno
34     {
35         get
36         {
37             return apellidoPaterno;
38         } // fin de get
39     } // fin de la propiedad ApellidoPaterno
40
41     // propiedad de sólo lectura que obtiene el
42     // número de seguro social del empleado por comisión
43     public string NumeroSeguroSocial
44     {
45         get
46         {
47             return numeroSeguroSocial;
48         } // fin de get
49     } // fin de la propiedad NumeroSeguroSocial
50
51     // propiedad que obtiene y establece las ventas brutas del empleado por comisión
52     public decimal VentasBrutas
53     {
54         get
55         {
56             return ventasBrutas;
57         } // fin de get
58         set

```

Figura 10.13 | La clase EmpleadoPorComision3 representa a un empleado por comisión. (Parte 1 de 2).

```

59     {
60         ventasBrutas = ( value < 0 ) ? 0 : value;
61     } // fin de set
62 } // fin de la propiedad VentasBrutas
63
64 // propiedad que obtiene y establece la tarifa de comisión del empleado por comisión
65 public decimal TarifaComision
66 {
67     get
68     {
69         return tarifaComision;
70     } // fin de get
71     set
72     {
73         tarifaComision = ( value > 0 && value < 1 ) ? value : 0;
74     } // fin de set
75 } // fin de la propiedad TarifaComision
76
77 // calcula el salario del empleado por comisión
78 public virtual decimal Ingresos()
79 {
80     return TarifaComision * VentasBrutas;
81 } // fin del método Ingresos
82
83 // devuelve representación string del objeto EmpleadoPorComision
84 public override string ToString()
85 {
86     return string.Format(
87         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}",
88         "empleado por comisión", PrimerNombre, ApellidoPaterno,
89         "número de seguro social", NumeroSeguroSocial,
90         "ventas brutas", VentasBrutas, "tarifa de comisión", TarifaComision );
91 } // fin del método ToString
92 } // fin de la clase EmpleadoPorComision3

```

Figura 10.13 | La clase `EmpleadoPorComision3` representa a un empleado por comisión. (Parte 2 de 2).

(líneas 5-9) y proporciona las propiedades `public` `PrimerNombre`, `ApellidoPaterno`, `NumeroSeguroSocial`, `VentasBrutas` y `TarifaComision` para manipular estos valores. Observe que los métodos `Ingresos` (líneas 78-81) y `ToString` (líneas 84-91) utilizan las propiedades de la clase para obtener los valores de sus variables de instancia. Si decidimos modificar los nombres de las variables de instancia, no habrá que modificar las declaraciones de `Ingresos` y de `ToString`; sólo habrá que modificar los cuerpos de las propiedades que manipulan directamente estas variables de instancia. Observe que estos cambios ocurren sólo dentro de la clase base; no se necesitan cambios en la clase derivada. La localización de los efectos de los cambios como éste es una buena práctica de ingeniería de software. La clase derivada `EmpleadoBaseMasComision4` (figura 10.14) hereda los miembros no `private` de `EmpleadoPorComision3` y puede acceder a los miembros `private` de su clase base a través de sus propiedades `public`.

La clase `EmpleadoBaseMasComision4` (figura 10.14) tiene varios cambios en las implementaciones de sus métodos, que la diferencian de la clase `EmpleadoBaseMasComision3` (figura 10.11). Los métodos `Ingresos` (figura 10.14, líneas 33-36) y `ToString` (líneas 39-43) invocan cada uno al descriptor de acceso `get` de la propiedad `SalarioBase` para obtener el valor del salario base, en vez de acceder en forma directa a `salarioBase`. Si decidimos cambiar el nombre de la variable de instancia `salarioBase`, sólo tendrá que cambiar el cuerpo de la propiedad `SalarioBase`.

El método `Ingresos` de la clase `EmpleadoBaseMasComision4` (figura 10.14, líneas 33-36) redefine al método `Ingresos` de la clase `EmpleadoPorComision3` (figura 10.13, líneas 78-81) para calcular los ingresos de un empleado por comisión con sueldo base. La nueva versión obtiene la porción de los ingresos del empleado,

```

1  // Fig. 10.14: EmpleadoBaseMasComision4.cs
2  // EmpleadoBaseMasComision4 hereda de EmpleadoPorComision3 y tiene
3  // acceso a los datos private de EmpleadoPorComision3 a través de
4  // sus propiedades public.
5  public class EmpleadoBaseMasComision4 : EmpleadoPorComision3
6  {
7      private decimal salarioBase; // salario base por semana
8
9      // constructor de la clase derivada con seis parámetros
10     // con una llamada al constructor de la clase base EmpleadoPorComision3
11     public EmpleadoBaseMasComision4( string nombre, string apellido,
12         string nss, decimal ventas, decimal tarifa, decimal salario )
13         : base( nombre, apellido, nss, ventas, tarifa )
14     {
15         SalarioBase = salario; // valida el salario base a través de una propiedad
16     } // fin del constructor de EmpleadoBaseMasComision4 con seis parámetros
17
18     // propiedad que obtiene y establece el
19     // salario base del empleado por comisión con salario base
20     public decimal SalarioBase
21     {
22         get
23         {
24             return salarioBase;
25         } // fin de get
26         set
27         {
28             salarioBase = ( value < 0 ) ? 0 : value;
29         } // fin de set
30     } // fin de la propiedad SalarioBase
31
32     // calcula los ingresos
33     public override decimal Ingresos()
34     {
35         return SalarioBase + base.Ingresos();
36     } // fin del método Ingresos
37
38     // devuelve representación string de EmpleadoBaseMasComision4
39     public override string ToString()
40     {
41         return string.Format( "{0} {1}\n{2}: {3:C}",
42             "salario base +", base.ToString(), "salario base", SalarioBase );
43     } // fin del método ToString
44 } // fin de la clase EmpleadoBaseMasComision4

```

Figura 10.14 | EmpleadoBaseMasComision4 hereda de EmpleadoPorComision3 y tiene acceso a los datos private de EmpleadoPorComision3 a través de sus propiedades public.

con base en la comisión solamente, mediante una llamada al método Ingresos de EmpleadoPorComision3 con la expresión `base.Ingresos()` (figura 10.14, línea 35). El método Ingresos de EmpleadoBasePorComision4 agrega después el salario base a este valor, para calcular los ingresos totales del empleado. Observe la sintaxis utilizada para invocar un método de clase base redefinido desde una clase derivada: coloque la palabra clave `base` y el operador punto (`.`) antes del nombre del método de la clase base. Esta forma de invocar métodos es una buena práctica de ingeniería de software; al hacer que el método Ingresos de EmpleadoBaseMasComision4 invoque al método Ingresos de EmpleadoPorComision3 para calcular parte de los ingresos del objeto EmpleadoBaseMasComision4, evitamos duplicar el código y se reducen los problemas de mantenimiento del mismo.



Error común de programación 10.3

Cuando se redefine un método de la clase base en una clase derivada, por lo general la versión correspondiente a la clase derivada llama a la versión de la clase base para que realice una parte del trabajo. Si no se antepone al nombre del método de la clase base la palabra clave base y el operador punto (.) cuando se hace referencia al método de la clase base, el método de la clase derivada se llama a sí mismo, creando un error conocido como recursividad infinita. La recursividad, si se utiliza en forma correcta, es una poderosa herramienta, como vimos en la sección 7.13, Recursividad.



Error común de programación 10.4

El uso de referencias base “encadenadas” para hacer referencia a un miembro (un método, propiedad o variable) que esté varios niveles arriba en la jerarquía [como en `base.base.Ingresos()`] es un error de compilación.

De manera similar, el método `ToString` de `EmpleadoBaseMasComision4` (figura 10.14, líneas 39-43) redefine el método `ToString` de la clase `EmpleadoPorComision3` (figura 10.13, líneas 84-91) para devolver una representación string apropiada para un empleado por comisión con salario base. La nueva versión crea parte de la representación string de un objeto `EmpleadoBaseMasComision4` (es decir, la cadena “empleado por comisión con sueldo base” y los valores de las variables de instancia `private` de la clase `EmpleadoPorComision3`), mediante una llamada al método `ToString` de `EmpleadoPorComision3` con la expresión `base.ToString()` (figura 10.14, línea 42). Después, el método `ToString` de `EmpleadoBaseMasComision4` imprime en pantalla el resto de la representación string de un objeto `EmpleadoBaseMasComision4` (es decir, el valor del salario base de la clase `EmpleadoBaseMasComision4`).

La figura 10.15 realiza las mismas manipulaciones sobre un objeto `EmpleadoBaseMasComision4` que las de las figuras 10.7 y 10.12 sobre objetos de las clases `EmpleadoBaseMasComision` y `EmpleadoBaseMasComision3`, respectivamente. Aunque cada clase de “empleado por comisión con salario base” se comporta en forma idéntica, la clase `EmpleadoBaseMasComision4` es la mejor diseñada. Mediante el uso de la herencia y las propiedades que ocultan los datos y aseguran la consistencia, hemos construido una clase bien diseñada con eficiencia y efectividad.

```

1  // Fig. 10.15: PruebaEmpleadoBaseMasComision4.cs
2  // Prueba de la clase EmpleadoBaseMasComision4.
3  using System;
4
5  public class PruebaEmpleadoBaseMasComision4
6  {
7      public static void Main( string[] args )
8      {
9          // crea instancia de un objeto EmpleadoBaseMasComision4
10         EmpleadoBaseMasComision4 empleado =
11             new EmpleadoBaseMasComision4( "Bob", "Lewis",
12                 "333-33-3333", 5000.00M, .04M, 300.00M );
13
14         // muestra los datos del empleado por comisión con salario base
15         Console.WriteLine(
16             "Información del empleado obtenida por las propiedades y los métodos: \n" );
17         Console.WriteLine( "{0} {1}", "El primer nombre es",
18             empleado.PrimerNombre );
19         Console.WriteLine( "{0} {1}", "El apellido paterno es",
20             empleado.ApellidoPaterno );
21         Console.WriteLine( "{0} {1}", "El número de seguro social es",
22             empleado.NumeroSeguroSocial );
23         Console.WriteLine( "{0} {1:C}", "Las ventas brutas son",
24             empleado.VentasBrutas );
25         Console.WriteLine( "{0} {1:F2}", "La tarifa de comisión es",
26             empleado.TarifaComision );
27         Console.WriteLine( "{0} {1:C}", "Los ingresos son",

```

Figura 10.15 | Prueba de la clase `EmpleadoBaseMasPorComision4`. (Parte I de 2).

```

28     empleado.Ingresos() );
29     Console.WriteLine( "{0} {1:C}", "El salario base es",
30         empleado.SalarioBase );
31
32     empleado.SalarioBase = 1000.00M; // establece el salario base
33
34     Console.WriteLine( "\n{0}:\n\n{1}",
35         "Información actualizada del empleado, obtenida por ToString", empleado );
36     Console.WriteLine( "ingresos: {0:C}", empleado.Ingresos() );
37 } // fin de Main
38 } // fin de la clase PruebaEmpleadoBaseMasComision4

```

Información del empleado obtenida por las propiedades y los métodos:

El primer nombre es Bob
 El apellido paterno es Lewis
 El número de seguro social es 333-33-3333
 Las ventas brutas son \$5,000.00
 La tarifa de comisión es 0.04
 Los ingresos son \$500.00
 El salario base es \$300.00

Información actualizada del empleado, obtenida por ToString:

salario base + empleado por comisión: Bob Lewis
 número de seguro social: 333-33-3333
 ventas brutas: \$5,000.00
 tarifa de comisión: 0.04
 salario base: \$1,000.00
 ingresos: \$1,200.00

Figura 10.15 | Prueba de la clase `EmpleadoBaseMasPorComision4`. (Parte 2 de 2).

En esta sección vio la evolución de un conjunto de ejemplos diseñados cuidadosamente para enseñar las capacidades clave de la buena ingeniería de software mediante el uso de la herencia. Aprendió a crear una clase derivada mediante la herencia, a utilizar miembros `protected` de la clase base para permitir que una clase derivada acceda a las variables de instancia heredadas de una clase base y cómo redefinir los métodos de la clase base para proporcionar versiones más apropiadas para los objetos de las clases derivadas. Además, aplicó las técnicas de ingeniería de software de los capítulos 4, 9 y este capítulo para crear clases que sean fáciles de mantener, modificar y depurar.

10.5 Los constructores en las clases derivadas

Como explicamos en la sección anterior, al crear una instancia de un objeto de clase derivada se empieza una cadena de llamadas a los constructores, en los que el constructor de la clase derivada, antes de realizar sus propias tareas, invoca al constructor de su clase base directa, ya sea en forma explícita (por medio de un inicializador de constructor con la referencia base) o implícita (llamando al constructor predeterminado o sin parámetros de la clase base). De manera similar, si la clase base se deriva de otra clase (como sucede con cualquier clase, excepto `object`), el constructor de la clase base invoca al constructor de la siguiente clase que se encuentre a un nivel más arriba en la jerarquía, y así en lo sucesivo. El último constructor que se llama en la cadena es siempre el de la clase `object`. El cuerpo del constructor original de la clase derivada termina de ejecutarse al último. El constructor de cada clase base manipula las variables de instancia de la clase base que hereda el objeto de la clase derivada. Por ejemplo, considere de nuevo la jerarquía `EmpleadoPorComision3`–`EmpleadoBaseMasComision4` de las figuras 10.13 y 10.14. Cuando una aplicación crea un objeto `EmpleadoBaseMasComision4`, se hace una llamada al constructor de `EmpleadoBaseMasComision4`. Ese constructor llama al constructor de la clase `EmpleadoPor-`

Comision3, que a su vez llama en forma implícita al constructor de `object`. El constructor de la clase `object` tiene un cuerpo vacío, por lo que devuelve de inmediato el control al constructor de `EmpleadoPorComision3`, el cual inicializa las variables de instancia `private` de `EmpleadoPorComision3` que son parte del objeto `EmpleadoBaseMasComision4`. Cuando este constructor termina de ejecutarse, devuelve el control al constructor de `EmpleadoBaseMasComision4`, el cual inicializa el `salarioBase` del objeto `EmpleadoBaseMasComision4`.



Observación de ingeniería de software 10.6

Cuando una aplicación crea un objeto de una clase derivada, el constructor de la clase derivada llama de inmediato al constructor de la clase base (ya sea en forma explícita, mediante `base`, o implícita). El cuerpo del constructor de la clase base se ejecuta para inicializar las variables de instancia de la clase base que forman parte del objeto de la clase derivada, después se ejecuta el cuerpo del constructor de la clase derivada para inicializar las variables de instancia que son parte sólo de la clase derivada. Aun si un constructor no asigna un valor a una variable de instancia, la variable de todas formas se inicializa con su valor predeterminado (es decir, 0 para los tipos numéricos simples, `false` para `bool` y `null` para las referencias).

En nuestro siguiente ejemplo volvemos a utilizar la jerarquía de empleado por comisión, al declarar las clases `EmpleadoPorComision4` (figura 10.16) y `EmpleadoBaseMasComision5` (figura 10.17). El constructor de cada clase imprime un mensaje cuando se le invoca, lo cual nos permite observar el orden en el que se ejecutan los constructores en la jerarquía.

La clase `EmpleadoPorComision4` (figura 10.16) contiene las mismas características que la versión de la clase que se muestra en la figura 10.13. Se modificó el constructor (líneas 14-25) para desplegar el texto cuando es llamado. Observe que si concatenamos la palabra clave `this` con una literal `string` (línea 24) se invoca en forma implícita al método `ToString` del objeto que se está creando, para obtener la representación `string` de ese objeto.

```

1  // Fig. 10.16: EmpleadoPorComision4.cs
2  // La clase EmpleadoPorComision4 representa a un empleado por comisión.
3  using System;
4
5  public class EmpleadoPorComision4
6  {
7      private string primerNombre;
8      private string apellidoPaterno;
9      private string numeroSeguroSocial;
10     private decimal ventasBrutas; // ventas semanales totales
11     private decimal tarifaComision; // porciento de comisión
12
13     // constructor con cinco parámetros
14     public EmpleadoPorComision4( string nombre, string apellido, string nss,
15         decimal ventas, decimal tarifa )
16     {
17         // la llamada implícita al constructor del objeto se realiza aquí
18         primerNombre = nombre;
19         apellidoPaterno = apellido;
20         numeroSeguroSocial = nss;
21         VentasBrutas = ventas; // valida las ventas brutas a través de una propiedad
22         TarifaComision = tarifa; // valida la tarifa de comisión a través de una propiedad
23
24         Console.WriteLine( "\nConstructor de EmpleadoPorComision4:\n" + this );
25     } // fin del constructor de EmpleadoPorComision4 con cinco parámetros
26
27     // propiedad de sólo lectura que obtiene el primer nombre del empleado por comisión
28     public string PrimerNombre

```

Figura 10.16 | La clase `EmpleadoPorComision4` representa a un empleado por comisión. (Parte I de 3).


```

29     {
30         get
31         {
32             return primerNombre;
33         } // fin de get
34     } // fin de la propiedad PrimerNombre
35
36     // propiedad de sólo lectura que obtiene el apellido paterno del empleado por comisión
37     public string ApellidoPaterno
38     {
39         get
40         {
41             return apellidoPaterno;
42         } // fin de get
43     } // fin de la propiedad ApellidoPaterno
44
45     // propiedad de sólo lectura que obtiene el
46     // número de seguro social del empleado por comisión
47     public string NumeroSeguroSocial
48     {
49         get
50         {
51             return numeroSeguroSocial;
52         } // fin de get
53     } // fin de la propiedad NumeroSeguroSocial
54
55     // propiedad que obtiene y establece las ventas brutas del empleado por comisión
56     public decimal VentasBrutas
57     {
58         get
59         {
60             return ventasBrutas;
61         } // fin de get
62         set
63         {
64             ventasBrutas = ( value < 0 ) ? 0 : value;
65         } // fin de set
66     } // fin de la propiedad VentasBrutas
67
68     // propiedad que obtiene y establece la tarifa de comisión de un empleado por
69     // comisión
70     public decimal TarifaComision
71     {
72         get
73         {
74             return tarifaComision;
75         } // fin de get
76         set
77         {
78             tarifaComision = ( value > 0 && value < 1 ) ? value : 0;
79         } // fin de set
80     } // fin de la propiedad TarifaComision
81
82     // calcula el salario del empleado por comisión
83     public virtual decimal Ingresos()
84     {
85         return TarifaComision * VentasBrutas;
86     } // fin del método Ingresos

```

Figura 10.16 | La clase `EmpleadoPorComision4` representa a un empleado por comisión. (Parte 2 de 3).


```

86
87 // devuelve representación string del objeto EmpleadoPorComision
88 public override string ToString()
89 {
90     return string.Format(
91         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}",
92         "empleado por comisión", PrimerNombre, ApellidoPaterno,
93         "número de seguro social", NumeroSeguroSocial,
94         "ventas brutas", VentasBrutas, "tarifa de comisión", TarifaComision );
95     } // fin del método ToString
96 } // fin de la clase EmpleadoPorComision4

```

Figura 10.16 | La clase `EmpleadoPorComision4` representa a un empleado por comisión. (Parte 3 de 3).

La clase `EmpleadoBaseMasComision5` (figura 10.17) es casi idéntica a `EmpleadoBaseMasComision4` (figura 10.14), sólo que el constructor de `EmpleadoBaseMasComision5` imprime texto cuando se invoca. Al igual que en `EmpleadoPorComision4` (figura 10.16), concatenamos la palabra clave `this` con una literal `string` para obtener de manera implícita la representación `string` del objeto.

```

1 // Fig. 10.17: EmpleadoBaseMasComision5.cs
2 // Declaración de la clase EmpleadoBaseMasComision5.
3 using System;
4
5 public class EmpleadoBaseMasComision5 : EmpleadoPorComision4
6 {
7     private decimal salarioBase; // salario base por semana
8
9     // constructor de la clase derivada con seis parámetros
10    // con una llamada al constructor de la clase base EmpleadoPorComision4
11    public EmpleadoBaseMasComision5( string nombre, string apellido,
12        string nss, decimal ventas, decimal tarifa, decimal salario )
13        : base( nombre, apellido, nss, ventas, tarifa )
14    {
15        SalarioBase = salario; // valida el salario base a través de una propiedad
16
17        Console.WriteLine(
18            "\nConstructor de EmpleadoBaseMasComision5:\n" + this );
19    } // fin del constructor de EmpleadoBaseMasComision5 con seis parámetros
20
21    // propiedad que obtiene y establece el
22    // salario base del empleado por comisión con salario base
23    public decimal SalarioBase
24    {
25        get
26        {
27            return salarioBase;
28        } // fin de get
29        set
30        {
31            salarioBase = ( value < 0 ) ? 0 : value;
32        } // fin de set
33    } // fin de la propiedad SalarioBase
34
35    // calcula los ingresos

```

Figura 10.17 | Declaración de la clase `EmpleadoBaseMasComision5`. (Parte 1 de 2).

```

36  public override decimal Ingresos()
37  {
38      return SalarioBase + base.Ingresos();
39  } // fin del método Ingresos
40
41  // devuelve representación string de EmpleadoBaseMasComision5
42  public override string ToString()
43  {
44      return string.Format( "{0} {1}\n{2}: {3:C}",
45          "salario base +", base.ToString(), "salario base", SalarioBase );
46  } // fin del método ToString
47  } // fin de la clase EmpleadoBaseMasComision5

```

Figura 10.17 | Declaración de la clase `EmpleadoBaseMasComision5`. (Parte 2 de 2).

La figura 10.18 demuestra el orden en el que se llaman los constructores para los objetos de las clases que forman parte de una jerarquía de herencia. El método `Main` empieza por crear una instancia del objeto `empleado1` de la clase `EmpleadoPorComision4` (líneas 10-11). A continuación, las líneas 14-16 crean una instancia del objeto `empleado2` de `EmpleadoBaseMasComision5`. Esto invoca al constructor de `EmpleadoPorComision4`, el cual imprime los resultados con los valores que recibe del constructor de `EmpleadoBaseMasComision5` y después imprime los resultados especificados en el constructor de `EmpleadoBaseMasComision5`. Después, las líneas 19-21 crean una instancia del objeto `empleado3` de `EmpleadoBaseMasComision5`. De nuevo, se hacen llamadas a los constructores de `EmpleadoPorComision4` y `EmpleadoBaseMasComision5`. En cada caso, el cuerpo del constructor de `EmpleadoPorComision4` se ejecuta antes que el cuerpo del constructor de `EmpleadoBaseMasComision5`. Observe que `empleado2` se crea por completo antes que empiece el constructor de `empleado3`.

```

1  // Fig. 10.18: PruebaConstructor.cs
2  // Muestra el orden en el que se llaman los constructores de
3  // la clase base y la clase derivada.
4  using System;
5
6  public class PruebaConstructor
7  {
8      public static void Main( string[] args )
9      {
10         EmpleadoPorComision4 empleado1 = new EmpleadoPorComision4( "Bob",
11             "Lewis", "333-33-3333", 5000.00M, .04M );
12
13         Console.WriteLine();
14         EmpleadoBaseMasComision5 empleado2 =
15             new EmpleadoBaseMasComision5( "Lisa", "Jones",
16                 "555-55-5555", 2000.00M, .06M, 800.00M );
17
18         Console.WriteLine();
19         EmpleadoBaseMasComision5 empleado3 =
20             new EmpleadoBaseMasComision5( "Mark", "Sands",
21                 "888-88-8888", 8000.00M, .15M, 2000.00M );
22     } // fin de Main
23 } // fin de la clase PruebaConstructor

```

Figura 10.18 | Orden de visualización en el que se llaman los constructores de la clase base y la clase derivada. (Parte 1 de 2).

```

Constructor de EmpleadoPorComision4:
empleado por comisión: Bob Lewis
número de seguro social: 333-33-3333
ventas brutas: $5,000.00
tarifa de comisión: 0.04

Constructor de EmpleadoPorComision4:
salario base + empleado por comisión: Lisa Jones
número de seguro social: 555-55-5555
ventas brutas: $2,000.00
tarifa de comisión: 0.06
salario base: $0.00

Constructor de EmpleadoBaseMasComision5:
salario base + empleado por comisión: Lisa Jones
número de seguro social: 555-55-5555
ventas brutas: $2,000.00
tarifa de comisión: 0.06
salario base: $800.00

Constructor de EmpleadoPorComision4:
salario base + empleado por comisión: Mark Sands
número de seguro social: 888-88-8888
ventas brutas: $8,000.00
tarifa de comisión: 0.15
salario base: $0.00

Constructor de EmpleadoBaseMasComision5:
salario base + empleado por comisión: Mark Sands
número de seguro social: 888-88-8888
ventas brutas: $8,000.00
tarifa de comisión: 0.15
salario base: $2,000.00

```

Figura 10.18 | Orden de visualización en el que se llaman los constructores de la clase base y la clase derivada. (Parte 2 de 2).

10.6 Ingeniería de software mediante la herencia

En esta sección hablaremos sobre la personalización del software existente mediante la herencia. Cuando una nueva clase extiende a una clase existente, la nueva clase hereda los miembros de la clase existente. Podemos personalizar la nueva clase para cumplir nuestras necesidades, mediante la inclusión de miembros adicionales y la redefinición de miembros de la clase base. Para hacer esto, el programador de la clase derivada no tiene que modificar el código fuente de la clase base. C# sólo requiere el acceso al código compilado de la clase base, para poder compilar y ejecutar cualquier aplicación que utilice o extienda la clase base. Esta poderosa capacidad es atractiva para los distribuidores independientes de software (ISVs), quienes pueden desarrollar clases propietarias para vender o licenciar, y ponerlas a disposición de los usuarios en bibliotecas de clases. Después, los usuarios pueden derivar con rapidez nuevas clases a partir de estas clases de biblioteca, sin necesidad de acceder al código fuente propietario del ISV.



Observación de ingeniería de software 10.7

A pesar del hecho de que al heredar de una clase no se requiere acceso a su código fuente, los desarrolladores insisten con frecuencia en ver el código fuente para comprender cómo está implementada la clase. Por ejemplo, tal vez quieran asegurarse de que están extendiendo una clase que se desempeña bien y que se implementa en forma segura.

Algunas veces, los estudiantes tienen dificultad para apreciar el alcance de los problemas a los que se enfrentan los diseñadores que trabajan en proyectos de software a gran escala en la industria. Las personas experimentadas

con esos proyectos dicen que la reutilización efectiva del software mejora el proceso de desarrollo del mismo. La programación orientada a objetos facilita la reutilización de software, con lo que se obtiene una potencial reducción en el tiempo de desarrollo. La disponibilidad de bibliotecas de clases extensas y útiles produce los máximos beneficios de la reutilización de software a través de la herencia. Las bibliotecas de clases de la FCL que utiliza C# tienden a ser más de propósito general. Existen muchas bibliotecas de clases de propósito especial y muchas más están en proceso de crearse.



Observación de ingeniería de software 10.8

En la etapa de diseño de un sistema orientado a objetos, el diseñador encuentra comúnmente que ciertas clases están muy relacionadas. Es conveniente que el diseñador “factorice” los miembros comunes y los coloque en una clase base. Después debe usar la herencia para desarrollar clases derivadas, especializándolas con herramientas que estén más allá de las heredadas de parte de la clase base.



Observación de ingeniería de software 10.9

Declarar una clase derivada no afecta el código fuente de la clase base. La herencia preserva la integridad de la clase base.



Observación de ingeniería de software 10.10

Así como los diseñadores de sistemas no orientados a objetos deben evitar la proliferación de métodos, los diseñadores de sistemas orientados a objetos deben evitar la proliferación de clases. Dicha proliferación crea problemas administrativos y puede obstaculizar la reutilización de software, ya que en una biblioteca de clases enorme es difícil para un cliente localizar las clases más apropiadas. La alternativa es crear menos clases que proporcionen una funcionalidad más sustancial, pero dichas clases podrían volverse incómodas.



Tip de rendimiento 10.1

Si las clases derivadas son más grandes de lo necesario (es decir, que contengan demasiada funcionalidad), podrían desperdiciarse los recursos de memoria y de procesamiento. Extienda la clase base que contenga la funcionalidad que esté más cerca de lo que necesita.

Puede ser confuso leer las declaraciones de las clases derivadas, ya que los miembros heredados no se declaran de manera explícita en las clases derivadas, pero sin embargo están presentes en ellas. Hay un problema similar a la hora de documentar los miembros de las clases derivadas.

10.7 La clase object

Como vimos al principio en este capítulo, todas las clases heredan ya sea en forma directa o indirecta de la clase object (System.Object en la FCL), por lo que todas las demás clases heredan sus siete métodos. La figura 10.19 muestra un resumen de los métodos de object.

A lo largo de este libro hablaremos sobre varios métodos de object (como se indica en la tabla). También puede aprender más acerca de los métodos de object en la documentación en línea de object en la Referencia de la Biblioteca de clases del .NET Framework, en:

msdn2.microsoft.com/en-us/library/system.object_members (inglés)

msdn.microsoft.com/library/spa/default.asp?url=/library/SPA/cpref/html/fr1rfSystem-ObjectClassTopic.asp (español)

Todos los tipos de arreglos heredan en forma implícita de la clase Array en el espacio de nombres System, que a su vez extiende a la clase object. Como resultado y al igual que otros objetos, un arreglo hereda los miembros de la clase object. Para obtener más información acerca de la clase Array, consulte la documentación para esta clase en la Referencia de la FCL:

msdn2.microsoft.com/en-us/library/system.array_members (inglés)

msdn.microsoft.com/library/spa/default.asp?url=/library/SPA/cpref/html/fr1rfsystem-arrayclasstopic.asp (español)

Método	Descripción
Equals	Este método compara la igualdad entre dos objetos; devuelve true si son iguales y false en caso contrario. El método recibe cualquier objeto como argumento. Cuando debe compararse la igualdad entre objetos de una clase en particular, la clase debe redefinir el método Equals para comparar el contenido de los dos objetos. La implementación de este método debe cumplir los siguientes requerimientos: • Debe devolver false si el argumento es null. • Debe devolver true si un objeto se compara consigo mismo, como en <code>objeto1.Equals(objeto1)</code>. • Debe devolver true sólo si tanto <code>objeto1.Equals(objeto2)</code> como <code>objeto2.Equals(objeto1)</code> devuelven true. • Para tres objetos, si <code>objeto1.Equals(objeto2)</code> devuelve true y <code>objeto3.Equals(objeto3)</code> devuelve true, entonces <code>objeto.Equals(objeto)</code> también debe devolver true. • Una clase que redefina el método Equals también debe redefinir el método GetHashCode para asegurar que los objetos iguales tengan códigos hash idénticos. La implementación predeterminada de Equals sólo determina si dos referencias <i>hacen referencia al mismo objeto</i> en memoria.
Finalize	Este método no puede declararse o llamarse en forma explícita. Cuando una clase contiene un destructor, el compilador lo renombra en forma implícita para que redefina el método <code>protected Finalize</code>, que sólo el recolector de basura puede llamar antes de reclamar la memoria de un objeto. Como no se garantiza que el recolector de basura reclame a un objeto, tampoco se garantiza que se vaya a ejecutar el método Finalize de un objeto. Cuando se ejecuta el método Finalize de una clase derivada, éste realiza su tarea y después invoca al método Finalize de la clase base. La implementación predeterminada de Finalize es un receptáculo que tan sólo invoca el método Finalize de la clase base.
GetHashCode	Una tabla hash es una estructura de datos que relaciona un objeto (conocido como la clave) con otro objeto (conocido como el valor). En el capítulo 26, Colecciones, hablaremos sobre HashTable. Cuando se inserta al principio un valor en una tabla hash, se hace una llamada al método GetHashCode de la clave. La tabla hash utiliza el valor de código hash devuelto para determinar la ubicación en la que se debe insertar el valor correspondiente. La tabla hash también utiliza el código hash de la clave para localizar el valor correspondiente de la clave.
GetType	Cada objeto conoce su propio tipo en tiempo de ejecución. El método GetType (que utilizaremos en la sección 11.5) devuelve un objeto de la clase Type (espacio de nombres System) que contiene información acerca del tipo del objeto, como su nombre de clase (que se obtiene de la propiedad FullName de Type).
Memberwise-Clone	Este método protected, que no recibe argumentos y devuelve una referencia object, crea una copia del objeto desde el cual se llamó. La implementación de este método realiza una <i>copia superficial</i>: los valores de las variables de instancia en un objeto se copian a otro objeto del mismo tipo. Para los tipos de referencia, sólo se copian las referencias.
Reference-Equals	Este método static recibe dos argumentos object y devuelve true si dos objetos son la misma instancia, o si son referencias null. En cualquier otro caso, devuelve false.
ToString	Este método (presentado en la sección 7.4) devuelve una representación string de un objeto. La implementación predeterminada de este método devuelve el espacio de nombres, seguido de un punto y del nombre de la clase del objeto.

Figura 10.19 | Los métodos de object que todas las clases heredan en forma directa o indirecta.

10.8 Conclusión

En este capítulo se introdujo el concepto de la herencia: la habilidad de crear clases mediante la absorción de los miembros de una clase existente, mejorándolos con nuevas capacidades. Aprendió las nociones de las clases base y las clases derivadas, y creó una clase derivada que hereda miembros de una clase base. En este capítulo se introdujo también el modificador de acceso `protected`; los métodos de clases derivadas pueden acceder a los miembros `protected` de la clase base. Aprendió también cómo acceder a los miembros de la clase base mediante `base`. Vio además cómo se utilizan los constructores en las jerarquías de herencia. Por último, aprendió acerca de los métodos de la clase `object`, la clase base directa o indirecta de todas las clases.

En el capítulo 11, Polimorfismo, interfaces y sobrecarga de operadores, continuaremos con nuestra discusión sobre la herencia al introducir el polimorfismo: un concepto orientado a objetos que nos permite escribir aplicaciones que puedan manipular, de una forma más general, objetos de una amplia variedad de clases relacionadas por la herencia. Después de estudiar el capítulo 11, estará familiarizado con las clases, los objetos, el encapsulamiento, la herencia y el polimorfismo: los aspectos más esenciales de la programación orientada a objetos.

II

Polimorfismo, interfaces y sobrecarga de operadores

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- El concepto de polimorfismo y cómo le permite “programar en general”.
- Cómo utilizar métodos redefinidos para efectuar el polimorfismo.
- Diferenciar entre las clases concretas y abstractas.
- Declarar métodos abstractos para crear clases abstractas.
- Cómo el polimorfismo hace que los sistemas sean extensibles y administrables.
- Determinar el tipo de un objeto en tiempo de ejecución.
- Crear métodos y clases sealed.
- Declarar e implementar interfaces.
- Sobrecargar operadores para permitirles manipular objetos.

*Un anillo para gobernarlos
a todos, un anillo para
encontrarlos, un anillo
para traerlos a todos y en la
oscuridad enlazarlos.*

—John Ronald Reuel Tolkien

*Las proposiciones generales
no deciden casos concretos.*

—Oliver Wendell Holmes

*Un filósofo de imponente
estatura no piensa en un
vacío. Incluso, sus ideas más
abstractas son, en cierta
medida, condicionadas por
lo que se conoce o no en el
tiempo en que vive.*

—Alfred North Whitehead

*¿Por qué, alma mía,
desfalleces y te agitas por mí?*

—Salmos 42:5

- 11.1 Introducción
- 11.2 Ejemplos de polimorfismo
- 11.3 Demostración del comportamiento polimórfico
- 11.4 Clases y métodos abstractos
- 11.5 Caso de estudio: sistema de nómina utilizando polimorfismo
 - 11.5.1 Creación de la clase base abstracta `Empleado`
 - 11.5.2 Creación de la clase derivada concreta `EmpleadoAsalariado`
 - 11.5.3 Creación de la clase derivada concreta `EmpleadoPorHoras`
 - 11.5.4 Creación de la clase derivada concreta `EmpleadoPorComision`
 - 11.5.5 Creación de la clase derivada concreta indirecta `EmpleadoBaseMasComision`
 - 11.5.6 El procesamiento polimórfico, el operador `is` y la conversión descendente
 - 11.5.7 Resumen de las asignaciones permitidas entre variables de la clase base y de la clase derivada
- 11.6 Métodos y clases `sealed`
- 11.7 Caso de estudio: creación y uso de interfaces
 - 11.7.1 Desarrollo de una jerarquía `IPorPagar`
 - 11.7.2 Declaración de la interfaz `IPorPagar`
 - 11.7.3 Creación de la clase `Factura`
 - 11.7.4 Modificación de la clase `Empleado` para implementar la interfaz `IPorPagar`
 - 11.7.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `IPorPagar`
 - 11.7.6 Uso de la interfaz `IPorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo
 - 11.7.7 Interfaces comunes de la Biblioteca de clases del .NET Framework
- 11.8 Sobrecarga de operadores
- 11.9 (Opcional) Caso de estudio de ingeniería de software: incorporación de la herencia y el polimorfismo en el sistema ATM
- 11.10 Conclusión

11.1 Introducción

Ahora continuaremos nuestro estudio de la programación orientada a objetos, explicando y demostrando el *polimorfismo* con las jerarquías de herencia. El polimorfismo nos permite “programar en forma general”, en vez de “programar en forma específica”. En especial, nos permite escribir aplicaciones que procesen objetos de clases que formen parte de la misma jerarquía de clases, como si todos fueran objetos de la clase.

Considere el siguiente ejemplo de polimorfismo. Suponga que crearemos una aplicación que simula el movimiento de varios tipos de animales para un estudio biológico. Las clases `Pez`, `Rana` y `Ave` representan los tres tipos de animales bajo investigación. Imagine que cada una de estas clases extiende a la clase base `Animal`, que contiene un método llamado `Mover` y mantiene la posición actual de un animal, en forma de coordenadas x - y . Cada clase derivada implementa el método `Mover`. Nuestra aplicación mantiene un arreglo de referencias a objetos de las diversas clases derivadas de `Animal`. Para simular los movimientos de los animales, la aplicación envía a cada objeto el mismo mensaje una vez por segundo; a saber, `Mover`. No obstante, cada tipo específico de `Animal` responde a un mensaje `Mover` de manera única; un `Pez` podría nadar tres pies, una `Rana` podría saltar cinco pies y un `Ave` podría volar 10 pies. La aplicación envía el mismo mensaje (es decir, `Mover`) a cada objeto animal en forma genérica, pero cada objeto sabe cómo modificar sus coordenadas x - y en forma apropiada para su tipo específico de movimiento. Confiar en que cada objeto sepa cómo “hacer lo correcto” (es decir, lo que sea apropiado para ese tipo de objeto) en respuesta a la llamada al mismo método es el concepto clave del polimorfismo. El mismo mensaje (en este caso, `Mover`) que se envía a una variedad de objetos tiene “muchas formas” de resultados; de aquí que se utilice el término polimorfismo.

Con el polimorfismo podemos diseñar e implementar sistemas que puedan extenderse con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales de la aplicación, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que la aplicación procesa en forma genérica. Las únicas partes de una aplicación que deben alterarse para alojar nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador va a agregar a la jerarquía. Por ejemplo, si extendemos la clase `Animal` para crear la clase `Tortuga` (que podría responder a un mensaje `Mover` caminando una pulgada), necesitamos escribir sólo la clase `Tortuga` y la parte de la simulación que crea una instancia de un objeto `Tortuga`. Las porciones de la simulación que procesan a cada `Animal` en forma genérica pueden permanecer iguales.

Este capítulo se divide en varias partes. Primero hablaremos sobre los ejemplos comunes del polimorfismo. Después proporcionaremos un ejemplo de código activo, en el que se demuestra el comportamiento polimórfico. Como pronto verá, utilizará referencias a la clase base para manipular tanto los objetos de la clase base como los de las clases derivadas mediante el polimorfismo.

Después presentaremos un caso de estudio en el que volveremos a utilizar la jerarquía de empleados de la sección 10.4.5. Desarrollaremos una aplicación simple de nómina que, mediante el polimorfismo, calcula el salario semanal de varios tipos de empleados, usando el método `Ingresos` de cada empleado. Aunque los ingresos de cada tipo de empleado se calculan de una manera específica, el polimorfismo nos permite procesar a los empleados “en general”. En el caso de estudio ampliaremos la jerarquía para incluir dos nuevas clases: `EmpleadoAsalariado` (para las personas que reciben un salario semanal fijo) y `EmpleadoPorHoras` (para las personas que reciben un salario por horas y “tiempo y medio” por el tiempo extra). Declararemos un conjunto común de funcionalidad para todas las clases en la jerarquía actualizada en una clase “abstracta” llamada `Empleado`, a partir de la cual las clases `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` heredan en forma directa, y la clase `EmpleadoBaseMasComision4` hereda en forma indirecta. Como veremos a continuación, cuando invocamos el método `Ingresos` de cada empleado desde una referencia a la clase base `Empleado`, se realiza el cálculo correcto de los ingresos debido a las capacidades polimórficas de C#.

Algunas veces, cuando se lleva a cabo el procesamiento del polimorfismo, es necesario programar “en forma específica”. Nuestro caso de estudio con `Empleado` demuestra que una aplicación puede determinar el tipo de un objeto en tiempo de ejecución, y actuar sobre ese objeto de manera acorde. En el caso de estudio utilizamos estas capacidades para determinar si cierto objeto empleado *es un* `EmpleadoBaseMasComision`. Si es así, incrementamos el salario base de ese empleado en 10 por ciento.

El capítulo continúa con una introducción a las interfaces en C#. Una interfaz describe a un conjunto de métodos y propiedades que pueden llamarse en un objeto, pero no proporciona implementaciones concretas para ellos. Los programadores pueden declarar clases que *implementen* a (es decir, que proporcionen implementaciones concretas para los métodos y propiedades de) una o más interfaces. Cada miembro de interfaz debe declararse en todas las clases que implementen a la interfaz. Una vez que una clase implementa a una interfaz, todos los objetos de esa clase tienen una relación *es un* con el tipo de la interfaz, y se garantiza que todos los objetos de la clase proporcionarán la funcionalidad descrita por la interfaz. Esto se aplica también para todas las clases derivadas de esa clase.

En especial, las interfaces son útiles para asignar la funcionalidad común a clases que posiblemente no estén relacionadas. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de las clases que implementan la misma interfaz pueden responder a las mismas llamadas a los métodos. Para demostrar la creación y el uso de interfaces, modificaremos nuestra aplicación de nómina para crear una aplicación general de cuentas por pagar, que puede calcular los pagos vencidos por los ingresos de los empleados de la compañía y los montos de las facturas a pagar por los bienes comprados. Como verá, las interfaces permiten capacidades polimórficas similares a las que permite la herencia.

Este capítulo termina con una introducción a la sobrecarga de operadores. En capítulos anteriores, declaramos nuestras propias clases y utilizamos métodos para realizar tareas con objetos de esas clases. La sobrecarga de operadores nos permite definir el comportamiento de los operadores integrados, como `+`, `-` y `<`, cuando los utilizamos con objetos de nuestras propias clases. Esto proporciona una notación mucho más conveniente que las llamadas a métodos para realizar tareas con los objetos.

11.2 Ejemplos de polimorfismo

Ahora consideraremos diversos ejemplos adicionales de polimorfismo. Si la clase `Rectangulo` se deriva de la clase `Cuadrilatero` (una figura con cuatro lados), entonces un `Rectangulo` es una versión más específica de

Cuadrilátero. Cualquier operación (por ejemplo, calcular el perímetro o el área) que pueda realizarse en un objeto Cuadrilátero también puede realizarse en un objeto Rectángulo. Estas operaciones también pueden realizarse en otros objetos Cuadrilátero, como Cuadrado, Paralelogramo y Trapecioide. El polimorfismo ocurre cuando una aplicación invoca a un método a través de una variable de la clase base; en tiempo de ejecución, se hace una llamada a la versión correcta del método de la clase derivada, con base en el tipo del objeto referenciado. En la sección 11.3 veremos un ejemplo de código simple, en el cual se ilustra este proceso.

Como otro ejemplo, suponga que vamos a diseñar un videojuego que manipule objetos de muchos tipos distintos, incluyendo objetos de las clases Marciano, Venusino, Plutonio, NaveEspacial y RayoLaser. Imagine que cada clase hereda de la clase base común ObjetoEspacial, el cual contiene el método Dibujar. Cada clase derivada implementa a este método. Una aplicación de administración de la pantalla mantiene una colección (por ejemplo, un arreglo ObjetoEspacial) de referencias a objetos de las diversas clases. Para refrescar la pantalla, el administrador de pantalla envía en forma periódica el mismo mensaje a cada objeto; a saber, Dibujar. No obstante, cada objeto responde de una manera única. Por ejemplo, un objeto Marciano podría dibujarse a sí mismo en color rojo, con el número apropiado de antenas. Un objeto NaveEspacial podría dibujarse a sí mismo como un platillo volador de color plata brillante. Un objeto RayoLaser podría dibujarse a sí mismo como un rayo color rojo brillante a lo largo de la pantalla. De nuevo, el mismo mensaje (en este caso, Dibujar) que se envía a una variedad de objetos tiene “muchas formas” de resultados.

Un administrador de pantalla polimórfico podría utilizar el polimorfismo para facilitar el proceso de agregar nuevas clases a un sistema, con el menor número de modificaciones al código del sistema. Suponga que deseamos agregar objetos Mercuriano a nuestro videojuego. Para ello, debemos crear una clase Mercuriano que extienda a ObjetoEspacial y proporcione su propia implementación del método Dibujar. Cuando aparezcan objetos de la clase Mercuriano en la colección ObjetoEspacial, el código del administrador de pantalla invocará al método Dibujar, de la misma forma que para cualquier otro objeto en la colección, sin importar su tipo. Por lo tanto, los nuevos objetos Mercuriano simplemente se integran al videojuego sin necesidad de que el programador modifique el código del administrador de pantalla. Así, sin modificar el sistema (más que para crear nuevas clases y modificar el código que genera nuevos objetos), los programadores pueden utilizar el polimorfismo para incluir tipos adicionales que no se hayan considerado a la hora de crear el sistema.



Observación de ingeniería de software 11.1

El polimorfismo promueve la extensibilidad: el software que invoca el comportamiento polimórfico es independiente de los tipos de los objetos a los cuales se envían los mensajes. Se pueden incorporar nuevos tipos de objetos que pueden responder a las llamadas de los métodos existentes, sin necesidad de modificar el sistema base. Sólo el código cliente que crea instancias de los nuevos objetos debe modificarse para dar cabida a los nuevos tipos.

11.3 Demostración del comportamiento polimórfico

En la sección 10.4 creamos una jerarquía de clases de empleados por comisión, en la cual la clase EmpleadoBaseMasComision heredó de la clase EmpleadoPorComision. Los ejemplos en esa sección manipulan objetos EmpleadoPorComision y EmpleadoBaseMasComision mediante el uso de referencias a ellos para invocar a sus métodos. Dirigimos las referencias a la clase base a los objetos de la clase base, y las referencias a la clase derivada a los objetos de la clase derivada. Estas asignaciones son naturales y directas; las referencias a la clase base están diseñadas para referirse a objetos de la clase base, y las referencias a la clase derivada están diseñadas para referirse a objetos de la clase derivada. No obstante, es posible realizar otras asignaciones.

En el siguiente ejemplo, dirigiremos una referencia a la clase base a un objeto de la clase derivada. Después mostraremos cómo al invocar un método en un objeto de la clase derivada a través de una referencia a la clase base se invoca a la funcionalidad de la clase derivada; el tipo del *objeto actual al que se hace referencia*, no el tipo de *referencia*, es el que determina cuál método se va a llamar. Este ejemplo demuestra el concepto clave de que un objeto de una clase derivada puede tratarse como un objeto de su clase base. Esto permite varias manipulaciones interesantes. Una aplicación puede crear un arreglo de referencias a la clase base, que se refieran a objetos de muchos tipos de clases derivadas. Esto se permite, ya que cada objeto de clase derivada *es un* objeto de su clase base. Por ejemplo, podemos asignar la referencia de un objeto EmpleadoBaseMasComision a una variable EmpleadoPorComision de la clase base, ya que un EmpleadoBasePorComision *es un* EmpleadoPorComision; por lo tanto, podemos tratar a un EmpleadoBasePorComision como un EmpleadoPorComision.

Un objeto de la clase base no es un objeto de ninguna de sus clases derivadas. Por ejemplo, no podemos asignar la referencia de un objeto `EmpleadoPorComision` a una variable de la clase derivada `EmpleadoBaseMasComision`, ya que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`; por ejemplo, un `EmpleadoPorComision` no tiene una variable de instancia `salarioBase` y no tiene una propiedad `SalarioBase`. La relación *es un* se aplica de una clase derivada a sus clases base directa e indirecta, pero no viceversa.

Resulta ser que el compilador permite la asignación de una referencia a la clase base a una variable de la clase derivada, si convertimos explícitamente la referencia a la clase base al tipo de la clase derivada; una técnica que veremos con más detalle en la sección 11.5.6. ¿Para qué nos serviría, en un momento dado, realizar una asignación así? Una referencia a la clase base puede usarse para invocar sólo a los métodos declarados en la clase base; si tratamos de invocar métodos que sólo pertenezcan a la clase derivada a través de una referencia a la clase base se producen errores de compilación. Si una aplicación necesita realizar una operación específica para la clase derivada en un objeto de la clase derivada al que se haga una referencia mediante una variable de la clase base, la aplicación primero debe convertir la referencia a la clase base en una referencia a la clase derivada, mediante una técnica conocida como *conversión descendente*. Esto permite a la aplicación invocar métodos de la clase derivada que no se encuentren en la clase base. En la sección 11.5.6 presentaremos un ejemplo concreto de conversión descendente.

El ejemplo de la figura 11.1 demuestra tres formas de usar variables de la clase base y la clase derivada para almacenar referencias a objetos de la clase base y de la clase derivada. Las primeras dos formas son simples: al igual que en la sección 10.4, asignamos una referencia a la clase base a una variable de la clase base, y asignamos una referencia a la clase derivada a una variable de la clase derivada. Después demostramos la relación entre las clases derivadas y las clases base (es decir, la relación *es un*) mediante la asignación de una referencia a la clase derivada a una variable de la clase base. [Nota: esta aplicación utiliza las clases `EmpleadoPorComision3` y `EmpleadoBaseMasComision4` de las figuras 10.13 y 10.14, respectivamente.]

```

1 // Fig. 11.1: PruebaPolimorfismo.cs
2 // Asignación de referencias a la clase base y la clase derivada a
3 // variables de la clase base y de la clase derivada.
4 using System;
5
6 public class PruebaPolimorfismo
7 {
8     public static void Main( string[] args )
9     {
10         // asigna una referencia a la clase base a una variable de la clase base
11         EmpleadoPorComision3 empleadoPorComision = new EmpleadoPorComision3(
12             "Sue", "Jones", "222-22-2222", 10000.00M, .06M );
13
14         // asigna una referencia a la clase derivada a una variable de la clase derivada
15         EmpleadoBaseMasComision4 empleadoBaseMasComision4 =
16             new EmpleadoBaseMasComision4( "Bob", "Lewis",
17                 "333-33-3333", 5000.00M, .04M, 300.00M );
18
19         // invoca a ToString y a Ingresos en el objeto de la clase base,
20         // usando la variable de la clase base
21         Console.WriteLine( "{0} {1}:\n{n}{2}\n{3}: {4:C}\n",
22             "Llama a ToString de EmpleadoPorComision3 con referencia de clase base",
23             "a objeto de clase base", empleadoPorComision.ToString(),
24             "ingresos", empleadoPorComision.Ingresos() );
25
26         // invoca a ToString e Ingresos en objeto de clase derivada,
27         // usando variable de clase derivada
28         Console.WriteLine( "{0} {1}:\n{n}{2}\n{3}: {4:C}\n",
29             "Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase",

```

Figura 11.1 | Asignación de referencias a la clase base y a la clase derivada a variables de la clase base y de la clase derivada. (Parte 1 de 2).

```

30         "derivada a objeto de clase derivada",
31         empleadoBaseMasComision4.ToString(),
32         "ingresos", empleadoBaseMasComision4.Ingresos() );
33
34         // invoca a ToString e Ingresos en objeto de clase derivada,
35         // usando variable de clase base
36         EmpleadoPorComision3 empleadoPorComision2 =
37             empleadoBaseMasComision4;
38         Console.WriteLine( "{0} {1}:\n{n{2}\n{n{3}: {4:C}",
39             "Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase",
40             "base a objeto de clase derivada",
41             empleadoPorComision2.ToString(), "ingresos",
42             empleadoPorComision2.Ingresos() );
43     } // fin de Main
44 } // fin de la clase PruebaPolimorfismo

```

Llama a ToString de EmpleadoPorComision3 con referencia de clase base a objeto de clase base:

```

empleado por comisión: Sue Jones
número de seguro social: 222-22-2222
ventas brutas: $10,000.00
tarifa de comisión: 0.06
ingresos: $600.00

```

Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase derivada a objeto de clase derivada:

```

salario base + empleado por comisión: Bob Lewis
número de seguro social: 333-33-3333
ventas brutas: $5,000.00
tarifa de comisión: 0.04
salario base: $300.00
ingresos: $500.00

```

Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase base a objeto de clase derivada:

```

salario base + empleado por comisión: Bob Lewis
número de seguro social: 333-33-3333
ventas brutas: $5,000.00
tarifa de comisión: 0.04
salario base: $300.00
ingresos: $500.00

```

Figura 11.1 | Asignación de referencias a la clase base y a la clase derivada a variables de la clase base y de la clase derivada. (Parte 2 de 2).

En la figura 11.1, las líneas 11-12 crean un nuevo objeto `EmpleadoPorComision3` y asignan su referencia a una variable `EmpleadoPorComision3`. Las líneas 15-17 crean un nuevo objeto `EmpleadoBaseMasComision4` y asignan su referencia a una variable `EmpleadoBaseMasComision4`. Estas asignaciones son naturales; por ejemplo, el principal propósito de una variable `EmpleadoPorComision3` es guardar una referencia a un objeto `EmpleadoPorComision3`. Las líneas 21-24 utilizan la referencia `empleadoPorComision` para invocar a los métodos `ToString` e `Ingresos`. Como `empleadoPorComision` hace referencia a un objeto `EmpleadoPorComision3`, se hacen llamadas a los métodos de la versión de la clase base `EmpleadoPorComision3`. De manera similar, las líneas 28-32 utilizan a `empleadoBaseMasComision` para invocar a los métodos `ToString` e `Ingresos` en el objeto `EmpleadoBaseMasComision4`. Esto invoca a la versión de los métodos de la clase derivada `EmpleadoBaseMasComision4`.

Después, las líneas 36-37 asignan la referencia al objeto `EmpleadoBaseMasComision` de la clase derivada a una variable de la clase base `EmpleadoPorComision3`, que las líneas 38-42 utilizan para invocar a los métodos `ToString` e `Ingresos`. En sí, una variable de la clase base que contiene una referencia a un objeto de la clase derivada y se utiliza para llamar a un método virtual, es la que llama a la versión del método de la clase derivada que lo redefine. Por ende, `EmpleadoPorComision2.ToString()` en la línea 41 en realidad llama al método `ToString` de la clase base `EmpleadoBaseMasComision4`. El compilador permite este “cruzamiento”, ya que un objeto de una clase derivada *es un* objeto de su clase base (pero no viceversa). Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la *variable*. Si esa clase contiene la declaración del método apropiada (o hereda uno), el compilador permite compilar la llamada. En tiempo de ejecución, *el tipo del objeto al cual se refiere la variable* es el que determina el método que se va a utilizar.

11.4 Clases y métodos abstractos

Cuando pensamos en un tipo de clase, asumimos que las aplicaciones crearán objetos de ese tipo. No obstante, en algunos casos es conveniente declarar clases para las cuales el programador nunca creará instancias de objetos. A dichas clases se les conoce como *clases abstractas*. Como se utilizan sólo como clases base en jerarquías de herencia, nos referimos a ellas como *clases base abstractas*. Estas clases no pueden utilizarse para instancias objetos, ya que como veremos pronto, las clases abstractas están incompletas; las clases derivadas deben declarar las “piezas faltantes”. En la sección 11.5.1 demostraremos las clases abstractas.

El propósito principal de una clase abstracta es proporcionar una clase base apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común. Por ejemplo, en la jerarquía de *Figura* de la figura 10.3, las clases derivadas heredan la noción de lo que significa ser una *Figura*; los atributos comunes tales como `posicion`, `color` y `grosorBorde`, y los comportamientos tales como `Dibujar`, `Mover`, `CambiarTamano` y `CambiarColor`. Las clases que pueden utilizarse para instanciar objetos se llaman *clases concretas*. Dichas clases proporcionan implementaciones de *cada* método que declaran (algunas de las implementaciones pueden heredarse). Por ejemplo, podríamos derivar las clases concretas `Circulo`, `Cuadrado` y `Triangulo` de la clase base abstracta `FiguraBidimensional`. De manera similar, podríamos derivar las clases concretas `Esfera`, `Cubo` y `Tetraedro` de la clase base abstracta `FiguraTridimensional`. Las clases base abstractas son demasiado generales como para crear objetos reales; sólo especifican lo que tienen en común las clases derivadas. Necesitamos ser más específicos para poder crear objetos. Por ejemplo, si envía el mensaje `Dibujar` a la clase abstracta `FiguraBidimensional`, la clase sabe que las figuras bidimensionales deben poder dibujarse, pero no sabe qué figura específica dibujar, por lo que no puede implementar un verdadero método `Dibujar`. Las clases concretas proporcionan los detalles específicos que hacen razonable la creación de instancias de objetos.

No todas las jerarquías de herencia contienen clases abstractas. Sin embargo, a menudo los programadores escriben código cliente que utiliza sólo tipos de clases bases abstractas para reducir las dependencias del código cliente en un rango de tipos de clases derivadas específicas. Por ejemplo, un programador puede escribir un método con un parámetro de un tipo de clase base abstracta. Cuando se llama, ese método puede recibir un objeto de cualquier clase concreta que extienda en forma directa o indirecta a la clase base especificada como el tipo del parámetro.

Algunas veces las clases abstractas constituyen varios niveles de la jerarquía. Por ejemplo, la jerarquía de *Figura* de la figura 10.3 empieza con la clase abstracta *Figura*. En el siguiente nivel de la jerarquía hay dos clases abstractas más, `FiguraBidimensional` y `FiguraTridimensional`. El siguiente nivel de la jerarquía declara clases concretas para objetos de `FiguraBidimensional` (`Circulo`, `Cuadrado` y `Triangulo`) y para objetos de `FiguraTridimensional` (`Esfera`, `Cubo` y `Tetraedro`).

Para hacer una clase abstracta, ésta se declara con la palabra clave `abstract`. Por lo general, una clase base abstracta contiene uno o más *métodos abstractos*. Un método abstracto tiene la palabra clave `abstract` en su declaración, como en

```
public abstract void Dibujar(); // método abstracto
```

Los métodos abstractos no proporcionan implementaciones. Una clase que contiene métodos abstractos debe declararse como clase abstracta, aún si esa clase contiene métodos concretos (no abstractos). Cada clase derivada concreta de una clase base abstracta también debe proporcionar implementaciones concretas de los métodos

abstractos de la clase base. En la figura 11.4 mostramos un ejemplo de una clase abstracta con un método abstracto.

Las propiedades también pueden declararse como `abstract` y después se redefinen en clases derivadas con la palabra clave `override`, justo igual que los métodos. Esto permite a una clase base abstracta especificar propiedades comunes de sus clases derivadas. Las declaraciones de propiedades abstractas tienen la siguiente forma:

```
public abstract TipoPropiedad MiPropiedad
{
    get;
    set;
} // fin de la propiedad abstracta
```

Los signos de punto y coma después de las palabras clave `get` y `set` indican que no se proporciona la implementación para estos descriptores de acceso. Una propiedad abstracta puede omitir las implementaciones para el descriptor de acceso `get`, el descriptor de acceso `set` o ambos. Las clases derivadas concretas deben proporcionar implementaciones para *cada* descriptor de acceso declarado en la propiedad abstracta. Cuando se especifican ambos descriptores de acceso `get` y `set` (como en la declaración anterior), cada clase derivada concreta debe implementar ambos descriptores. Si se omite un descriptor de acceso, no se permite que la clase derivada lo implemente. Esto produce un error de compilación.

Los constructores y los métodos `static` no pueden declararse `abstract`. Los constructores no se heredan, por lo que un constructor `abstract` nunca podría implementarse. De manera similar, las clases derivadas no pueden redefinir métodos `static`, por lo que un método `abstract static` nunca podría implementarse.



Observación de ingeniería de software 11.2

Una clase abstracta declara los atributos y comportamientos comunes de las diversas clases que heredan de ella, ya sea en forma directa o indirecta, en una jerarquía de clases. Por lo general, una clase abstracta contiene uno o más métodos o propiedades abstractos, que las clases derivadas concretas deben redefinir. Las variables de instancia, los métodos y las propiedades concretas de una clase abstracta están sujetos a las reglas normales de la herencia.



Error común de programación 11.1

Tratar de instanciar un objeto de una clase abstracta es un error de compilación.



Error común de programación 11.2

Si no se implementan los métodos y las propiedades abstractas de la clase base en una clase derivada, se produce un error de compilación, a menos que la clase derivada también se declare como `abstract`.

Aunque no podemos instanciar objetos de clases base abstractas, pronto veremos que *podemos* usar clases base abstractas para declarar variables que puedan guardar referencias a objetos de cualquier clase concreta que se derive de esas clases abstractas. Por lo general, las aplicaciones utilizan dichas variables para manipular los objetos de las clases derivadas mediante el polimorfismo. Además, puede usar los nombres de las clases base abstractas para invocar métodos `static` que estén declarados en esas clases base abstractas.

En especial, el polimorfismo es efectivo para implementar los denominados sistemas de software en capas. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico puede operar en forma muy distinta a los demás. Aún así, los comandos comunes pueden leer o escribir datos desde y hacia los dispositivos. Para cada dispositivo, el sistema operativo utiliza una pieza de software llamada controlador de dispositivos para controlar toda la comunicación entre el sistema y el dispositivo. El mensaje de escritura que se envía a un objeto controlador de dispositivo necesita implementarse de manera específica en el contexto de ese controlador y la forma en que manipula a un dispositivo específico. No obstante, la llamada de escritura en sí no es distinta a la escritura en cualquier otro dispositivo en el sistema: colocar cierto número de bytes de memoria en ese dispositivo. Un sistema operativo orientado a objetos podría usar una clase base abstracta para proporcionar una “interfaz” apropiada para todos los controladores de dispositivos. Después, a través de la herencia de esa clase base abstracta, se forman clases derivadas que se comporten todas de manera similar. Los métodos del controlador de dispositivos se declaran como métodos abstractos en la clase base abstracta. Las implementaciones de estos métodos abstractos se proporcionan en las clases derivadas que corresponden a los tipos específicos de controladores de dispositivos. Siempre se

están desarrollando nuevos dispositivos, a menudo mucho después de que se ha liberado el sistema operativo. Cuando usted compra un nuevo dispositivo, éste incluye un controlador de dispositivo proporcionado por el distribuidor. El dispositivo opera de inmediato, una vez que usted lo conecta a la computadora e instala el controlador de dispositivo. Éste es otro elegante ejemplo acerca de cómo el polimorfismo hace que los sistemas sean extensibles.

En la programación orientada a objetos es común declarar una *clase iteradora* que pueda recorrer todos los objetos en una colección, como un arreglo (capítulo 8) o un arreglo `ArrayList` (capítulo 26, Colecciones). Por ejemplo, una aplicación puede imprimir un arreglo `ArrayList` de objetos creando un objeto iterador, y luego usándolo para obtener el siguiente elemento de la lista cada vez que se llame al iterador. Los iteradores se utilizan comúnmente en la programación polimórfica para recorrer una colección que contiene referencias a objetos de diversas clases en una jerarquía de herencia. (Los capítulos 25-26 presentan un tratamiento detallado de las nuevas capacidades “genéricas”, `ArrayList` y los iteradores). Por ejemplo, un arreglo `ArrayList` de referencias a objetos de la clase `FiguraBidimensional` podría contener referencias a objetos de las clases derivadas `Cuadrado`, `Circulo`, `Triangulo` y así, sucesivamente. Si utilizáramos el polimorfismo para llamar al método `Dibujar` para cada objeto `FiguraBidimensional` mediante una variable `FiguraBidimensional`, se dibujaría a cada objeto correctamente en la pantalla.

11.5 Caso de estudio: sistema de nómina utilizando polimorfismo

En esta sección analizamos de nuevo la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` que exploramos a lo largo de la sección 10.4. Ahora podemos usar un método abstracto y polimorfismo para realizar cálculos de nómina, con base en el tipo de empleado. Vamos a crear una jerarquía de empleados mejorada para resolver el siguiente problema:

Una compañía paga a sus empleados por semana. Los empleados son de cuatro tipos: empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas; empleados por horas, que reciben un sueldo por hora y pago por tiempo extra, por todas las horas trabajadas que excedan a 40 horas; empleados por comisión, que reciben un porcentaje de sus ventas y empleados asalariados por comisión, que reciben un salario base más un porcentaje de sus ventas. Para este periodo de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, agregando un 10% a sus salarios base. La compañía desea implementar una aplicación en C# que realice sus cálculos de nómina en forma polimórfica.

Utilizaremos la clase abstracta `Empleado` para representar el concepto general de un empleado. Las clases que extienden a `Empleado` son `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoPorHoras`. La clase `EmpleadoBaseMasComision` (que extiende a `EmpleadoPorComision`) representa el último tipo de empleado. El diagrama de clases de UML en la figura 11.2 muestra la jerarquía de herencia para nuestra aplicación polimórfica de nómina de empleados. Observe que la clase abstracta `Empleado` está en cursivas, según la convención de UML.

La clase base abstracta `Empleado` declara la “interfaz” para la jerarquía; esto es, el conjunto de métodos que puede invocar una aplicación en todos los objetos `Empleado`. Aquí utilizamos el término “interfaz” en un sentido general, para referirnos a las diversas formas en que las aplicaciones pueden comunicarse con los objetos de cualquier clase derivada de `Empleado`. Tenga cuidado de no confundir la noción general de una “interfaz” con la noción formal de una interfaz en C#, el tema de la sección 11.7. Cada empleado, sin importar la manera en que se calculen sus ingresos, tiene un primer nombre, un apellido paterno y un número de seguro social, por lo que las variables de instancia `private primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` aparecen en la clase base abstracta `Empleado`.



Observación de ingeniería de software 11.3

*Una clase derivada puede heredar la “interfaz” o “implementación” de una clase base. Las jerarquías diseñadas para la **herencia de implementación** tienden a tener su funcionalidad en niveles altos de la jerarquía; cada nueva clase derivada hereda uno o más métodos que se implementaron en una clase base, y la clase derivada utiliza las implementaciones de la clase base. Las jerarquías diseñadas para la **herencia de interfaz** tienden a tener su funcionalidad en niveles bajos de la jerarquía; una clase base específica uno o más métodos abstractos que deben declararse para cada clase concreta en la jerarquía, y las clases derivadas individuales redefinen estos métodos para proporcionar la implementación específica para cada clase derivada.*

Las siguientes secciones implementan la jerarquía de clases de `Empleado`. La primera sección implementa la clase base abstract `Empleado`. Las siguientes cuatro secciones implementan cada una de las clases concretas. La sexta sección implementa una aplicación de prueba que crea objetos de todas estas clases y procesa esos objetos mediante el polimorfismo.

11.5.1 Creación de la clase base abstracta `Empleado`

La clase `Empleado` (figura 11.4) proporciona los métodos `Ingresos` y `ToString`, además de las propiedades que manipulan las variables de instancia de `Empleado`. Es evidente que un método `Ingresos` se aplica en forma general a todos los empleados. Pero cada cálculo de los ingresos depende de la clase de empleado. Por lo tanto, declaramos a `Ingresos` como abstract en la clase base `Empleado`, ya que una implementación predeterminada no tiene sentido para ese método; no hay suficiente información para determinar qué monto debe devolver `Ingresos`. Cada una de las clases derivadas redefine a `Ingresos` con una implementación apropiada. Para calcular los ingresos de un empleado, la aplicación asigna una referencia al objeto de empleado a una variable de la clase base `Empleado`, y después invoca al método `Ingresos` en esa variable. Mantenemos un arreglo de variables `Empleado`, cada una de las cuales guarda una referencia a un objeto `Empleado` (desde luego que no puede haber objetos `Empleado`, ya que ésta es una clase abstracta; sin embargo, debido a la herencia todos los objetos de todas las clases derivadas de `Empleado` pueden considerarse como objetos `Empleado`). La aplicación itera a través del arreglo y llama al método `Ingresos` para cada objeto `Empleado`. C# procesa estas llamadas a los métodos en forma polimórfica. Al incluir a `Ingresos` como un método abstracto en `Empleado`, se obliga a cada clase concreta derivada de `Empleado` a redefinir `Ingresos` con un método que realice el cálculo apropiado del salario.

El método `ToString` en la clase `Empleado` devuelve un objeto `string` que contiene el primer nombre, el apellido paterno y el número de seguro social del empleado. Cada clase derivada de `Empleado` redefine al método `ToString` para crear una representación `string` de un objeto de esa clase que contiene el tipo del empleado (por ejemplo, "empleado asalariado:"), seguido del resto de la información del empleado.

El diagrama en la figura 11.3 muestra cada una de las cinco clases en la jerarquía, hacia abajo en la columna de la izquierda, y los métodos `Ingresos` y `ToString` en la fila superior. Para cada clase, el diagrama muestra los resultados deseados de cada método. [Nota: no listamos las propiedades de la clase base `Empleado` porque no se redefinen en ninguna de las clases derivadas; cada una de estas propiedades se hereda y cada una de las clases derivadas las utiliza "como están".]

Consideremos ahora la declaración de la clase `Empleado` (figura 11.4). Esta clase incluye un constructor que recibe el primer nombre, el apellido paterno y el número de seguro social como argumentos (líneas 10-15); las propiedades de sólo lectura para obtener el primer nombre, el apellido paterno y el número de seguro social (líneas 18-24, 27-33 y 36-42, respectivamente); el método `ToString` (líneas 45-49), el cual utiliza las propiedades para devolver la representación `string` de `Empleado`; y el método abstract `Ingresos` (línea 52), que las clases derivadas concretas deben implementar. Observe que el constructor de `Empleado` no valida el número de seguro social en este ejemplo. Por lo general, se debe proporcionar esa validación.

¿Por qué declaramos a `Ingresos` como un método abstracto? Simplemente, no tiene sentido proporcionar una implementación de este método en la clase `Empleado`. No podemos calcular los ingresos para un `Empleado`

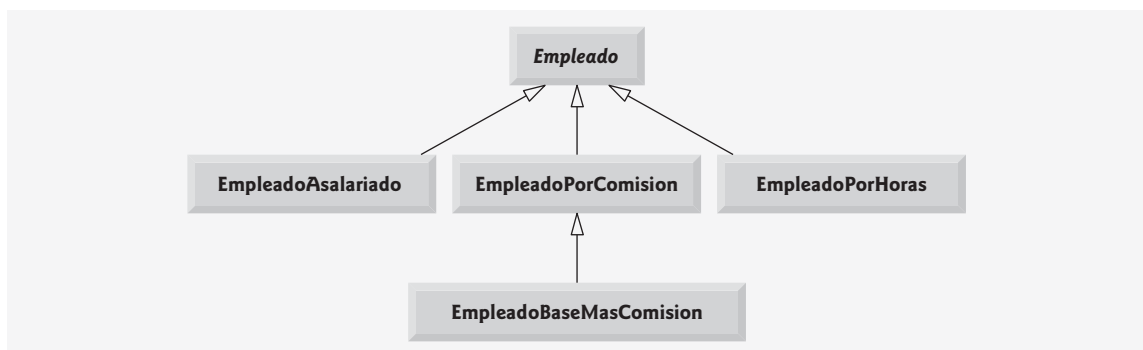


Figura 11.2 | Diagrama de clases de UML para la jerarquía de `Empleado`.

	Ingresos	ToString
Empleado	abstract	<i>primerNombre apellidoPaterno</i> número de seguro social: <i>NSS</i>
Empleado-Asalariado	salarioSemanal	empleado asalariado: <i>primerNombre apellidoPaterno</i> número de seguro social: <i>NSS</i> salario semanal: <i>salarioSemanal</i>
Empleado-PorHoras	<i>Si horas <= 40 sueldo * horas Si horas > 40 40 * sueldo + (horas - 40) * sueldo * 1.5</i>	empleado por horas: <i>primerNombre apellidoPaterno</i> número de seguro social: <i>NSS</i> sueldo por horas: <i>sueldo</i> horas trabajadas: <i>horas</i>
Empleado-PorComision	tarifaComision * ventasBrutas	empleado por comisión: <i>primerNombre apellidoPaterno</i> número de seguro social: <i>NSS</i> ventas brutas: <i>ventasBrutas</i> tarifa de comisión: <i>tarifaComision</i>
Empleado-BaseMas-Comision	(tarifaComision * ventasBrutas) + salarioBase	empleado por comisión con salario base: <i>primerNombre apellidoPaterno</i> número de seguro social: <i>NSS</i> ventas brutas: <i>ventasBrutas</i> tarifa de comisión: <i>tarifaComision</i> salario base: <i>salarioBase</i>

Figura 11.3 | Interfaz polimórfica para las clases de la jerarquía de Empleado.

general; primero debemos conocer el tipo de Empleado específico para determinar el cálculo apropiado de los ingresos. Al declarar este método abstract, indicamos que cada clase derivada concreta *debe* proporcionar una implementación apropiada para Ingresos, y que una aplicación podrá utilizar las variables de la clase base Empleado para invocar al método Ingresos en forma polimórfica, para cualquier tipo de Empleado.

```

1  // Fig. 11.4: Empleado.cs
2  // La clase base abstracta Empleado.
3  public abstract class Empleado
4  {
5      private string primerNombre;
6      private string apellidoPaterno;
7      private string numeroSeguroSocial;
8
9      // constructor con tres parámetros
10     public Empleado( string nombre, string apellido, string nss )
11     {
12         primerNombre = nombre;
13         apellidoPaterno = apellido;
14         numeroSeguroSocial = nss;
15     } // fin de constructor de Empleado con tres parámetros
16
17     // propiedad de sólo lectura que obtiene el primer nombre del empleado
18     public string PrimerNombre
19     {

```

Figura 11.4 | La clase base abstracta Empleado. (Parte 1 de 2).

```

20     get
21     {
22         return primerNombre;
23     } // fin de get
24 } // fin de la propiedad PrimerNombre
25
26 // propiedad de sólo lectura que obtiene el apellido paterno del empleado
27 public string ApellidoPaterno
28 {
29     get
30     {
31         return apellidoPaterno;
32     } // fin de get
33 } // fin de la propiedad ApellidoPaterno
34
35 // propiedad de sólo lectura que obtiene el número de seguro social del empleado
36 public string NumeroSeguroSocial
37 {
38     get
39     {
40         return numeroSeguroSocial;
41     } // fin de get
42 } // fin de la propiedad NumeroSeguroSocial
43
44 // devuelve representación string del objeto Empleado, usando las propiedades
45 public override string ToString()
46 {
47     return string.Format( "{0} {1}\nnúmero de seguro social: {2}",
48         PrimerNombre, ApellidoPaterno, NumeroSeguroSocial );
49 } // fin del método ToString
50
51 // método abstracto redefinido por las clases derivadas
52 public abstract decimal Ingresos(); // no hay implementación aquí
53 } // fin de la clase abstracta Empleado

```

Figura 11.4 | La clase base abstracta Empleado. (Parte 2 de 2).

11.5.2 Creación de la clase derivada concreta EmpleadoAsalariado

La clase EmpleadoAsalariado (figura 11.5) extiende a la clase Empleado (línea 3) y redefine a Ingresos (líneas 28-31), lo cual convierte a EmpleadoAsalariado en una clase concreta. La clase incluye un constructor (líneas 8-12) que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal como argumentos; la propiedad SalarioSemanal para manipular la variable de instancia salarioSemanal, incluyendo un descriptor de acceso set que asegura que asignemos sólo valores no negativos a salarioSemanal (líneas 15-25); el método Ingresos (líneas 28-31) para calcular los ingresos de un EmpleadoAsalariado; y el método ToString (líneas 34-38) que devuelve un objeto string que incluye el tipo del empleado, a saber, "empleado asalariado: ", seguido de la información específica para el empleado producida por el método ToString de la clase base Empleado y la propiedad SalarioSemanal de EmpleadoAsalariado. El constructor de la clase EmpleadoAsalariado pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de Empleado (línea 9) a través de un inicializador de constructor, para inicializar las variables de instancia private que no se heredan de la clase base. El método Ingresos redefine al método abstracto Ingresos de Empleado para proporcionar una implementación concreta que devuelva el salario semanal del EmpleadoAsalariado. Si no implementamos Ingresos, la clase EmpleadoAsalariado debe declararse como abstract; en caso contrario, se produce un error de compilación (y desde luego, queremos que EmpleadoAsalariado sea una clase concreta).

El método ToString (líneas 34-38) de la clase EmpleadoAsalariado redefine al método ToString de Empleado. Si la clase EmpleadoAsalariado no redefiniera a ToString, EmpleadoAsalariado habría heredado la versión de ToString de Empleado. En ese caso, el método ToString de EmpleadoAsalariado simplemente

```

1  // Fig. 11.5: EmpleadoAsalariado.cs
2  // La clase EmpleadoAsalariado que extiende a Empleado.
3  public class EmpleadoAsalariado : Empleado
4  {
5      private decimal salarioSemanal;
6
7      // constructor con cuatro parámetros
8      public EmpleadoAsalariado( string nombre, string apellido, string nss,
9          decimal salario ) : base( nombre, apellido, nss )
10     {
11         SalarioSemanal = salario; // valida el salario a través de una propiedad
12     } // fin del constructor de EmpleadoAsalariado con cuatro parámetros
13
14     // propiedad que obtiene y establece el salario del empleado asalariado
15     public decimal SalarioSemanal
16     {
17         get
18         {
19             return salarioSemanal;
20         } // fin de get
21         set
22         {
23             salarioSemanal = ( ( value >= 0 ) ? value : 0 ); // validación
24         } // fin de set
25     } // fin de la propiedad SalarioSemanal
26
27     // calcula los ingresos; redefine el método abstracto Ingresos en Empleado
28     public override decimal Ingresos()
29     {
30         return SalarioSemanal;
31     } // fin del método Ingresos
32
33     // devuelve representación string del objeto EmpleadoAsalariado
34     public override string ToString()
35     {
36         return string.Format( "empleado asalariado: {0}\n{1}: {2:C}",
37             base.ToString(), "salario semanal", SalarioSemanal );
38     } // fin del método ToString
39 } // fin de la clase EmpleadoAsalariado

```

Figura 11.5 | La clase EmpleadoAsalariado que extiende a Empleado.

devolvería el nombre completo del empleado y su número de seguro social, lo cual no representa en forma adecuada a un EmpleadoAsalariado. Para producir una representación string completa de EmpleadoAsalariado, el método ToString de la clase derivada devuelve "empleado asalariado: ", seguido de la información específica de la clase base Empleado (es decir, el primer nombre, el apellido paterno y el número de seguro social) que se obtiene al invocar el método ToString de la clase base (línea 37); éste es un excelente ejemplo de reutilización de código. La representación string de un EmpleadoAsalariado también contiene el salario semanal del empleado, el cual se obtiene mediante el uso de la propiedad SalarioSemanal de la clase.

11.5.3 Creación de la clase derivada concreta EmpleadoPorHoras

La clase EmpleadoPorHoras (figura 11.6) también extiende a la clase Empleado (línea 3). La clase incluye un constructor (líneas 9-15) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un sueldo por horas y el número de horas trabajadas. Las líneas 18-28 y 31-42 declaran las propiedades Sueldo y Horas para las variables de instancia sueldo y horas, respectivamente. El descriptor de acceso set en la propiedad Sueldo (líneas 24-27) asegura que sueldo sea no negativo, y el descriptor de acceso set en la propiedad Horas (líneas 37-41) asegura que horas se encuentre en el rango de 0-168 (el número total de horas

en la semana). La clase `EmpleadoPorHoras` también incluye el método `Ingresos` (líneas 45-51) para calcular los ingresos de un `EmpleadoPorHoras`; y el método `ToString` (líneas 54-59), que devuelve el tipo del empleado, a saber, "empleado por horas: ", e información específica para ese empleado. Observe que el constructor de `EmpleadoPorHoras`, al igual que el constructor de `EmpleadoAsalariado`, pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de la clase base `Empleado` (línea 11) para inicializar las variables de instancia `private` de la clase base. Además, el método `ToString` llama al método `ToString` de la clase base (línea 58) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social); éste es otro excelente ejemplo de reutilización de código.

```

1  // Fig. 11.6: EmpleadoPorHoras.cs
2  // La clase EmpleadoPorHoras que extiende a Empleado.
3  public class EmpleadoPorHoras : Empleado
4  {
5      private decimal sueldo; // sueldo por hora
6      private decimal horas; // horas trabajadas durante la semana
7
8      // constructor con cinco parámetros
9      public EmpleadoPorHoras( string nombre, string apellido, string nss,
10         decimal sueldoPorHoras, decimal horasTrabajadas )
11         : base( nombre, apellido, nss )
12     {
13         Sueldo = sueldoPorHoras; // valida el sueldo por horas a través de una propiedad
14         Horas = horasTrabajadas; // valida las horas trabajadas a través de una propiedad
15     } // fin del constructor de EmpleadoPorHoras con cinco parámetros
16
17     // propiedad que obtiene y establece el sueldo del empleado por horas
18     public decimal Sueldo
19     {
20         get
21         {
22             return sueldo;
23         } // fin de get
24         set
25         {
26             sueldo = ( value >= 0 ) ? value : 0; // validación
27         } // fin de set
28     } // fin de la propiedad Sueldo
29
30     // propiedad que obtiene y establece las horas del empleado por horas
31     public decimal Horas
32     {
33         get
34         {
35             return horas;
36         } // fin de get
37         set
38         {
39             horas = ( ( value >= 0 ) && ( value <= 168 ) ) ?
40                 value : 0; // validación
41         } // fin de set
42     } // fin de la propiedad Horas
43
44     // calcula los ingresos; redefine el método abstracto Ingresos de Empleado
45     public override decimal Ingresos()
46     {
47         if ( Horas <= 40 ) // no hay tiempo extra

```

Figura 11.6 | La clase `EmpleadoPorHoras` que extiende a `Empleado`. (Parte 1 de 2).

```

48     return Sueldo * Horas;
49     else
50         return ( 40 * Sueldo ) + ( ( Horas - 40 ) * Sueldo * 1.5M );
51 } // fin del método Ingresos
52
53 // devuelve representación string del objeto EmpleadoPorHoras
54 public override string ToString()
55 {
56     return string.Format(
57         "empleado por horas: {0}\n{1}: {2:C}; {3}: {4:F2}",
58         base.ToString(), "sueldo por horas", Sueldo, "horas trabajadas", Horas );
59 } // fin del método ToString
60 } // fin de la clase EmpleadoPorHoras

```

Figura 11.6 | La clase EmpleadoPorHoras que extiende a Empleado. (Parte 2 de 2).

11.5.4 Creación de la clase derivada concreta EmpleadoPorComision

La clase EmpleadoPorComision (figura 11.7) extiende a la clase Empleado (línea 3). Esta clase incluye a un constructor (líneas 9-14) que recibe como argumentos un primer nombre, un apellido, un número de seguro social, un monto de ventas y una tarifa de comisión; las propiedades (líneas 17-28 y 31-41) para las variables de instancia tarifaComision y ventasBrutas, respectivamente; el método Ingresos (líneas 44-47) para calcular los ingresos de un EmpleadoPorComision; y el método ToString (líneas 50-55) que devuelve el tipo del empleado, a saber, "empleado por comisión: ", e información específica del empleado.

```

1 // Fig. 11.7: EmpleadoPorComision.cs
2 // La clase EmpleadoPorComision que extiende a Empleado.
3 public class EmpleadoPorComision : Empleado
4 {
5     private decimal ventasBrutas; // ventas semanales totales
6     private decimal tarifaComision; // porcentaje de comisión
7
8     // constructor con cinco parámetros
9     public EmpleadoPorComision( string nombre, string apellido, string nss,
10         decimal ventas, decimal tarifa ) : base( nombre, apellido, nss )
11     {
12         VentasBrutas = ventas; // valida las ventas brutas a través de una propiedad
13         TarifaComision = tarifa; // valida la tarifa de comisión a través de una propiedad
14     } // fin del constructor de EmpleadoPorComision con cinco parámetros
15
16     // propiedad que obtiene y establece la tarifa de comisión del empleado por comisión
17     public decimal TarifaComision
18     {
19         get
20         {
21             return tarifaComision;
22         } // fin de get
23         set
24         {
25             tarifaComision = ( value > 0 && value < 1 ) ?
26                 value : 0; // validación
27         } // fin de set
28     } // fin de la propiedad TarifaComision
29 }

```

Figura 11.7 | La clase EmpleadoPorHoras que extiende a Empleado. (Parte 1 de 2).

```

30 // propiedad que obtiene y establece las ventas brutas del empleado por comisión
31 public decimal VentasBrutas
32 {
33     get
34     {
35         return ventasBrutas;
36     } // fin de get
37     set
38     {
39         ventasBrutas = ( value >= 0 ) ? value : 0; // validación
40     } // fin de set
41 } // fin de la propiedad VentasBrutas
42
43 // calcula los ingresos; redefine el método abstracto Ingresos en Empleado
44 public override decimal Ingresos()
45 {
46     return TarifaComision * VentasBrutas;
47 } // fin del método Ingresos
48
49 // devuelve representación string del objeto EmpleadoPorComision
50 public override string ToString()
51 {
52     return string.Format( "{0}: {1}\n{2}: {3:C}\n{4}: {5:F2}",
53         "empleado por comisión", base.ToString(),
54         "ventas brutas", VentasBrutas, "tarifa de comisión", TarifaComision );
55 } // fin del método ToString
56 } // fin de la clase EmpleadoPorComision

```

Figura 11.7 | La clase EmpleadoPorHoras que extiende a Empleado. (Parte 2 de 2).

El constructor de EmpleadoPorComision también pasa el primer nombre, el apellido y el número de seguro social al constructor de Empleado (línea 10) para inicializar las variables de instancia `private` de Empleado. El método `ToString` llama al método `ToString` de la clase base (línea 53) para obtener la información específica del Empleado (es decir, primer nombre, apellido paterno y número de seguro social).

11.5.5 Creación de la clase derivada concreta indirecta EmpleadoBaseMasComision

La clase EmpleadoBaseMasComision (figura 11.8) extiende a la clase EmpleadoPorComision (línea 3) y, por lo tanto, es una clase derivada indirecta de la clase Empleado. La clase EmpleadoBaseMasComision tiene un constructor (líneas 8-13) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas, una tarifa de comisión y un salario base. Después pasa el primer nombre, el apellido paterno, el número de seguro social, el monto de ventas y la tarifa de comisión al constructor de EmpleadoPorComision (línea 10) para inicializar los miembros de datos `private` de la clase base. EmpleadoBaseMasComision también contiene la propiedad SalarioBase (líneas 17-27) para manipular la variable de instancia `salarioBase`. El método Ingresos (líneas 30-33) calcula los ingresos de un EmpleadoBaseMasComision. Observe que la línea 32 en el método Ingresos llama al método Ingresos de la clase base EmpleadoPorComision para calcular la porción basada en la comisión de los ingresos del empleado. Éste es otro buen ejemplo de reutilización de código. El método ToString de EmpleadoBaseMasComision (líneas 36-40) crea una representación string de un EmpleadoBaseMasComision, la cual contiene "salario base +", seguida del objeto string que se obtiene al invocar el método ToString de la clase base EmpleadoPorComision (otro buen ejemplo de reutilización de código), y después el salario base. El resultado es un objeto string que empieza con "salario base + empleado por comisión", seguido del resto de la información de EmpleadoBaseMasComision. Recuerde que el método ToString de EmpleadoPorComision obtiene el primer nombre, el apellido paterno y el número de seguro social del empleado mediante la invocación del método ToString de su clase base (es decir, Empleado); otro ejemplo


```

1  // Fig. 11.8: EmpleadoBaseMasComision.cs
2  // La clase EmpleadoBaseMasComision que extiende a EmpleadoPorComision.
3  public class EmpleadoBaseMasComision : EmpleadoPorComision
4  {
5      private decimal salarioBase; // salario base por semana
6
7      // constructor con seis parámetros
8      public EmpleadoBaseMasComision( string nombre, string apellido,
9          string nss, decimal ventas, decimal tarifa, decimal salario )
10         : base( nombre, apellido, nss, ventas, tarifa )
11     {
12         SalarioBase = salario; // valida el salario base a través de una propiedad
13     } // fin del constructor de EmpleadoBaseMasComision con seis parámetros
14
15     // propiedad que obtiene y establece
16     // el salario base del empleado por comisión con salario base
17     public decimal SalarioBase
18     {
19         get
20         {
21             return salarioBase;
22         } // fin de get
23         set
24         {
25             salarioBase = ( value >= 0 ) ? value : 0; // validación
26         } // fin de set
27     } // fin de la propiedad SalarioBase
28
29     // calcula los ingresos; redefine el método Ingresos en EmpleadoPorComision
30     public override decimal Ingresos()
31     {
32         return SalarioBase + base.Ingresos();
33     } // fin del método Ingresos
34
35     // devuelve representación string del objeto EmpleadoBaseMasComision
36     public override string ToString()
37     {
38         return string.Format( "{0} {1}; {2}: {3:C}",
39             "salario base +", base.ToString(), "salario base", SalarioBase );
40     } // fin del método ToString
41 } // fin de la clase EmpleadoBaseMasComision

```

Figura 11.8 | La clase EmpleadoBaseMasComision que extiende a EmpleadoPorComision.

más de reutilización de código. Observe que el método ToString de EmpleadoBaseMasComision inicia una cadena de llamadas a métodos que abarcan los tres niveles de la jerarquía de Empleado.

11.5.6 El procesamiento polimórfico, el operador `is` y la conversión descendente

Para probar nuestra jerarquía de Empleado, la aplicación en la figura 11.9 crea un objeto de cada una de las cuatro clases concretas EmpleadoAsalariado, EmpleadoPorHoras, EmpleadoPorComision y EmpleadoBaseMasComision. La aplicación manipula estos objetos, primero mediante variables del mismo tipo de cada objeto y después mediante el polimorfismo, utilizando un arreglo de variables Empleado. Al procesar los objetos mediante el polimorfismo, la aplicación incrementa el salario base de cada EmpleadoBaseMasComision en 10% (desde luego que para esto se requiere determinar el tipo del objeto en tiempo de ejecución). Por último, la aplicación determina e imprime en forma polimórfica el tipo de cada objeto en el arreglo Empleado. Las líneas 10-20 crean objetos de cada una de las cuatro clases concretas derivadas de Empleado. Las líneas 24-32 imprimen en pantalla

la representación `string` y los ingresos de cada uno de estos objetos. Observe que `Write` llama en forma implícita al método `ToString` de cada objeto, cuando éste se imprime en pantalla como un objeto `string` con elementos de formato.

```

1 // Fig. 11.9: PruebaSistemaNomina.cs
2 // Aplicación de prueba de la jerarquía de Empleado.
3 using System;
4
5 public class PruebaSistemaNomina
6 {
7     public static void Main( string[] args )
8     {
9         // crea objetos de clases derivadas
10        EmpleadoAsalariado empleadoAsalariado =
11            new EmpleadoAsalariado( "John", "Smith", "111-11-1111", 800.00M );
12        EmpleadoPorHoras empleadoPorHoras =
13            new EmpleadoPorHoras( "Karen", "Price",
14                "222-22-2222", 16.75M, 40.0M );
15        EmpleadoPorComision empleadoPorComision =
16            new EmpleadoPorComision( "Sue", "Jones",
17                "333-33-3333", 10000.00M, .06M );
18        EmpleadoBaseMasComision empleadoBaseMasComision =
19            new EmpleadoBaseMasComision( "Bob", "Lewis",
20                "444-44-4444", 5000.00M, .04M, 300.00M );
21
22        Console.WriteLine( "Empleados procesados en forma individual:\n" );
23
24        Console.WriteLine( "{0}\n{1}: {2:C}\n",
25            empleadoAsalariado, "ingresos", empleadoAsalariado.Ingresos() );
26        Console.WriteLine( "{0}\n{1}: {2:C}\n",
27            empleadoPorHoras, "ingresos", empleadoPorHoras.Ingresos() );
28        Console.WriteLine( "{0}\n{1}: {2:C}\n",
29            empleadoPorComision, "ingresos", empleadoPorComision.Ingresos() );
30        Console.WriteLine( "{0}\n{1}: {2:C}\n",
31            empleadoBaseMasComision,
32            "ingresos", empleadoBaseMasComision.Ingresos() );
33
34        // crea arreglo Empleado de cuatro elementos
35        Empleado[] empleados = new Empleado[ 4 ];
36
37        // inicializa arreglo con objetos Empleado de tipos derivados
38        empleados[ 0 ] = empleadoAsalariado;
39        empleados[ 1 ] = empleadoPorHoras;
40        empleados[ 2 ] = empleadoPorComision;
41        empleados[ 3 ] = empleadoBaseMasComision;
42
43        Console.WriteLine( "Empleados procesados en forma polimórfica:\n" );
44
45        // procesa en forma generica cada elemento en el arreglo de empleados
46        foreach ( Empleado empleadoActual in empleados )
47        {
48            Console.WriteLine( empleadoActual ); // invoca a ToString
49
50            // determina si el elemento es un EmpleadoBaseMasComision
51            if ( empleadoActual is EmpleadoBaseMasComision )
52            {

```

Figura 11.9 | Aplicación de prueba de la jerarquía de `Empleado`. (Parte 1 de 3).

```

53         // conversión descendente de referencia de Empleado a
54         // referencia de EmpleadoBaseMasComision
55         EmpleadoBaseMasComision empleado =
56             ( EmpleadoBaseMasComision ) empleadoActual;
57
58         empleado.SalarioBase *= 1.10M;
59         Console.WriteLine(
60             "nuevo salario base con incremento del 10%: {0:C}",
61             empleado.SalarioBase );
62     } // fin de if
63
64     Console.WriteLine(
65         "ingresos {0:C}\n", empleadoActual.Ingresos() );
66 } // fin de foreach
67
68 // obtiene el nombre del tipo de cada objeto en el arreglo de empleados
69 for ( int j = 0; j < empleados.Length; j++ )
70     Console.WriteLine( "Empleado {0} es un {1}", j,
71         empleados[ j ].GetType() );
72 } // fin de Main
73 } // fin de la clase PruebaSistemaNomina

```

Empleados procesados en forma individual:

empleado asalariado: John Smith
 número de seguro social: 111-11-1111
 salario semanal: \$800.00
 ingresos: \$800.00

empleado por horas: Karen Price
 número de seguro social: 222-22-2222
 sueldo por horas: \$16.75; horas trabajadas: 40.00
 ingresos: \$670.00

empleado por comisión: Sue Jones
 número de seguro social: 333-33-3333
 ventas brutas: \$10,000.00
 tarifa de comisión: 0.06
 ingresos: \$600.00

salario base + empleado por comisión: Bob Lewis
 número de seguro social: 444-44-4444
 ventas brutas: \$5,000.00
 tarifa de comisión: 0.04; salario base: \$300.00
 ingresos: \$500.00

Empleados procesados en forma polimórfica:

empleado asalariado: John Smith
 número de seguro social: 111-11-1111
 salario semanal: \$800.00
 ingresos \$800.00

empleado por horas: Karen Price
 número de seguro social: 222-22-2222
 sueldo por horas: \$16.75; horas trabajadas: 40.00
 ingresos \$670.00

Figura 11.9 | Aplicación de prueba de la jerarquía de Empleado. (Parte 2 de 3).

```

empleado por comisión: Sue Jones
número de seguro social: 333-33-3333
ventas brutas: $10,000.00
tarifa de comisión: 0.06
ingresos $600.00

salario base + empleado por comisión: Bob Lewis
número de seguro social: 444-44-4444
ventas brutas: $5,000.00
tarifa de comisión: 0.04; salario base: $300.00
nuevo salario base con incremento del 10%: $330.00
ingresos $530.00

Empleado 0 es un EmpleadoAsalariado
Empleado 1 es un EmpleadoPorHoras
Empleado 2 es un EmpleadoPorComision
Empleado 3 es un EmpleadoBaseMasComision

```

Figura 11.9 | Aplicación de prueba de la jerarquía de Empleado. (Parte 3 de 3).

La línea 35 declara a `empleados` y le asigna un arreglo de cuatro variables `Empleado`. Las líneas 38-41 asignan un objeto `EmpleadoAsalariado`, un objeto `EmpleadoPorHoras`, un objeto `EmpleadoPorComision` y un objeto `EmpleadoBaseMasComision` a `empleados[ 0 ]`, `empleados[ 1 ]`, `empleados[ 2 ]` y `empleados[ 3 ]`, respectivamente. Cada asignación es permitida, ya que un `EmpleadoAsalariado` *es un* `Empleado`, un `EmpleadoPorHoras` *es un* `Empleado`, un `EmpleadoPorComision` *es un* `Empleado` y un `EmpleadoBaseMasComision` *es un* `Empleado`. Por lo tanto, podemos asignar las referencias de los objetos `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision` a variables de la clase base `Empleado`, aun cuando ésta es una clase abstracta.

Las líneas 46-66 iteran a través del arreglo `empleados` e invocan los métodos `ToString` e `Ingresos` con la variable `empleadoActual` de `Empleado`, a la cual se le asigna la referencia a un `Empleado` distinto en el arreglo, durante cada iteración. Los resultados ilustran que en definitivo se invocan los métodos apropiados para cada clase. Todas las llamadas a los métodos `ToString` e `Ingresos` se resuelven en tiempo de ejecución, con base en el tipo del objeto al que `empleadoActual` hace referencia. Este proceso se conoce como *vinculación dinámica* o *vinculación postergada*. Por ejemplo, la línea 48 invoca en forma implícita al método `ToString` del objeto al que `empleadoActual` hace referencia. Como resultado de la vinculación dinámica, el CLR decide qué método `ToString` de cuál clase va a llamar en tiempo de ejecución, en vez de hacerlo en tiempo de compilación. Observe que sólo los métodos de la clase `Empleado` pueden llamarse a través de una variable `Empleado`; y desde luego que `Empleado` incluye los métodos de la clase `object`, como `ToString`. (En la sección 10.7 vimos el conjunto de métodos que todas las clases heredan de la clase `object`). Una referencia a la clase base puede utilizarse para invocar sólo a métodos de la clase base.

Realizamos un procesamiento especial en los objetos `EmpleadoBasePorComision`; a medida que los encontramos, incrementamos su salario base en 10%. Cuando procesamos objetos en forma polimórfica, por lo general no necesitamos preocuparnos por los “detalles específicos”, pero para ajustar el salario base, tenemos que determinar el tipo específico de cada objeto `Empleado` en tiempo de ejecución. La línea 51 utiliza el operador `is` para determinar si el tipo de cierto objeto `Empleado` es `EmpleadoBaseMasComision`. La condición en la línea 51 es verdadera si el objeto al que hace referencia `empleadoActual` *es un* `EmpleadoBaseMasComision`. Esto también sería verdadero para cualquier objeto de una clase derivada de `EmpleadoBaseMasComision` (si hubiera alguna), debido a la relación *es un* que tiene una clase derivada con su clase base. Las líneas 55-65 realizan una conversión descendente en `empleadoActual`, del tipo `Empleado` al tipo `EmpleadoBaseMasComision`; esta conversión se permite sólo si el objeto tiene una relación *es un* con `EmpleadoBaseMasComision`. La condición en la línea 51 asegura que éste sea el caso. Esta conversión se requiere si vamos a utilizar la propiedad `SalarioBase` de la clase derivada `EmpleadoBaseMasComision` en el objeto `Empleado` actual; si tratamos de invocar a un método que pertenezca sólo a la clase derivada en una referencia a la clase base, se produce un error de compilación.



Error común de programación 11.3

Asignar una variable de la clase base a una variable de la clase derivada (sin una conversión descendente explícita) es un error de compilación.



Observación de ingeniería de software 11.4

Si en tiempo de ejecución se asigna la referencia a un objeto de la clase derivada a una variable de una de sus clases base directas o indirectas, es aceptable convertir la referencia almacenada en esa variable de la clase base, de vuelta a una referencia del tipo de la clase derivada. Antes de realizar dicha conversión, use el operador `is` para asegurar que el objeto sea de un tipo de clase derivada apropiado.



Error común de programación 11.4

Al realizar una conversión descendente sobre un objeto, se produce una excepción `InvalidCastException` (del espacio de nombres `System`) si, en tiempo de ejecución, el objeto no tiene una relación “es un” con el tipo especificado en el operador de conversión. Un objeto puede convertirse sólo a su propio tipo o al tipo de una de sus clases base.

Si la expresión `is` en la línea 51 es `true`, la instrucción `if` (líneas 51-62) realiza el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. Usando la variable `empleado` de `EmpleadoBaseMasComision`, la línea 58 utiliza la propiedad `SalarioBase`, que sólo pertenece a la clase derivada, para extraer y actualizar el salario base del empleado con el aumento del 10 por ciento.

Las líneas 64-65 invocan al método `Ingresos` en `empleadoActual`, el cual llama al método `Ingresos` del objeto de la clase derivada apropiada en forma polimórfica. Observe que al obtener en forma polimórfica los ingresos del `EmpleadoAsalariado`, el `EmpleadoPorHoras` y el `EmpleadoPorComision` en las líneas 64-65, se produce el mismo resultado que obtener los ingresos de estos empleados en forma individual, en las líneas 24-32. No obstante, el monto de los ingresos obtenidos para el `EmpleadoBaseMasComision` en las líneas 64-65 es más alto que el que se obtiene en las líneas 30-32, debido al aumento del 10% en su salario base.

Las líneas 69-71 imprimen en pantalla el tipo de cada empleado, como un objeto `string`. Todos los objetos en C# conocen su propia clase y pueden acceder a esta información a través del método `GetType`, que todas las clases heredan de la clase `object`. El método `GetType` devuelve un objeto de la clase `Type` (del espacio de nombres `System`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase, los nombres de sus métodos `public` y el nombre de su clase base. La línea 71 invoca al método `GetType` en el objeto para obtener su clase en tiempo de ejecución (es decir, un objeto `Type` que representa el tipo del objeto). Después el método `ToString` se invoca en forma implícita en el objeto devuelto por `GetType`. El método `ToString` de la clase `Type` devuelve el nombre de la clase.

En el ejemplo anterior, evitamos varios errores de compilación mediante la conversión descendente de la variable `Empleado` a una variable `EmpleadoBaseMasComision` en las líneas 55-56. Si eliminamos el operador de conversión (`EmpleadoBaseMasComision`) de la línea 56 y tratamos de asignar la variable `empleadoActual` de `Empleado` directamente a la variable `empleado` de `EmpleadoBaseMasComision`, recibiremos un error de compilación del tipo “No se puede convertir implícitamente al tipo”. Este error indica que el intento de asignar la referencia del objeto `empleadoPorComision` de la clase base a la variable `empleadoBaseMasComision` de la clase derivada no se permite sin un operador de conversión apropiado. El compilador evita esta asignación debido a que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`; de nuevo, la relación *es un* se aplica sólo entre la clase derivada y sus clases base, no viceversa.

De manera similar, si las líneas 58 y 61 utilizan la variable `empleadoActual` de la clase base, en vez de la variable `empleado` de la clase derivada, para usar la propiedad `SalarioBase` que pertenece sólo a la clase derivada, recibiríamos un error de compilación del tipo “'Empleado' no contiene una definición para 'SalarioBase'” en cada una de estas líneas. No se permite tratar de invocar métodos que pertenezcan sólo a la clase derivada en una referencia a la clase base. Mientras que las líneas 58 y 61 se ejecutan sólo si la instrucción `is` en la línea 51 devuelve `true` para indicar que a `empleadoActual` se le asignó una referencia a un objeto `EmpleadoBaseMasComision`, no podemos tratar de usar la propiedad `SalarioBase` de la clase derivada `EmpleadoBaseMasComision` con la referencia `empleadoActual` de la clase base `Empleado`. El compilador generaría errores en las líneas 58 y 61, ya que `SalarioBase` no es miembro de la clase base y no puede utilizarse con una variable de la clase base. Aunque el método que se vaya a llamar en realidad depende del tipo del objeto en tiempo de ejecución, puede

utilizarse una variable para invocar sólo a los métodos que sean miembros del tipo de esa variable, lo cual verifica el compilador. Si utilizamos una variable `Empleado` de la clase base, sólo podemos invocar a los métodos y propiedades que se encuentran en la clase `Empleado`: los métodos `Ingresos` y `ToString`, y las propiedades `PrimerNombre`, `ApellidoPaterno` y `NumeroSeguroSocial`.

11.5.7 Resumen de las asignaciones permitidas entre variables de la clase base y de la clase derivada

Ahora que hemos visto una aplicación completa que procesa diversos objetos de clases derivadas en forma polimórfica, sintetizaremos lo que puede y no puede hacer con los objetos y variables de las clases base y las clases derivadas. Aunque un objeto de una clase derivada también *es un* objeto de su clase base, los dos objetos son, sin embargo, distintos. Como vimos antes, los objetos de una clase derivada pueden tratarse como si fueran objetos de la clase base. No obstante, la clase derivada puede tener miembros adicionales que sólo pertenezcan a esa clase. Por esta razón, no se permite asignar una referencia de clase base a una variable de clase derivada sin una conversión explícita; dicha asignación dejaría los miembros de la clase derivada indefinidos para un objeto de la clase base.

Hemos visto cuatro maneras de asignar referencias de clase base y de clase derivada a las variables de los tipos de clase base y clase derivada:

1. Asignar una referencia de clase base a una variable de clase base es un proceso simple y directo.
2. Asignar una referencia de clase derivada a una variable de clase derivada es un proceso simple y directo.
3. Asignar una referencia de clase derivada a una variable de clase base es seguro, ya que el objeto de la clase derivada *es un* objeto de su clase base. No obstante, esta referencia puede usarse para referirse sólo a los miembros de la clase base. Si este código hace referencia a los miembros que pertenezcan sólo a la clase derivada, a través de la variable de la clase base, el compilador reporta errores.
4. Tratar de asignar una referencia de clase base a una variable de clase derivada es un error de compilación. Para evitar este error, la referencia de clase base debe convertirse en forma explícita a un tipo de clase derivada. En tiempo de ejecución, si el objeto al que se refiere la referencia no es un objeto de la clase derivada, se producirá una excepción. (Para más información sobre el manejo de excepciones, vea el capítulo 12, Manejo de excepciones). El operador `is` puede utilizarse para asegurar que dicha conversión se realice sólo si el objeto es de la clase derivada.

11.6 Métodos y clases `sealed`

En la sección 10.4 vimos que sólo pueden redefinirse en las clases derivadas los métodos que se declaran como `virtual`, `override` o `abstract`. Un método que se declara como **`sealed`** en una clase base no puede redefinirse en una clase derivada. Los métodos que se declaran como `private` son `sealed` de manera implícita, ya que es imposible redefinirlos en una clase derivada (aunque la clase derivada puede declarar un nuevo método con la misma firma que el método `private` en la clase base). Los métodos que se declaran como `static` también son `sealed` de manera implícita, ya que los métodos `static` tampoco pueden redefinirse. Un método de clase derivada que se declare tanto `override` como `sealed` puede redefinir a un método de la clase base, pero no puede redefinirse en clases derivadas que se encuentren a niveles más bajos de la jerarquía de herencia.

La declaración de un método `sealed` no puede cambiar nunca, por lo que todas las clases derivadas utilizan la misma implementación del método, y las llamadas a los métodos `sealed` se resuelven en tiempo de compilación; a esto se le conoce como *vinculación estática*. Como el compilador sabe que los métodos `sealed` no pueden ser redefinidos, a menudo puede optimizar el código eliminando llamadas a los métodos `sealed` y sustituyéndolas con el código expandido de sus declaraciones en cada ubicación de llamada al método; a esta técnica se le conoce como *poner el código en línea*.



Tip de rendimiento 11.1

El compilador puede decidir si va a poner en línea la llamada a un método `sealed`, y lo hará para los métodos `sealed` pequeños y simples. Poner en línea el código de una llamada a un método no viola el encapsulamiento ni el ocultamiento de información y aumenta el rendimiento, ya que elimina la sobrecarga en la que se incurre al hacer una llamada a un método.

Una clase que se declara como `sealed` no puede ser una clase base (es decir, una clase no puede extender a una clase `sealed`). Todos los métodos en una clase `sealed` son `sealed` de manera implícita. La clase `string` es una clase `sealed`. Esta clase no puede extenderse, por lo que las aplicaciones que utilizan objetos `string` pueden depender de la funcionalidad de los objetos `string` que se especifica en la FCL.



Error común de programación 11.5

Tratar de declarar una clase derivada de una clase `sealed` es un error de compilación.



Observación de ingeniería de software 11.5

En la FCL, la vasta mayoría de clases no se declaran como `sealed`. Esto permite el uso de la herencia y el polimorfismo: las herramientas fundamentales de la programación orientada a objetos.

11.7 Caso de estudio: creación y uso de interfaces

En nuestro siguiente ejemplo (figuras 11.11 a 11.15) utilizaremos otra vez el sistema de nómina de la sección 11.5. Suponga que la compañía involucrada desea realizar varias operaciones de contabilidad en una sola aplicación de cuentas por pagar; además de calcular los ingresos de nómina que deben pagarse a cada empleado, la compañía debe también calcular el pago vencido en cada una de varias facturas (por los bienes comprados). Aunque se aplican a cosas no relacionadas (es decir, empleados y facturas), ambas operaciones tienen que ver con el cálculo de algún tipo de monto a pagar. Para un empleado, el pago se refiere a los ingresos del empleado. Para una factura, el pago se refiere al costo total de los bienes listados en la factura. ¿Podemos calcular esas cosas distintas, tales como los pagos vencidos para los empleados y las facturas, en forma polimórfica en una sola aplicación? ¿Ofrece C# una capacidad que requiera que las clases no relacionadas implementen un conjunto de métodos comunes (por ejemplo, un método que calcule un monto a pagar)? Las interfaces de C# ofrecen exactamente esta capacidad.

Las interfaces definen y estandarizan las formas en que pueden interactuar las personas y los sistemas entre sí. Por ejemplo, los controles en un radio sirven como una interfaz entre los usuarios del radio y sus componentes internos. Los controles permiten a los usuarios realizar un conjunto limitado de operaciones (por ejemplo, cambiar la estación, ajustar el volumen, seleccionar AM o FM), y distintos radios pueden implementar los controles de distintas formas (por ejemplo, el uso de botones, perillas, comandos de voz). La interfaz especifica *qué* operaciones debe permitir el radio que realicen los usuarios, pero no especifica *cómo* deben realizarse las operaciones. De manera similar, la interfaz entre un conductor y un auto con transmisión manual incluye el volante, la palanca de cambios, el pedal del embrague, el pedal del acelerador y el pedal del freno. Esta misma interfaz se encuentra en casi todos los autos de transmisión manual, lo que permite que alguien que sabe cómo manejar cierto auto de transmisión manual sepa cómo manejar casi cualquier auto de transmisión manual. Los componentes de cada auto individual pueden tener una apariencia ligeramente distinta, pero el propósito general es el mismo; permitir que las personas conduzcan el auto.

Los objetos de software también se comunican a través de interfaces. Una interfaz de C# describe un conjunto de métodos que pueden llamarse sobre un objeto, para indicar al objeto que realice cierta tarea, o que devuelva cierta pieza de información, por ejemplo. El siguiente ejemplo introduce una interfaz llamada `IPorPagar`, la cual describe la funcionalidad de cualquier objeto que deba ser capaz de recibir un pago y, por lo tanto, debe ofrecer un método para determinar el monto de pago vencido apropiado. La *declaración de una interfaz* empieza con la palabra clave `interface` y sólo puede contener métodos, propiedades, indexadores y eventos (hablaremos sobre los eventos en el capítulo 13, Conceptos de interfaz gráfica de usuario: parte I) abstractos. Todos los miembros de la interfaz se declaran en forma implícita como `public` y `abstract`. Además, cada interfaz puede extender a otra interfaz o más interfaces, para crear una interfaz más elaborada que otras clases puedan implementar.



Error común de programación 11.6

Es un error de compilación declarar un miembro de la interfaz `public` o `abstract` en forma explícita, debido a que es redundante en las declaraciones de miembros de interfaces. También es un error de compilación especificar cualquier detalle de implementación, tal como las declaraciones de métodos concretos, en una interfaz.

Para utilizar una interfaz, una clase debe especificar que *implementa* a esa interfaz mediante un listado de esa interfaz después del signo de dos puntos (:) en la declaración de la clase. Observe que ésta es la misma sintaxis que se utiliza para indicar la herencia de una clase base. Una clase concreta que implementa a esta interfaz debe declarar cada miembro de la interfaz con la firma especificada en la declaración de la interfaz. Una clase que implemente a una interfaz, pero que no implementa a todos los miembros de la misma, es una clase abstracta; debe declararse como `abstract` y debe contener una declaración `abstract` para cada miembro de la interfaz que no implemente. Implementar una interfaz es como firmar un contrato con el compilador que diga, “Proporcionaré una implementación para todos los miembros especificados por la interfaz, o los declararé como `abstract`”.



Error común de programación 11.7

Si no declaramos ningún miembro de una interfaz, en una clase que implemente a esa interfaz, se produce un error de compilación.

Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos. Los programadores pueden crear una interfaz que describa la funcionalidad deseada y después implementar esta interfaz en cualquier clase que requiera esa funcionalidad. Por ejemplo, en la aplicación de cuentas por pagar que desarrollaremos en esta sección, implementamos la interfaz `IPorPagar` en cualquier clase que deba tener la capacidad de calcular el monto de un pago (por ejemplo, `Empleado`, `Factura`).

A menudo, una interfaz se utiliza en vez de una clase `abstract` cuando no hay una implementación predefinida que heredar; esto es, no hay campos ni implementaciones de métodos predeterminadas. Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`, por lo que se declaran en archivos por sí solas con el mismo nombre que la interfaz, y la extensión de archivo `.cs`.

11.7.1 Desarrollo de una jerarquía `IPorPagar`

Para crear una aplicación que pueda determinar los pagos para los empleados y facturas por igual, primero vamos a crear una interfaz llamada `IPorPagar`. Esta interfaz contiene el método `ObtenerMontoPago`, el cual devuelve un monto decimal que debe pagarse para un objeto de cualquier clase que implemente a la interfaz. El método `ObtenerMontoPago` es una versión de propósito general del método `Ingresos` de la jerarquía de `Empleado`; el método `Ingresos` calcula un monto de pago específicamente para un `Empleado`, mientras que `ObtenerMontoPago` puede aplicarse a un amplio rango de objetos no relacionados. Después de declarar la interfaz `IPorPagar` presentaremos la clase `Factura`, la cual implementa a la interfaz `IPorPagar`. Luego modificaremos la clase `Empleado` de tal forma que también implemente a la interfaz `IPorPagar`. Por último, actualizaremos la clase `EmpleadoAsalariado` derivada de `Empleado` para “ajustarla” en la jerarquía de `IPorPagar` (es decir, cambiaremos el nombre del método `Ingresos` de `EmpleadoAsalariado` por el de `ObtenerMontoPago`).



Buena práctica de programación 11.1

Por convención, el nombre de una interfaz empieza con “I”. Esto ayuda a diferenciar las interfaces de las clases, con lo cual se mejora la legibilidad del código.



Buena práctica de programación 11.2

Al declarar un método en una interfaz, seleccione un nombre para el método que describa su propósito en forma general, ya que el método podría implementarse por un amplio rango de clases no relacionadas.

Las clases `Factura` y `Empleado` representan cosas para las cuales la compañía debe calcular un monto a pagar. Ambas clases implementan a `IPorPagar`, por lo que una aplicación puede invocar al método `ObtenerMontoPago` en objetos `Factura` y `Empleado` por igual. Esto permite el procesamiento polimórfico de objetos `Factura` y `Empleado` requerido para la aplicación de cuentas por pagar de nuestra compañía.

El diagrama de clases de UML en la figura 11.10 muestra la jerarquía de interfaz y clases utilizada en nuestra aplicación de cuentas por pagar. La jerarquía comienza con la interfaz `IPorPagar`. UML diferencia a una interfaz de una clase colocando la palabra “interface” entre los signos « y », por encima del nombre de la interfaz. UML

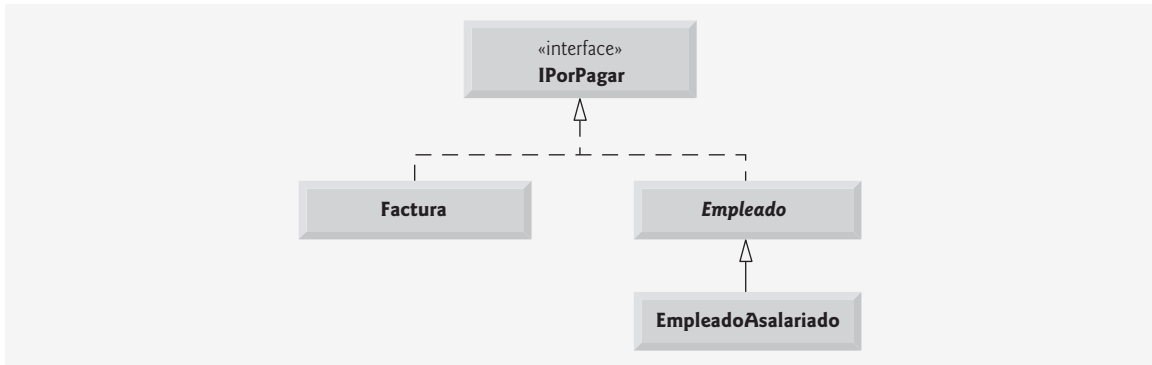


Figura 11.10 | Diagrama de clases de UML de la jerarquía de clases y la interfaz IPorPagar.

```

1 // Fig. 11.11: IPorPagar.cs
2 // Declaración de la interfaz IPorPagar.
3 public interface IPorPagar
4 {
5     decimal ObtenerMontoPago(); // calcula el pago; no hay implementación
6 } // fin de la interfaz IPorPagar
  
```

Figura 11.11 | Declaración de la interfaz IPorPagar.

expresa la relación entre una clase y una interfaz a través de una *realización*. Se dice que una clase “realiza”, o implementa, a una interfaz. Un diagrama de clases modela una realización como una flecha punteada que parte de la clase que va a realizar la implementación, hasta la interfaz. El diagrama en la figura 11.10 indica que cada una de las clases *Factura* y *Empleado* realizan (es decir, implementan) la interfaz *IPorPagar*. Observe que, al igual que en el diagrama de clases de la figura 11.2, la clase *Empleado* aparece en cursivas, lo cual indica que es una clase abstracta. La clase concreta *EmpleadoAsalariado* extiende a *Empleado* y hereda la relación de realización de la clase base con la interfaz *IPorPagar*.

11.7.2 Declaración de la interfaz IPorPagar

La declaración de la interfaz *IPorPagar* empieza en la figura 11.11, línea 3. La interfaz *IPorPagar* contiene el método `public abstract ObtenerMontoPago()` (línea 5). Recuerde que este método no puede declararse en forma explícita como `public` o `abstract`. La interfaz *IPorPagar* sólo tiene un método, pero las interfaces pueden tener cualquier número de miembros. Además, el método *ObtenerMontoPago* no tiene parámetros, pero los métodos de las interfaces pueden tener parámetros.

11.7.3 Creación de la clase Factura

Ahora vamos a crear la clase *Factura* (figura 11.12) para representar una factura simple que contiene información de facturación para cierto tipo de pieza. La clase declara las variables de instancia `private numeroPieza`, `descripcionPieza`, `cantidad` y `precioPorArticulo` (líneas 5-8), las cuales indican el número de pieza, su descripción, la cantidad de piezas ordenadas y el precio por artículo. La clase *Factura* también contiene un constructor (líneas 11-18), propiedades (líneas 21-70) que manipulan las variables de instancia de la clase y un método *ToString* (líneas 73-79) que devuelve una representación `string` de un objeto *Factura*. Observe que los descriptores de acceso `set` de las propiedades *Cantidad* (líneas 53-56) y *PrecioPorArticulo* (líneas 66-69) aseguran que a *cantidad* y a *precioPorArticulo* se les asignen sólo valores no negativos.

La línea 3 de la figura 11.12 indica que la clase *Factura* implementa a la interfaz *IPorPagar*. Al igual que todas las clases, la clase *Factura* también extiende a *object* de manera implícita. *C#* no permite que las clases derivadas hereden de más de una clase base, pero sí permite que una clase herede de una clase base e implemente cualquier número de interfaces. Todos los objetos de una clase que implementan varias interfaces tienen la

relación *es un* con cada tipo de interfaz implementada. Para implementar más de una interfaz, utilice una lista separada por comas de nombres de interfaz después del signo de dos puntos (:) en la declaración de la clase, como se muestra a continuación:

```
public class NombreClase : NombreClaseBase, PrimeraInterfaz, SegundaInterfaz, ...
```

Cuando una clase hereda de una clase base e implementa a una o más interfaces, la declaración de la clase debe listar el nombre de la clase base antes de cualquier nombre de interfaz.

```
1  // Fig. 11.12: Factura.cs
2  // La clase Factura implementa a IPorPagar.
3  public class Factura : IPorPagar
4  {
5      private string numeroPieza;
6      private string descripcionPieza;
7      private int cantidad;
8      private decimal precioPorArticulo;
9
10     // constructor con cuatro parámetros
11     public Factura( string pieza, string descripcion, int cuenta,
12                  decimal precio )
13     {
14         NumeroPieza = pieza;
15         DescripcionPieza = descripcion;
16         Cantidad = cuenta; // valida la cantidad a través de una propiedad
17         PrecioPorArticulo = precio; // valida el precio por artículo a través de una
18                                     // propiedad
19     } // fin del constructor de Factura con cuatro parámetros
20
21     // propiedad que obtiene y establece el número de pieza en la factura
22     public string NumeroPieza
23     {
24         get
25         {
26             return numeroPieza;
27         } // fin de get
28         set
29         {
30             numeroPieza = value; // debería validarse
31         } // fin de set
32     } // fin de propiedad NumeroPieza
33
34     // propiedad que obtiene y establece la descripción de la pieza en la factura
35     public string DescripcionPieza
36     {
37         get
38         {
39             return descripcionPieza;
40         } // fin de get
41         set
42         {
43             descripcionPieza = value; // debería validarse
44         } // fin de set
45     } // fin de la propiedad DescripcionPieza
46
47     // propiedad que obtiene y establece la cantidad en la factura
48     public int Cantidad
```

Figura 11.12 | La clase Factura implementa a IPorPagar. (Parte 1 de 2).

```

48     {
49         get
50         {
51             return cantidad;
52         } // fin de get
53         set
54         {
55             cantidad = ( value < 0 ) ? 0 : value; // valida la cantidad
56         } // fin de set
57     } // fin de la propiedad Cantidad
58
59     // propiedad que obtiene y establece el precio por artículo
60     public decimal PrecioPorArticulo
61     {
62         get
63         {
64             return precioPorArticulo;
65         } // fin de get
66         set
67         {
68             precioPorArticulo = ( value < 0 ) ? 0 : value; // valida el precio
69         } // fin de set
70     } // fin de la propiedad PrecioPorArticulo
71
72     // devuelve representación string de objeto Factura
73     public override string ToString()
74     {
75         return string.Format(
76             "{0}: \n{1}: {2} ({3}) \n{4}: {5} \n{6}: {7:C}",
77             "factura", "número de pieza", NumeroPieza, DescripcionPieza,
78             "cantidad", Cantidad, "precio por artículo", PrecioPorArticulo );
79     } // fin del método ToString
80
81     // método requerido para llevar a cabo el contrato con la interfaz IPorPagar
82     public decimal ObtenerMontoPago()
83     {
84         return Cantidad * PrecioPorArticulo; // calcula el costo total
85     } // fin del método ObtenerMontoPago
86 } // fin de la clase Factura

```

Figura 11.12 | La clase Factura implementa a IPorPagar. (Parte 2 de 2).

La clase Factura implementa el único método de la interfaz IPorPagar; el método ObtenerMontoPago se declara en las líneas 82-85. Este método calcula el monto requerido para pagar la factura. El método multiplica los valores de cantidad y precioPorArticulo (que se obtienen a través de las propiedades apropiadas) y devuelve el resultado (línea 84). Este método cumple con el requerimiento de implementación para el método en la interfaz IPorPagar; hemos cumplido el contrato de interfaz con el compilador.

11.7.4 Modificación de la clase Empleado para implementar la interfaz IPorPagar

Ahora vamos a modificar la clase Empleado para que implemente la interfaz IPorPagar. La figura 11.13 contiene la clase Empleado modificada. Esta declaración de la clase es idéntica a la de la figura 11.4, con dos excepciones. En primer lugar, la línea 3 de la figura 11.13 indica que la clase Empleado ahora implementa a la interfaz IPorPagar. En segundo lugar, como Empleado ahora implementa a la interfaz IPorPagar, debemos cambiar el nombre de Ingresos por el de ObtenerMontoPago en toda la jerarquía de Empleado. Sin embargo, al igual que con el método Ingresos en la versión de la clase Empleado de la figura 11.4, no tiene sentido implementar el método ObtenerMontoPago en la clase Empleado, ya que no podemos calcular el pago de los ingresos para un Empleado general; primero debemos conocer el tipo específico de Empleado. En la figura 11.4 declaramos el método

Ingresos como `abstract` por esta razón y, como resultado, la clase `Empleado` tuvo que declararse como `abstract`. Esto obliga a cada clase derivada de `Empleado` a redefinir `Ingresos` con una implementación concreta.

En la figura 11.13, manejamos esta situación de la misma forma. Recuerde que cuando una clase implementa a una interfaz, la clase hace un contrato con el compilador, en el que se establece que la clase va a implementar cada uno de los métodos en la interfaz, o los va a declarar como `abstract`. Si se elige la última opción, también debemos declarar la clase como `abstract`. Como vimos en la sección 11.4, cualquier clase derivada concreta de la clase abstracta debe implementar a los métodos `abstract` de la clase base. Si la clase derivada no lo hace, también debe declararse como `abstract`. Como lo indican los comentarios en las líneas 51-52, la clase `Empleado` de la figura 11.13 no implementa al método `ObtenerMontoPago`, por lo que la clase se declara como `abstract`.

```

1  // Fig. 11.13: Empleado.cs
2  // La clase base abstracta Empleado.
3  public abstract class Empleado : IPorPagar
4  {
5      private string primerNombre;
6      private string apellidoPaterno;
7      private string numeroSeguroSocial;
8
9      // constructor con tres parámetros
10     public Empleado( string nombre, string apellido, string nss )
11     {
12         primerNombre = nombre;
13         apellidoPaterno = apellido;
14         numeroSeguroSocial = nss;
15     } // fin del constructor de Empleado con tres parámetros
16
17     // propiedad de sólo lectura que obtiene el primer nombre del empleado
18     public string PrimerNombre
19     {
20         get
21         {
22             return primerNombre;
23         } // fin de get
24     } // fin de la propiedad PrimerNombre
25
26     // propiedad de sólo lectura que obtiene el apellido paterno del empleado
27     public string ApellidoPaterno
28     {
29         get
30         {
31             return apellidoPaterno;
32         } // fin de get
33     } // fin de la propiedad ApellidoPaterno
34
35     // propiedad de sólo lectura que obtiene el número de seguro social del empleado
36     public string NumeroSeguroSocial
37     {
38         get
39         {
40             return numeroSeguroSocial;
41         } // fin de get
42     } // fin de la propiedad NumeroSeguroSocial
43
44     // devuelve representación string del objeto Empleado
45     public override string ToString()

```

Figura 11.13 | La clase base abstracta `Empleado`. (Parte 1 de 2).

```

46     {
47         return string.Format( "{0} {1}\nnúmero de seguro social: {2}",
48             PrimerNombre, ApellidoPaterno, NumeroSeguroSocial );
49     } // fin del método ToString
50
51     // Nota: No implementamos aquí el método ObtenerMontoPago de IPorPagar, por lo
52     // que esta clase debe declararse como abstracta para evitar un error de compilación.
53     public abstract decimal ObtenerMontoPago();
54 } // fin de la clase abstracta Empleado

```

Figura 11.13 | La clase base abstracta Empleado. (Parte 2 de 2).

11.7.5 Modificación de la clase EmpleadoAsalariado para usarla en la jerarquía IPorPagar

La figura 11.14 contiene una versión modificada de la clase EmpleadoAsalariado que extiende a Empleado e implementa el método ObtenerMontoPago. Esta versión de EmpleadoAsalariado es idéntica a la de la figura 11.5, con la excepción de que esta versión implementa al método ObtenerMontoPago (líneas 29-32) en vez del método Ingresos. Los dos métodos contienen la misma funcionalidad, pero tienen distintos nombres. Recuerde que la versión de IPorPagar del método tiene un nombre más general para que pueda aplicarse a clases que sean posiblemente dispares. El resto de las clases derivadas de Empleado (EmpleadoPorHoras, EmpleadoPorComision y EmpleadoBaseMasComision) también deben modificarse para que contengan el método ObtenerMontoPago en vez de Ingresos, y así reflejar el hecho de que ahora Empleado implementa a IPorPagar. Dejaremos estas modificaciones como un ejercicio y sólo utilizaremos a EmpleadoAsalariado en nuestra aplicación de prueba en esta sección.

```

1  // Fig. 11.14: EmpleadoAsalariado.cs
2  // La clase EmpleadoAsalariado que extiende a Empleado.
3  public class EmpleadoAsalariado : Empleado
4  {
5      private decimal salarioSemanal;
6
7      // constructor con cuatro parámetros
8      public EmpleadoAsalariado( string nombre, string apellido, string nss,
9          decimal salario ) : base( nombre, apellido, nss )
10     {
11         SalarioSemanal = salario; // valida el salario a través de una propiedad
12     } // fin del constructor de EmpleadoAsalariado con cuatro parámetros
13
14     // propiedad que obtiene y establece el salario del empleado asalariado
15     public decimal SalarioSemanal
16     {
17         get
18         {
19             return salarioSemanal;
20         } // fin de get
21         set
22         {
23             salarioSemanal = value < 0 ? 0 : value; // validación
24         } // fin de set
25     } // fin de la propiedad SalarioSemanal
26
27     // calcula los ingresos; implementa el método de la interfaz
28     // IPorPagar que es abstracto en la clase base Empleado

```

Figura 11.14 | La clase EmpleadoAsalariado que extiende a Empleado. (Parte 1 de 2).

```

29  public override decimal ObtenerMontoPago()
30  {
31      return SalarioSemanal;
32  } // fin del método ObtenerMontoPago
33
34  // devuelve representación string del objeto EmpleadoAsalariado
35  public override string ToString()
36  {
37      return string.Format( "empleado asalariado: {0}\n{1}: {2:C}",
38                          base.ToString(), "salario semanal", SalarioSemanal );
39  } // fin del método ToString
40  } // fin de la clase EmpleadoAsalariado

```

Figura 11.14 | La clase `EmpleadoAsalariado` que extiende a `Empleado`. (Parte 2 de 2).

Cuando una clase implementa a una interfaz, se aplica la misma relación *es un* que proporciona la herencia. Por ejemplo, la clase `Empleado` implementa a `IPorPagar`, por lo que podemos decir que un `Empleado` es un `IPorPagar`, al igual que cualquier clase que extienda a `Empleado`. Por ejemplo, los objetos `EmpleadoAsalariado` son objetos `IPorPagar`. Al igual que con las relaciones de herencia, un objeto de una clase que implemente a una interfaz puede considerarse como un objeto del tipo de la interfaz. Los objetos de cualquier clase derivada de la clase que implementa a la interfaz también pueden considerarse como objetos del tipo de la interfaz. Por ende, así como podemos asignar la referencia de un objeto `EmpleadoAsalariado` a una variable de la clase base `Empleado`, también podemos asignar la referencia de un objeto `EmpleadoAsalariado` a una interfaz `IPorPagar`. `Factura` implementa a `IPorPagar`, por lo que un objeto `Factura` también *es un* objeto `IPorPagar`, y podemos asignar la referencia de un objeto `Factura` a una variable `IPorPagar`.



Observación de ingeniería de software 11.6

La herencia y las interfaces son similares en cuanto a su implementación de la relación es un. Un objeto de una clase que implementa a una interfaz puede considerarse como un objeto del tipo de esa interfaz. Un objeto de cualquier clase derivada de una clase que implementa a una interfaz también puede considerarse como un objeto del tipo de la interfaz.



Observación de ingeniería de software 11.7

La relación es un que existe entre las clases base y las clases derivadas, y entre las interfaces y las clases que las implementan, se mantiene cuando se pasa un objeto a un método. Cuando el parámetro de un método recibe una variable de una clase base o de un tipo de interfaz, el método procesa en forma polimórfica al objeto que recibe como argumento.

11.7.6 Uso de la interfaz `IPorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo

`PruebaInterfazPorPagar` (figura 11.15) ilustra que la interfaz `IPorPagar` puede usarse para procesar un conjunto de objetos `Factura` y `Empleado` en forma polimórfica en una sola aplicación. La línea 10 declara a `objetosPorPagar` y le asigna un arreglo de cuatro variables `IPorPagar`. Las líneas 13-14 asignan las referencias de objetos `EmpleadoAsalariado` a los dos elementos restantes de `objetosPorPagar`. Las líneas 15-18 asignan las referencias a los objetos `EmpleadoSalario` a los dos elementos restantes de `objetosPorPagar`. Estas asignaciones se permiten debido a que un objeto `Factura` *es un* objeto `IPorPagar`. Las líneas 24-29 utilizan una instrucción `foreach` para procesar cada objeto `IPorPagar` en `objetosPorPagar` de manera polimórfica, imprimiendo en pantalla el objeto como un `string`, junto con el pago vencido. Observe que la línea 27 invoca en forma implícita al método `ToString` de una referencia de la interfaz `IPorPagar`, aun cuando `ToString` no se declara en la interfaz `IPorPagar`; todas las referencias (incluyendo las de los tipos de interfaces) se refieren a objetos que extienden a `object` y, por lo tanto, tienen un método `ToString`. La línea 28 invoca al método `ObtenerMontoPago` de `IPorPagar` para obtener

el monto a pagar para cada objeto en `objetosPorPagar`, sin importar el tipo del objeto. Los resultados revelan que las llamadas a los métodos en las líneas 27-28 invocan a la implementación de la clase apropiada de los métodos `ToString` y `ObtenerMontoPago`. Por ejemplo, cuando `empleadoActual` se refiere a un objeto `Factura` durante la primera iteración del ciclo `foreach`, se ejecutan los métodos `ToString` y `ObtenerMontoPago` de la clase `Factura`.



Observación de ingeniería de software 11.8

Todos los métodos de la clase `object` pueden llamarse mediante el uso de una referencia de un tipo de interfaz; la referencia se refiere a un objeto, y todos los objetos heredan los métodos de la clase `object`.

```

1  // Fig. 11.15: PruebaInterfazPorPagar.cs
2  // Prueba la interfaz IPorPagar con clases dispares.
3  using System;
4
5  public class PruebaInterfazPorPagar
6  {
7      public static void Main( string[] args )
8      {
9          // crea arreglo IPorPagar de cuatro elementos
10         IPorPagar[] objetosPorPagar = new IPorPagar[ 4 ];
11
12         // llena el arreglo con objetos que implementan a IPorPagar
13         objetosPorPagar[ 0 ] = new Factura( "01234", "asiento", 2, 375.00M );
14         objetosPorPagar[ 1 ] = new Factura( "56789", "llanta", 4, 79.95M );
15         objetosPorPagar[ 2 ] = new EmpleadoAsalariado( "John", "Smith",
16             "111-11-1111", 800.00M );
17         objetosPorPagar[ 3 ] = new EmpleadoAsalariado( "Lisa", "Barnes",
18             "888-88-8888", 1200.00M );
19
20         Console.WriteLine(
21             "Facturas y Empleados procesados en forma polimórfica:\n" );
22
23         // procesa en forma genérica cada elemento en el arreglo objetosPorPagar
24         foreach ( IPorPagar porPagarActual in objetosPorPagar )
25         {
26             // imprime en pantalla el valor de porPagarActual y el monto a pagar apropiado
27             Console.WriteLine( "{0} \n{1}: {2:C}\n", porPagarActual,
28                 "pago vencido", porPagarActual.ObtenerMontoPago() );
29         } // fin de foreach
30     } // fin de Main
31 } // fin de la clase PruebaInterfazPorPagar

```

Facturas y Empleados procesados en forma polimórfica:

factura:
 número de pieza: 01234 (asiento)
 cantidad: 2
 precio por artículo: \$375.00
 pago vencido: \$750.00

factura:
 número de pieza: 56789 (llanta)
 cantidad: 4
 precio por artículo: \$79.95
 pago vencido: \$319.80

Figura 11.15 | Prueba de la interfaz `IPorPagar` con clases dispares. (Parte 1 de 2).

```
empleado asalariado: John Smith
número de seguro social: 111-11-1111
salario semanal: $800.00
pago vencido: $800.00

empleado asalariado: Lisa Barnes
número de seguro social: 888-88-8888
salario semanal: $1,200.00
pago vencido: $1,200.00
```

Figura 11.15 | Prueba de la interfaz IPorPagar con clases dispares. (Parte 2 de 2).

11.7.7 Interfaces comunes de la Biblioteca de clases del .NET Framework

En esta sección veremos las generalidades acerca de varias interfaces comunes en la Biblioteca de clases del .NET Framework. Estas interfaces se implementan y se utilizan de la misma forma que la interfaz que creamos en secciones anteriores (la interfaz IPorPagar en la sección 11.7.2). Las interfaces de la FCL le permiten extender muchos aspectos importantes de C# con sus propias clases. La figura 11.16 muestra los aspectos generales de varias interfaces de uso común de la FCL.

11.8 Sobrecarga de operadores

Las manipulaciones en los objetos de las clases se realizan mediante el envío de mensajes (en forma de llamadas a métodos) a los objetos. Esta notación de llamadas a métodos es incómoda para ciertos tipos de clases, en especial las clases matemáticas. Para estas clases, sería conveniente utilizar el robusto conjunto de operadores integrados de C# para especificar las manipulaciones de objetos. En esta sección le mostraremos cómo permitir que estos operadores trabajen con objetos de clases, a través de un proceso conocido como *sobrecarga de operadores*.

Interface	Descripción
IComparable	Como vio en el capítulo 3, C# contiene varios operadores de comparación (por ejemplo, <, <=, >, >=, ==, !=) que le permiten comparar valores de tipos simples. En la sección 11.8 verá que estos operadores se pueden definir para comparar dos objetos. La interfaz IComparable también puede utilizarse para permitir que los objetos de una clase que implementa a la interfaz se comparen unos con otros. La interfaz contiene un método, CompareTo, que compara el objeto que llama al método con el objeto que se pasa como argumento al método. Las clases deben implementar a CompareTo para devolver un valor que indica si el objeto en el cual se invoca es menor que (valor de retorno entero negativo), igual que (valor de retorno 0) o mayor que (valor de retorno entero positivo) el objeto que se pasa como argumento, usando cualquier criterio especificado por el programador. Por ejemplo, si la clase Empleado implementa a IComparable, su método CompareTo podría comparar objetos Empleado en base a sus montos de ingresos. La interfaz IComparable se utiliza comúnmente para ordenar objetos en una colección tal como un arreglo. En el capítulo 25, Genéricos y en el capítulo 26, Colecciones, utilizaremos la interfaz IComparable.
IComponent	Cualquier clase que represente un componente puede implementar a esta interfaz, incluyendo los controles (tales como botones o etiquetas) de la Interfaz Gráfica de Usuario (GUI). La interfaz IComponent define los comportamientos que deben implementar los componentes. En el capítulo 13, Conceptos de interfaz gráfica de usuario: parte 1, y en el capítulo 14, Conceptos de interfaz gráfica de usuario: parte 2, hablaremos sobre IComponent y muchos controles de GUI que implementan a esta interfaz.

Figura 11.16 | Interfaces comunes de la Biblioteca de clases del .NET Framework. (Parte 1 de 2).

Interface	Descripción
IDisposable	Las clases que deben proporcionar un mecanismo explícito para liberar recursos pueden implementar a esta interfaz. Algunos recursos pueden ser utilizados sólo por un programa a la vez. Además, algunos recursos tales como los archivos en disco son recursos no administrados que, a diferencia de la memoria, no pueden ser liberados por el recolector de basura. Las clases que implementan a la interfaz IDisposable proporcionan un método Dispose que puede llamarse para liberar recursos en forma explícita. En el capítulo 12, Manejo de excepciones, hablaremos un poco acerca de IDisposable. Puede aprender más acerca de esta interfaz en msdn2.microsoft.com/en-us/library/aax125c9 (inglés), o en msdn2.microsoft.com/es-es/library/system.idisposable(VS.80).aspx (español). El artículo de MSDN <i>Implementar un método Dispose</i> en msdn2.microsoft.com/es-es/library/fs2xkftw(VS.80).aspx habla sobre cómo implementar en forma apropiada esta interfaz en sus clases.
IEnumerator	Se utiliza para iterar a través de los elementos de una colección (como un arreglo), un elemento a la vez. La interfaz IEnumerator contiene el método MoveNext para desplazarse al siguiente elemento en una colección, el método Reset para desplazarse a la posición anterior al primer elemento, y la propiedad Current para devolver el objeto en la ubicación actual. En el capítulo 26, Colecciones, utilizaremos a IEnumerator.

Figura 11.16 | Interfaces comunes de la Biblioteca de clases del .NET Framework. (Parte 2 de 2).



Observación de ingeniería de software 11.9

Use la sobrecarga de operadores cuando ésta haga que la aplicación sea más clara que lograr las mismas operaciones con llamadas explícitas a métodos.

C# le permite sobrecargar la mayoría de los operadores para hacerlos sensibles al contexto en el que se utilizan. Algunos operadores se sobrecargan con frecuencia, en especial varios operadores aritméticos tales como + y -. El trabajo realizado por los operadores sobrecargados también puede realizarse mediante llamadas explícitas a métodos, pero la notación de los operadores es por lo común más natural. Las figuras 11.17 y 11.18 muestran un ejemplo del uso de la sobrecarga de operadores, con una clase `NumeroComplejo`.

La clase `NumeroComplejo` (figura 11.17) sobrecarga los operadores de suma (+), resta (-) y multiplicación (*) para permitir a los programas sumar, restar y multiplicar instancias de la clase `NumeroComplejo` mediante el uso de la notación matemática común. Las líneas 8-9 declaran variables de instancia para las partes real e imaginaria del número complejo.

```

1  // Fig. 11.17: NumeroComplejo.cs
2  // Clase que sobrecarga operadores para sumar, restar
3  // y multiplicar números complejos.
4  using System;
5
6  public class NumeroComplejo
7  {
8      private double real; // componente real del número complejo
9      private double imaginario; // componente imaginario del número complejo
10
11     // constructor
12     public NumeroComplejo( double a, double b )
13     {
14         real = a;
15         imaginario = b;

```

Figura 11.17 | Clase que sobrecarga operadores para sumar, restar y multiplicar números complejos. (Parte 1 de 2).


```

16     } // fin del constructor
17
18     // devuelve representación string de NumeroComplejo
19     public override string ToString()
20     {
21         return string.Format( "{0} {1} {2}i",
22             Real, ( Imaginario < 0 ? "-" : "+" ), Math.Abs( Imaginario ) );
23     } // fin del método ToString
24
25     // propiedad de sólo lectura que obtiene el componente real
26     public double Real
27     {
28         get
29         {
30             return real;
31         } // fin de get
32     } // fin de la propiedad Real
33
34     // propiedad de sólo lectura que obtiene el componente imaginario
35     public double Imaginario
36     {
37         get
38         {
39             return imaginario;
40         } // fin de get
41     } // fin de la propiedad Imaginario
42
43     // sobrecarga el operador de suma
44     public static NumeroComplejo operator+(
45         NumeroComplejo x, NumeroComplejo y )
46     {
47         return new NumeroComplejo( x.Real + y.Real,
48             x.Imaginario + y.Imaginario );
49     } // fin del operador +
50
51     // sobrecarga el operador de resta
52     public static NumeroComplejo operator-(
53         NumeroComplejo x, NumeroComplejo y )
54     {
55         return new NumeroComplejo( x.Real - y.Real,
56             x.Imaginario - y.Imaginario );
57     } // fin del operador -
58
59     // sobrecarga el operador de multiplicación
60     public static NumeroComplejo operator*(
61         NumeroComplejo x, NumeroComplejo y )
62     {
63         return new NumeroComplejo(
64             x.Real * y.Real - x.Imaginario * y.Imaginario,
65             x.Real * y.Imaginario + y.Real * x.Imaginario );
66     } // fin del operador *
67 } // fin de la clase NumeroComplejo

```

Figura 11.17 | Clase que sobrecarga operadores para sumar, restar y multiplicar números complejos. (Parte 2 de 2).

Las líneas 44-49 sobrecargan el operador de suma (+) para realizar la suma de objetos `NumeroComplejo`. La palabra clave **operator**, seguida de un símbolo de operador, indica que un método sobrecarga el operador especificado. Los métodos que sobrecargan operadores binarios deben recibir dos argumentos. El primer argumento

es el operando izquierdo, y el segundo argumento es el operando derecho. El operador de suma sobrecargado de la clase `NumeroComplejo` recibe dos referencias `NumeroComplejo` como argumentos y devuelve un `NumeroComplejo` que representa la suma de los argumentos. Observe que este método se marca como `public` y `static`, obligatorio para los operadores sobrecargados. El cuerpo del método (líneas 47-48) realiza la suma y devuelve el resultado como un nuevo `NumeroComplejo`. Observe que no modificamos el contenido de ninguno de los operandos originales que se pasan como los argumentos `x` y `y`. Esto concuerda con nuestro sentido intuitivo de cómo debe comportarse este operador: al sumar dos números no se modifica ninguno de los números originales. Las líneas 52-66 proporcionan operadores sobrecargados similares para restar y multiplicar objetos `NumeroComplejo`.



Observación de ingeniería de software 11.10

Sobrecargue operadores para realizar la misma función o funciones similares en los objetos de clases que los operadores realizan en objetos de tipos simples. Evite el uso no intuitivo de los operadores.



Observación de ingeniería de software 11.11

Al menos un argumento del método de un operador sobrecargado debe ser una referencia a un objeto de la clase en la que se sobrecarga el operador. Esto evita que los programadores cambien la forma en que trabajan los operadores con los tipos simples.

La clase `PruebaComplejo` (figura 11.18) demuestra el uso de los operadores sobrecargados para sumar, restar y multiplicar objetos `NumeroComplejo`. Las líneas 14-27 piden al usuario que introduzca dos números complejos, y después utilizan esta entrada para crear dos objetos `NumeroComplejo` y asignarlos a las variables `x` y `y`.

```

1  // Fig 11.18: PruebaComplejo.cs
2  // Sobrecarga de operadores para números complejos.
3  using System;
4
5  public class PruebaComplejo
6  {
7      public static void Main( string[] args )
8      {
9          // declara dos variables para almacenar números complejos
10         // que introduce el usuario
11         NumeroComplejo x, y;
12
13         // pide al usuario que introduzca el primer número complejo
14         Console.Write( "Escriba la parte real del número complejo x: " );
15         double parteReal = Convert.ToDouble( Console.ReadLine() );
16         Console.Write(
17             "Escriba la parte imaginaria del número complejo x: " );
18         double parteImaginaria = Convert.ToDouble( Console.ReadLine() );
19         x = new NumeroComplejo( parteReal, parteImaginaria );
20
21         // pide al usuario que introduzca el segundo número complejo
22         Console.Write( "\nEscriba la parte real del número complejo y: " );
23         parteReal = Convert.ToDouble( Console.ReadLine() );
24         Console.Write(
25             "Escriba la parte imaginaria del número complejo y: " );
26         parteImaginaria = Convert.ToDouble( Console.ReadLine() );
27         y = new NumeroComplejo( parteReal, parteImaginaria );
28
29         // muestra en pantalla los resultados de los cálculos con x y y
30         Console.WriteLine();
31         Console.WriteLine( "{0} + {1} = {2}", x, y, (x + y).ToString() );

```

Figura 11.18 | Sobrecarga de operadores para números complejos. (Parte I de 2).

```

32     Console.WriteLine( "{0} - {1} = {2}", x, y, x - y );
33     Console.WriteLine( "{0} * {1} = {2}", x, y, x * y );
34 } // fin del método Main
35 } // fin de la clase PruebaComplejo

```

Escriba la parte real del número complejo x: 2
 Escriba la parte imaginaria del número complejo x: 4

 Escriba la parte real del número complejo y: 4
 Escriba la parte imaginaria del número complejo y: -2

 $(2 + 4i) + (4 - 2i) = (6 + 2i)$
 $(2 + 4i) - (4 - 2i) = (-2 + 6i)$
 $(2 + 4i) * (4 - 2i) = (16 + 12i)$

Figura 11.18 | Sobrecarga de operadores para números complejos. (Parte 2 de 2).

Las líneas 31-33 suman, restan y multiplican x y y con los operadores sobrecargados, y después imprimen los resultados en pantalla. En la línea 31, para realizar la suma utilizamos el operador de suma con los operandos `NumeroComplejo` x y y . Sin la sobrecarga de operadores, la expresión $x + y$ no tendría sentido; el compilador no sabría cómo deberían sumarse los dos objetos. Esta expresión tiene sentido aquí debido a que hemos definido el operador de suma para dos objetos `NumeroComplejo` en las líneas 44-49 de la figura 11.17. Cuando se “suman” los dos objetos `NumeroComplejo` en la línea 31 de la figura 11.18, se invoca la declaración de `operator+`, se pasa el operando izquierdo como el primer argumento y el operando derecho como el segundo argumento. Cuando utilizamos los operadores de resta y multiplicación en las líneas 32-33, se invocan de manera similar sus respectivas declaraciones de operadores sobrecargados.

Observe que el resultado de cada cálculo es una referencia a un nuevo objeto `NumeroComplejo`. Cuando se pasa este nuevo objeto al método `WriteLine` de la clase `Console`, se invoca en forma implícita a su método `ToString` (líneas 19-23 de la figura 11.17). No necesitamos asignar un objeto a una variable de tipo de referencia para invocar a su método `ToString`. La línea 31 de la figura 11.18 podría reescribirse para invocar en forma explícita al método `ToString` del objeto creado por el operador de suma sobrecargado, como en:

```
Console.WriteLine( "{0} + {1} = {2}", x, y, ( x + y ).ToString() );
```

11.9 (Opcional) Caso de estudio de ingeniería de software: incorporación de la herencia y el polimorfismo en el sistema ATM

Ahora regresaremos a nuestro diseño del sistema ATM para ver cómo podría beneficiarse de la herencia y el polimorfismo. Para aplicar la herencia, primero buscamos características comunes entre las clases del sistema. Creamos una jerarquía de herencia para modelar las clases similares en una forma elegante y eficiente que nos permita procesar objetos de estas clases mediante el polimorfismo. Después modificamos nuestro diagrama de clases para incorporar las nuevas relaciones de herencia. Por último, demostramos cómo traducir los aspectos relacionados con la herencia de nuestro diseño actualizado, a código de C#.

En la sección 4.11 nos topamos con el problema de representar una transacción financiera en el sistema. En vez de crear una clase para representar a todos los tipos de transacciones, creamos tres clases distintas de transacciones (`SolicitudSaldo`, `Retiro` y `Deposito`) para representar las transacciones que puede realizar el sistema ATM. El diagrama de clases de la figura 11.19 muestra los atributos y operaciones de estas clases. Observe que tienen un atributo privado (`numeroCuenta`) y una operación pública (`Ejecutar`) en común. Cada clase requiere que el atributo `numeroCuenta` especifique la cuenta a la que se aplica la transacción.

Cada clase contiene la operación `Ejecutar()`, que el ATM invoca para realizar la transacción. Es evidente que `SolicitudSaldo`, `Retiro` y `Deposito` representan *tipos de transacciones*. La figura 11.19 revela las características comunes entre las clases de transacciones, por lo que el uso de la herencia para factorizar las características comunes parece apropiado para diseñar estas clases. Colocamos la funcionalidad común en la clase base `Transaccion` y derivamos las clases `SolicitudSaldo`, `Retiro` y `Deposito` de `Transaccion` (figura 11.20).

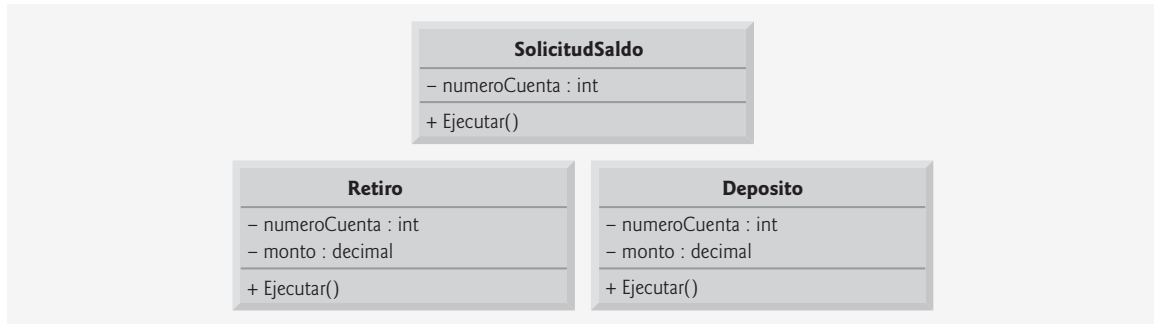


Figura 11.19 | Atributos y operaciones de las clases `SolicitudoSaldo`, `Retiro` y `Deposito`.

UML especifica una relación conocida como *generalización* para modelar la herencia. La figura 11.20 es el diagrama de clases que modela la relación de herencia entre la clase base `Transaccion` y sus tres clases derivadas. Las flechas con puntas triangulares huecas indican que las clases `SolicitudoSaldo`, `Retiro` y `Deposito` se derivan de la clase `Transaccion` por herencia. Se dice que la clase `Transaccion` es una generalización de sus clases derivadas. Se dice que las clases derivadas son *especializaciones* de la clase `Transaccion`.

Como se muestra en la figura 11.19, las clases `SolicitudoSaldo`, `Retiro` y `Deposito` comparten el atributo privado `int numeroCuenta`. Nos gustaría factorizar este atributo común y colocarlo en la clase base `Transaccion`. Sin embargo, recuerde que los atributos privados de una clase base no son accesibles en las clases derivadas. Las clases derivadas de `Transaccion` requieren acceso al atributo `numeroCuenta`, para poder especificar cuál Cuenta se va a procesar en la `BaseDatosBanco`. Como vimos en el capítulo 10, una clase derivada sólo puede tener acceso a los miembros `public`, `protected` o `internal` de su clase base. No obstante, las clases derivadas en este caso no necesitan modificar el atributo `numeroCuenta`; sólo necesitan acceder a su valor. Por esta razón hemos optado por sustituir el atributo privado `numeroCuenta` en nuestro modelo con la propiedad pública de sólo lectura `NumeroCuenta`. Como ésta es una propiedad de sólo lectura, sólo proporciona un descriptor de acceso `get` para acceder al número de cuenta. Cada clase derivada hereda esta propiedad, lo cual le permite acceder a su número de cuenta según lo necesite para ejecutar una transacción. Ya no listamos `numeroCuenta` en el segundo compartimiento de cada clase derivada, ya que las tres clases derivadas heredan la propiedad `NumeroCuenta` de `Transaccion`.

De acuerdo con la figura 11.19, las clases `SolicitudoSaldo`, `Retiro` y `Deposito` también comparten la operación `Ejecutar`, por lo que la clase base `Transaccion` debería contener la operación `public Ejecutar`. Sin embargo, no tiene sentido implementar a `Ejecutar` en la clase `Transaccion`, ya que la funcionalidad que proporciona esta operación depende del tipo específico de la transacción actual. Por lo tanto, declaramos a

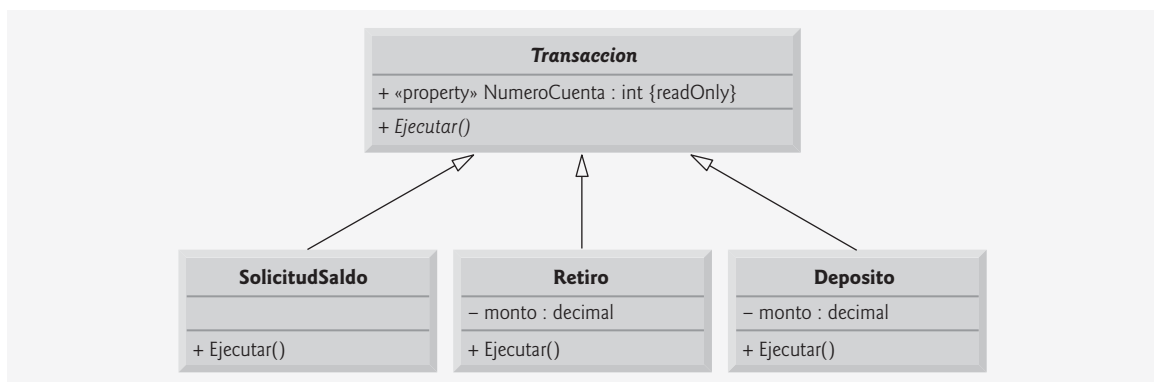


Figura 11.20 | Diagrama de clases que modela la relación de generalización (es decir, herencia) entre la clase base `Transaccion` y sus clases derivadas `SolicitudoSaldo`, `Retiro` y `Deposito`.

Ejecutar como una *operación abstracta* en la clase base Transaccion; se convertirá en un método abstract en la implementación en C#. Esto hace a Transaccion una clase abstracta y obliga a cualquier clase derivada de Transaccion que deba ser una clase concreta (es decir, SolicitudesSaldo, Retiro y Deposito) a implementar la operación Ejecutar, para hacer que la clase derivada sea concreta. UML requiere que coloquemos los nombres de clase abstractos y las operaciones abstractas en cursivas. Por ende, en la figura 11.20 Transaccion y Ejecutar aparecen en cursivas para la clase Transaccion; Ejecutar no se coloca en cursivas en las clases derivadas SolicitudesSaldo, Retiro y Deposito. Cada clase derivada redefine a la operación Ejecutar de Transaccion con una implementación concreta apropiada. Observe que la figura 11.20 incluye la operación Ejecutar en el tercer compartimiento de las clases SolicitudesSaldo, Retiro y Deposito, ya que cada clase tiene una implementación concreta distinta de la operación redefinida.

Como aprendió en este capítulo, una clase derivada puede heredar la interfaz y la implementación de una clase base. En comparación con una jerarquía diseñada para la herencia de implementación, una diseñada para la herencia de interfaz tiende a tener su funcionalidad a un nivel más bajo en la jerarquía; una clase base significa una o más operaciones que debe definir cada clase en la jerarquía, pero las clases derivadas individuales proporcionan sus propias implementaciones de la(s) operación(es). La jerarquía de herencia diseñada para el sistema ATM aprovecha este tipo de herencia, la cual proporciona a la clase ATM una manera elegante de ejecutar todas las transacciones “en general” (es decir, en forma polimórfica). Cada clase derivada de Transaccion hereda ciertos detalles de implementación (por ejemplo, la propiedad NumeroCuenta), pero el principal beneficio de incorporar la herencia en nuestro sistema es que las clases derivadas comparten una interfaz común (por ejemplo, la operación abstracta Ejecutar). La clase ATM puede dirigir una referencia Transaccion a cualquier transacción, y cuando ATM invoca a la operación Ejecutar a través de esta referencia, se ejecuta la versión de Ejecutar específica para esa transacción (en forma polimórfica) de manera automática (debido al polimorfismo). Por ejemplo, suponga que un usuario selecciona la opción para solicitar su saldo. La clase ATM dirige una referencia Transaccion a un nuevo objeto de la clase SolicitudesSaldo, que el compilador de C# permite debido a que un objeto SolicitudesSaldo es un objeto Transaccion. Cuando la clase ATM utiliza esta referencia para invocar a Ejecutar, se llama a la versión de Ejecutar de SolicitudesSaldo (en forma polimórfica).

Este enfoque polimórfico también facilita la extensibilidad del sistema. Si deseamos crear un nuevo tipo de transacción (por ejemplo, una transferencia de fondos o el pago de un recibo), tan sólo tenemos que crear una clase derivada de Transaccion adicional que redefine la operación Ejecutar con una versión apropiada para el nuevo tipo de transacción. Sólo tendríamos que realizar pequeñas modificaciones al código del sistema, para permitir que los usuarios seleccionaran el nuevo tipo de transacción del menú principal y para que la clase ATM creara instancias y ejecutara objetos de la nueva clase derivada. La clase ATM podría ejecutar transacciones del nuevo tipo utilizando el código actual, ya que éste ejecuta todas las transacciones de manera idéntica (a través del polimorfismo).

Como aprendió antes en este capítulo, una clase abstracta tal como Transaccion es una para la cual el programador nunca tendrá la intención de (y de hecho, no podrá) crear objetos. Una clase abstracta sólo declara los atributos y comportamientos comunes para sus clases derivadas en una jerarquía de herencia. La clase Transaccion define el concepto de lo que significa ser una transacción que tiene un número de cuenta y puede ejecutarse. Tal vez usted se pregunte por qué nos tomamos la molestia de incluir la operación abstracta Ejecutar en la clase Transaccion, si Ejecutar carece de una implementación completa. En concepto, incluimos esta operación porque es el comportamiento que define a todas las transacciones: ejecutarse. Técnicamente, debemos incluir la operación Ejecutar en la clase base Transaccion, de manera que la clase ATM (o cualquier otra clase) pueda invocar a la versión redefinida de esta operación de cada clase derivada en forma polimórfica, a través de una referencia Transaccion.

Las clases derivadas SolicitudesSaldo, Retiro y Deposito heredan la propiedad NumeroCuenta de la clase base Transaccion, pero las clases Retiro y Deposito contienen el atributo adicional monto que las diferencia de la clase SolicitudesSaldo. Las clases Retiro y Deposito requieren este atributo adicional para almacenar el monto de dinero que el usuario desea retirar o depositar. La clase SolicitudesSaldo no necesita dicho atributo puesto que sólo requiere un número de cuenta para ejecutarse. Aun cuando dos de las tres clases derivadas de Transaccion comparten el atributo monto, no lo colocamos en la clase base Transaccion; en la clase base sólo colocamos las características comunes para *todas* las clases derivadas, para que éstas no hereden atributos (y operaciones) innecesarios.

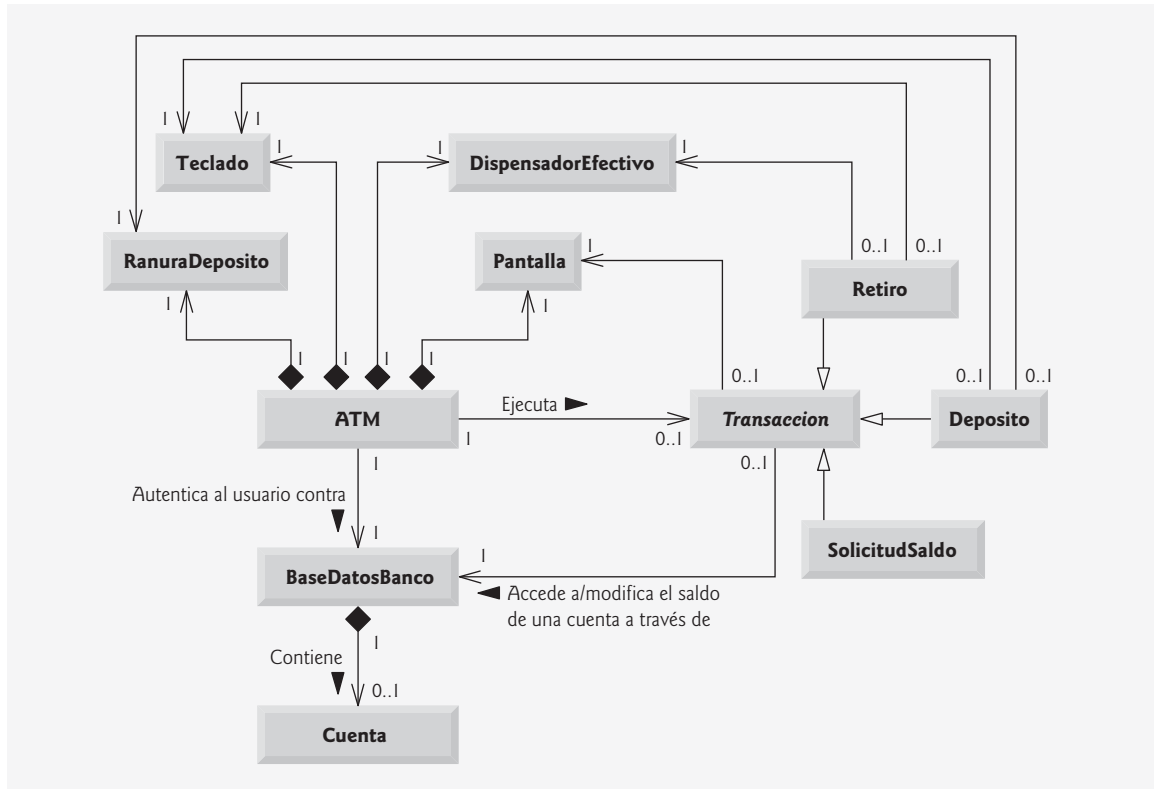


Figura 11.21 | Diagrama de clases del sistema ATM (en el que se incorpora la herencia). Observe que el nombre de la clase abstracta *Transaccion* aparece en cursivas.

La figura 11.21 presenta un diagrama de clases actualizado de nuestro modelo, en el cual se incorpora la herencia y se introduce la clase base abstracta *Transaccion*. Modelamos una asociación entre la clase ATM y la clase *Transaccion* para mostrar que la clase ATM, en cualquier momento dado, está ejecutando una transacción o no lo está (es decir, existen cero o un objetos de tipo *Transaccion* en el sistema, en cualquier momento dado). Como un Retiro es un tipo de *Transaccion*, ya no dibujamos una línea de asociación directamente entre la clase ATM y la clase Retiro; la clase derivada Retiro hereda la asociación de la clase *Transaccion* con la clase ATM. Las clases derivadas SolicitusSaldo y Deposito también heredan esta asociación, la cual sustituye a las asociaciones que omitimos antes entre las clases SolicitusSaldo y Deposito, y la clase ATM. Observe de nuevo el uso de las puntas de flecha triangulares sin relleno para indicar las especializaciones (es decir, clases derivadas) de la clase *Transaccion*, como se indica en la figura 11.20.

También agregamos una asociación entre la clase *Transaccion* y la clase BaseDatosBanco (figura 11.21). Todos los objetos *Transaccion* requieren una referencia a BaseDatosBanco, de manera que puedan acceder y modificar la información de las cuentas. Cada clase derivada de *Transaccion* hereda esta referencia, por lo que ya no tenemos que modelar la asociación entre la clase Retiro y BaseDatosBanco. Observe que la asociación entre la clase *Transaccion* y BaseDatosBanco sustituye a las asociaciones que omitimos antes entre las clases SolicitusSaldo y Deposito, y BaseDatosBanco.

Incluimos una asociación entre la clase *Transaccion* y la clase Pantalla, debido a que todos los objetos *Transaccion* muestran los resultados al usuario a través de la Pantalla. Cada clase derivada hereda esta asociación. Por lo tanto, ya no incluimos la asociación que modelamos antes entre Retiro y Pantalla. No obstante, la clase Retiro aún participa en las asociaciones con DispensadorEfectivo y Teclado; estas asociaciones se aplican a la clase derivada Retiro, pero no a las clases derivadas SolicitusSaldo y Deposito, por lo que no movemos estas asociaciones a la clase base *Transaccion*.

Nuestro diagrama de clases que incorpora la herencia (figura 11.21) también modela las clases `Deposito` y `SolicitudSaldo`. Mostramos las asociaciones entre `Deposito` y las clases `RanuraDeposito` y `Teclado`. Observe que la clase `SolicitudSaldo` participa sólo en las asociaciones heredadas de la clase `Transaccion`; una `SolicitudSaldo` interactúa sólo con la `BaseDatosBanco` y la `Pantalla`.

El diagrama de clases de la figura 9.23 muestra los atributos, las propiedades y las operaciones con marcadores de visibilidad. Ahora presentamos un diagrama de clases modificado en la figura 11.22, el cual incluye la clase base abstracta `Transaccion`. Este diagrama abreviado no muestra las relaciones de herencia (éstas aparecen en la figura 11.21), sino los atributos y las operaciones después de haber empleado la herencia en nuestro sistema. Observe que el nombre de la clase base abstracta `Transaccion` y el nombre de la operación abstracta `Ejecutar` aparecen en cursivas. Para ahorrar espacio, como hicimos en la figura 5.16, no incluimos los atributos mostrados por las asociaciones en la figura 11.22; sin embargo, los incluimos en la implementación en C# del apéndice J. También omitimos todos los parámetros de las operaciones, como hicimos en la figura 9.23; al incorporar la herencia no se afectan los parámetros que ya estaban modelados en las figuras 7.22-7.24.

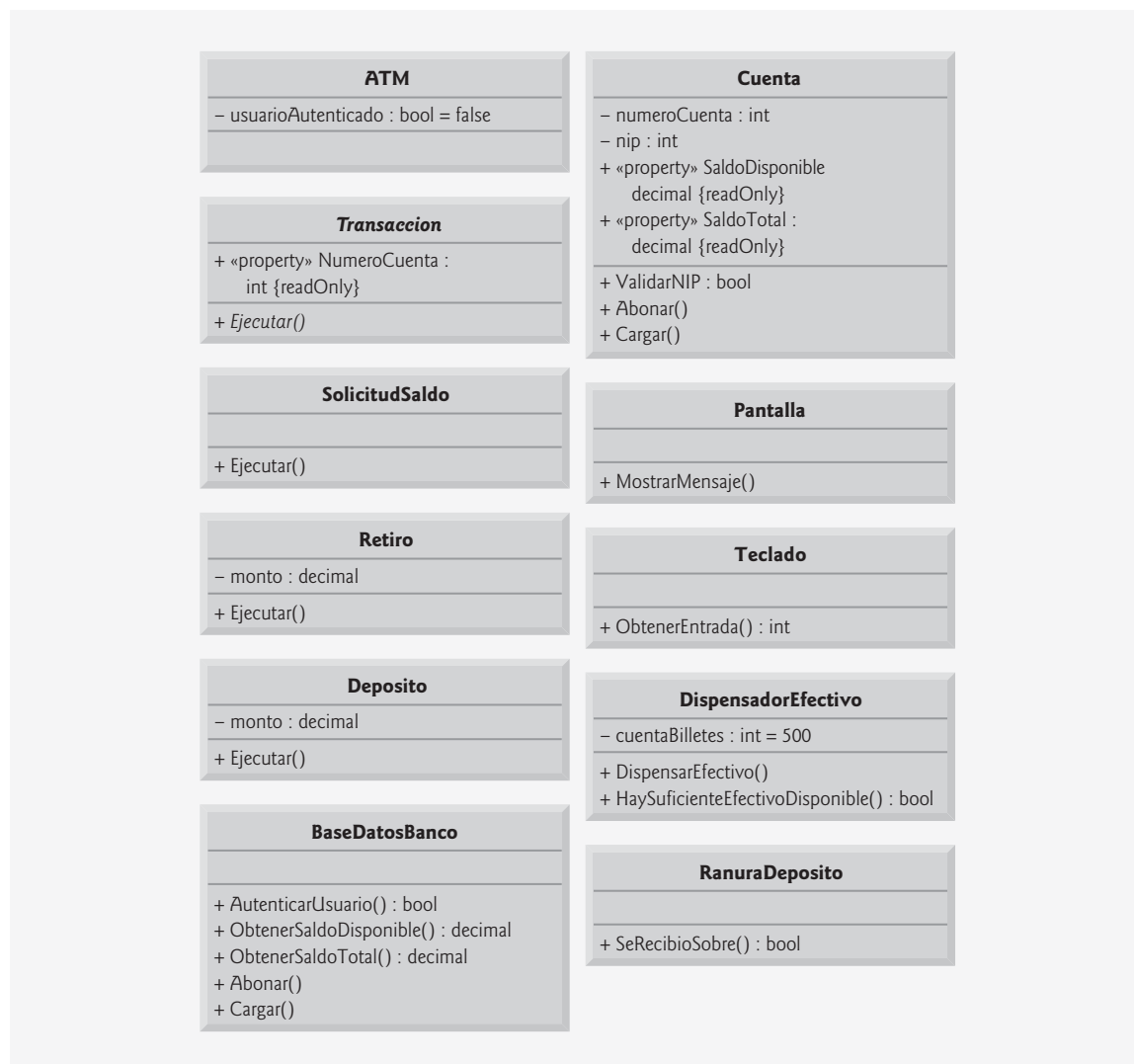


Figura 11.22 | Diagrama de clases después de incorporar la herencia en el sistema.



Observación de ingeniería de software 11.12

Un diagrama de clases completo muestra todas las asociaciones entre clases, junto con todos los atributos y operaciones para cada clase. Cuando el número de atributos, operaciones y asociaciones de las clases es substancial (como en las figuras 11.21 y 11.22), una buena práctica que promueve la legibilidad es dividir esta información entre dos diagramas de clases: uno que se enfoque en las asociaciones y el otro en los atributos y operaciones. Al examinar las clases modeladas en esta forma, es imprescindible considerar ambos diagramas de clases para obtener una idea completa acerca de las clases. Por ejemplo, uno debe referirse a la figura 11.21 para observar la relación de herencia entre *Transaccion* y sus clases derivadas; esa relación se omite en la figura 11.22.

Implementación del diseño del sistema ATM en el que se incorpora la herencia

En la sección 9.17 empezamos a implementar el diseño del sistema ATM en código de C#. Ahora modificaremos nuestra implementación para incorporar la herencia, usando la clase *Retiro* como ejemplo.

1. Si la clase A es una generalización de la clase B, entonces la clase B se deriva (y es una especialización) de la clase A. Por ejemplo, la clase base abstracta *Transaccion* es una generalización de la clase *Retiro*. Por ende, la clase *Retiro* se deriva (y es una especialización) de la clase *Transaccion*. La figura 11.23 contiene la estructura de la clase *Retiro*, en la que la definición de la clase indica la relación de herencia entre *Retiro* y *Transacción* (línea 3).
2. Si la clase A es una clase abstracta y la clase B se deriva de la clase A, entonces la clase B debe implementar las operaciones abstractas de la clase A, si la clase B va a ser una clase concreta. Por ejemplo, la clase *Transaccion* contiene la operación abstracta *Ejecutar*, por lo que la clase *Retiro* debe implementar esta operación si queremos crear instancias de objetos *Retiro*. La figura 11.24 contiene las porciones del código en C# para la clase *Retiro* que pueden inferirse de las figuras 11.21 y 11.22. La clase *Retiro* hereda la propiedad *NumeroCuenta* de la clase base *Transaccion*, por lo que *Retiro* no declara esta propiedad. La clase *Retiro* también hereda referencias a las clases *Pantalla* y *BaseDatosBanco* de la clase *Transaccion*, por lo que no incluimos estas referencias en nuestro código. La figura 11.22 especifica el atributo *monto* y la operación *Ejecutar* para la clase *Retiro*. La línea 6 de la figura 11.24 declara una variable de instancia para el atributo *monto*. Las líneas 17-20 declaran la estructura de un método para la operación *Ejecutar*. Recuerde que la clase derivada *Retiro* debe proporcionar una implementación concreta del método abstracto *Ejecutar* de la clase base *Transaccion*. Las referencias *teclado* y *dispensadorEfectivo* (líneas 7-8) son variables de instancia, cuya necesidad es aparente debido a las asociaciones de la clase *Retiro* en la figura 11.21; en la implementación en C# de esta clase en el apéndice J, un constructor inicializa estas referencias a objetos reales.

```

1 // Fig 11.23: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM.
3 public class Retiro : Transaccion
4 {
5     // código para los miembros de la clase Retiro
6 } // fin de la clase Retiro

```

Figura 11.23 | Código en C# para la estructura de la clase *Retiro*.

```

1 // Fig 11.24: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM.
3 public class Retiro : Transaccion
4 {
5     // atributos
6     private decimal monto; // monto a retirar
7     private Teclado teclado; // referencia al teclado

```

Figura 11.24 | Código en C# para la clase *Retiro*, con base en las figuras 11.21 y 11.22. (Parte I de 2).


```

8      private DispensadorEfectivo dispensadorEfectivo; // referencia al dispensador de
      efectivo
9
10     // constructor sin parámetros
11     public Retiro()
12     {
13         // código del cuerpo del constructor
14     } // fin del constructor
15
16     // método que redefine a Ejecutar
17     public override void Ejecutar()
18     {
19         // código del cuerpo del método Ejecutar
20     } // fin del método Ejecutar
21 } // fin de la clase Retiro

```

Figura 11.24 | Código en C# para la clase Retiro, con base en las figuras 11.21 y 11.22. (Parte 2 de 2).

En la sección J.2 de la implementación del ATM hablaremos sobre el procesamiento polimórfico de objetos Transaccion. La clase ATM realiza la llamada polimórfica al método Ejecutar en la línea 99 de la figura J.1.

Conclusión del caso de estudio sobre el ATM

Esto concluye nuestro diseño orientado a objetos del sistema ATM. En el apéndice J aparece una implementación en C# completa del sistema ATM, en 655 líneas de código. Esta implementación funcional utiliza la mayor parte de los conceptos de programación orientada a objetos que hemos cubierto hasta este punto en el libro, incluyendo: clases, objetos, encapsulamiento, visibilidad, composición, herencia y polimorfismo. El código contiene abundantes comentarios y se conforma a las prácticas de codificación que usted ha aprendido hasta ahora. Dominar este código será una maravillosa experiencia culminante para usted, después de haber estudiado las nueve secciones del Caso de estudio de ingeniería de software en los capítulos 1, 3 al 9 y 11.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

11.1 UML utiliza una flecha con una _____ para indicar una relación de generalización.

- a) punta con relleno sólido
- b) punta triangular sin relleno
- c) punta hueca en forma de diamante
- d) punta lineal

11.2 Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: UML requiere que subrayemos los nombres de las clases abstractas y los nombres de las operaciones abstractas.

11.3 Escriba código en C# para empezar a implementar el diseño para la clase Transaccion que se especifica en las figuras 11.21 y 11.22. Asegúrese de incluir las referencias *private* basadas en las asociaciones de la clase Transaccion. Asegúrese también de incluir las propiedades con los descriptores de acceso *public get* para cualquiera de las variables de instancia *private* a las que deban acceder las clases derivadas, para realizar sus tareas.

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

11.1 b.

11.2 Falso. UML requiere que se escriban los nombres de las clases y de las operaciones en cursivas.

11.3 El diseño para la clase Transaccion produce el código de la figura 11.25. En la implementación en el apéndice J, un constructor inicializa las variables de instancia *private pantallaUsuario* y *baseDatos* para objetos reales, y las propiedades de sólo lectura *PantallaUsuario* y *BaseDatos* acceden a estas variables de instancia. Estas propiedades permiten que las clases derivadas de Transaccion accedan a la pantalla del ATM e interactúen con la base de datos del banco. Observe que elegimos los nombres de las propiedades *PantallaUsuario* y *BaseDatos* por cuestión de claridad; queremos evitar nombres de propiedades que sean iguales que los nombres de las clases *Pantalla* y *BaseDatosBanco*, lo cual puede ser confuso.

```

1  // Fig 11.25: Transaccion.cs
2  // La clase base abstracta Transaccion representa una transacción del ATM.
3  public abstract class Transaccion
4  {
5      private int numeroCuenta; // indica la cuenta involucrada
6      private Pantalla pantallaUsuario; // pantalla del ATM
7      private BaseDatosBanco baseDatos; // base de datos de información de las cuentas
8
9      // constructor sin parámetros
10     public Transaccion()
11     {
12         // código del cuerpo del constructor
13     } // fin del constructor
14
15     // propiedad de sólo lectura que obtiene el número de cuenta
16     public int NumeroCuenta
17     {
18         get
19         {
20             return numeroCuenta;
21         } // fin de get
22     } // fin de la propiedad NumeroCuenta
23
24     // propiedad de sólo lectura que obtiene la referencia a la pantalla
25     public Pantalla pantallaUsuario
26     {
27         get
28         {
29             return pantallaUsuario;
30         } // fin de get
31     } // fin de la propiedad PantallaUsuario
32
33     // propiedad de sólo lectura que obtiene la referencia a la base de datos del banco
34     public BaseDatosBanco BaseDatos
35     {
36         get
37         {
38             return baseDatos;
39         } // fin de get
40     } // fin de la propiedad BaseDatos
41
42     // realiza la transacción (cada clase derivada redefine este método)
43     public abstract void Ejecutar();
44 } // fin de la clase Transaccion

```

Figura 11.25 | Código en C# para la clase Transaccion, con base en las figuras 11.21 y 11.22.

11.10 Conclusión

En este capítulo se introdujo el polimorfismo: la habilidad de procesar objetos que comparten la misma clase base en una jerarquía de clases, como si todos fueran objetos de la clase base. En este capítulo hablamos sobre cómo el polimorfismo facilita la extensibilidad y manejabilidad de los sistemas, y después demostramos cómo utilizar métodos redefinidos para llevar a cabo el comportamiento polimórfico. Presentamos la noción de una clase abstracta, la cual nos proporciona una clase base apropiada, a partir de la cual otras clases pueden heredar. Aprendió que una clase abstracta puede declarar métodos abstractos que debe implementar cada clase derivada para convertirse en clase concreta, y que una aplicación puede utilizar variables de una clase abstracta para invocar implementaciones de clases derivadas de los métodos abstractos en forma polimórfica. También aprendió a determinar el tipo de un objeto en tiempo de ejecución. Le mostramos cómo crear métodos y clases *sealed*. Hablamos

también sobre la declaración e implementación de una interfaz, como otra manera de lograr el comportamiento polimórfico, a menudo entre objetos de distintas clases. Por último, aprendió a definir el comportamiento de los operadores integrados en objetos de sus propias clases, mediante la sobrecarga de operadores.

Ahora deberá estar familiarizado con las clases, los objetos, el encapsulamiento, la herencia, las interfaces y el polimorfismo: los aspectos más esenciales de la programación orientada a objetos. En el siguiente capítulo analizaremos con más detalle cómo utilizar el manejo de excepciones para lidiar con los errores en tiempo de ejecución.

12

Manejo de excepciones

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Qué son las excepciones y cómo se manejan.
- Cuándo utilizar el manejo de excepciones.
- Utilizar bloques `try` para delimitar código en el que podrían ocurrir las excepciones.
- Lanzar (`throw`) excepciones para indicar un problema.
- Utilizar bloques `catch` para especificar manejadores de excepciones.
- Utilizar el bloque `finally` para liberar recursos.
- La jerarquía de clases de excepciones de .NET.
- Las propiedades de `Exception`.
- Crear excepciones definidas por el usuario.

*Es cuestión de sentido común
tomar un método y probarlo.
Si falla, admítalo francamente
y pruebe otro. Pero sobre todo,
inténtelo.*

—Franklin Delano Roosevelt

*¡Oh! Arroja
la peor parte de ello,
y vive en forma más pura
con la otra mitad.*

—William Shakespeare

*Si están corriendo y no saben
hacia dónde se dirigen
tengo que salir de alguna parte
y atraparlos.*

—J. D. Salinger

*A menudo,
la excusa de una falta
hace que ésta sea aún peor
debido a la excusa.*

—William Shakespeare

*¡Oh, infinita virtud! ¿Vienes
sonriendo de la gran trampa
del mundo sin ser atrapada?*

—William Shakespeare

- 12.1 Introducción
- 12.2 Generalidades acerca del manejo de excepciones
- 12.3 Ejemplo: división entre cero sin manejo de excepciones
- 12.4 Ejemplo: manejo de las excepciones `DivideByZeroException` y `FormatException`
 - 12.4.1 Encerrar código en un bloque `try`
 - 12.4.2 Atrapar excepciones
 - 12.4.3 Excepciones no atrapadas
 - 12.4.4 Modelo de terminación del manejo de excepciones
 - 12.4.5 Flujo de control cuando ocurren las excepciones
- 12.5 Jerarquía `Exception` de .NET
 - 12.5.1 Las clases `ApplicationException` y `SystemException`
 - 12.5.2 Determinar cuáles excepciones debe lanzar un método
- 12.6 El bloque `finally`
- 12.7 Propiedades de `Exception`
- 12.8 Clases de excepciones definidas por el usuario
- 12.9 Conclusión

12.1 Introducción

En este capítulo presentaremos el *manejo de excepciones*. Una *excepción* es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre “excepción” viene del hecho de que, aunque puede ocurrir un problema, éste ocurre con poca frecuencia. Si la “regla” es que una instrucción generalmente se ejecuta en forma correcta, entonces la ocurrencia de un problema representa la “excepción a la regla”. El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que el programa continúe su ejecución como si no se hubieran encontrado problemas. No obstante, los problemas más graves podrían evitar que un programa continuara su ejecución normal, en vez de requerir al programa que notifique al usuario sobre el problema y después terminar de una manera controlada. Las características que presentamos en este capítulo permiten a los programadores escribir *programas más tolerantes a fallas y robustos* (es decir, programas que pueden lidiar con problemas que pueden surgir y continúan ejecutándose). El estilo y los detalles del manejo de excepciones de C# se basan, en parte, en el trabajo de Andrew Koenig y Bjarne Stroustrup. Las “mejores prácticas” para el manejo de excepciones en Visual C# 2005 se especifican en la documentación de Visual Studio.¹



Tip de prevención de errores 12.1

El manejo de excepciones ayuda a mejorar la tolerancia a fallas de un programa.

Este capítulo empieza con una descripción general de los conceptos y demostraciones de las técnicas básicas del manejo de excepciones. El capítulo también presenta una descripción general de la jerarquía de clases de manejo de excepciones de .NET. Por lo general, los programas solicitan y liberan recursos (como archivos en el disco) durante la ejecución de un programa. A menudo, el suministro de estos recursos está limitado, o sólo un programa puede utilizar los recursos a la vez. Demostraremos una parte del mecanismo de manejo de excepciones que permite que un programa utilice un recurso y después garantice que será liberado para que lo utilicen otros programas, aun cuando ocurra una excepción. El capítulo demuestra varias propiedades de la clase `System.Exception` (la clase base de todas las clases de excepciones) y habla acerca de cómo usted puede crear y utilizar sus propias clases de excepciones.

1. “Best Practices for Handling Exceptions [C#]”, *Guía del desarrollador de .NET Framework*, Ayuda en línea de Visual Studio .NET. Disponible en msdn2.microsoft.com/en-us/library/seyhshzts.aspx.

12.2 Generalidades acerca del manejo de excepciones

Con frecuencia, los programas evalúan condiciones para determinar cómo debe proceder la ejecución de un programa. Considere el siguiente pseudocódigo:

```
Realizar una tarea

Si la tarea anterior no se ejecutó en forma correcta
    Realizar el procesamiento de los errores

Realizar la siguiente tarea

Si la tarea anterior no se ejecutó en forma correcta
    Realizar el procesamiento de los errores

...
```

En este pseudocódigo, empezaremos por realizar una tarea; después evaluaremos si se ejecutó en forma correcta. Si no lo hizo, realizamos el procesamiento de los errores. De otra manera, continuamos con la siguiente tarea. Aunque esta forma de manejo de excepciones funciona, al entremezclar la lógica del programa con la lógica del manejo de errores el programa podría ser difícil de leer, modificar, mantener y depurar; especialmente en aplicaciones extensas.

El manejo de excepciones permite a los programadores remover el código para manejo de errores de la “línea principal” de ejecución del programa, lo cual mejora su claridad y aumenta la capacidad de modificación del mismo. Los programadores pueden optar por manejar todas las excepciones que elijan: todas las excepciones, todas las excepciones de cierto tipo o todas las excepciones de un grupo de tipos relacionados (por ejemplo, los tipos de excepciones que están relacionados a través de una jerarquía de herencia). Esta flexibilidad reduce la probabilidad de que los errores se pasen por alto y, por consecuencia, hace que un programa sea más robusto.

Con lenguajes de programación que no soportan el manejo de excepciones, los programadores a menudo retrasan la escritura de *código de procesamiento de errores* y en ocasiones olvidan incluirlo. Esto hace que los productos de software sean menos robustos. C# permite a los programadores tratar con el manejo de excepciones fácilmente, desde el comienzo de un proyecto.

12.3 Ejemplo: división entre cero sin manejo de excepciones

Primero demostraremos lo que ocurre cuando surgen los errores en una aplicación de consola que no utiliza el manejo de excepciones. La figura 12.1 recibe como entrada dos enteros del usuario, y después divide el primer entero entre el segundo entero, utilizando la división entera para obtener un resultado `int`. En este ejemplo veremos que se *lanza* una excepción (es decir, ocurre una excepción) cuando un método detecta un problema y no puede manejarlo.

```
1 // Fig. 12.1: DivisionEntreCeroSinManejoExcepciones.cs
2 // Una aplicación que trata de dividir entre cero.
3 using System;
4
5 class DivisionEntreCeroSinManejoExcepciones
6 {
7     static void Main()
8     {
9         // obtiene el numerador y el denominador
10        Console.Write( "Escriba un numerador entero: " );
11        int numerador = Convert.ToInt32( Console.ReadLine() );
12        Console.Write( "Escriba un denominador entero: " );
13        int denominador = Convert.ToInt32( Console.ReadLine() );
14
15        // divide los dos enteros, después muestra en pantalla el resultado
```

Figura 12.1 | División de enteros sin el manejo de excepciones. (Parte 1 de 2).

```

16     int resultado = numerador / denominador;
17     Console.WriteLine( "\nResultado: {0:D} / {1:D} = {2:D}",
18         numerador, denominador, resultado );
19 } // fin de Main
20 } // fin de la clase DivisionEntreCeroSinManejoExcepciones

```

Escriba un numerador entero: 100
Escriba un denominador entero: 7

Resultado: 100 / 7 = 14

Escriba un numerador entero: 100
Escriba un denominador entero: 0

Excepción no controlada: System.DivideByZeroException:
Intento de dividir por cero.
en DivisionEntreCeroSinManejoExcepciones.Main()
en C:\MisProyectos\fig12_01\fig12_01\
DivisionEntreCeroSinManejoExcepciones.cs:línea 16

Escriba un numerador entero: 100
Escriba un denominador entero: **hola**

Excepción no controlada: System.FormatException:
La cadena de entrada no tiene el formato correcto.
en System.Number.StringToNumber(String str, NumberStyles options,
NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
en System.Number.ParseInt32(String s, NumberStyles style,
NumberFormatInfo info)
en System.Convert.ToString32(String value)
en DivisionEntreCeroSinManejoExcepciones.Main()
en C:\MisProyectos\fig12_01\fig12_01\
DivisionEntreCeroSinManejoExcepciones.cs:línea 13

Figura 12.1 | División de enteros sin el manejo de excepciones. (Parte 2 de 2).

Ejecución de la aplicación

En la mayoría de los ejemplos que hemos creado hasta ahora, la aplicación parece ejecutarse igual con o sin la depuración. Como veremos en breve, el ejemplo en la figura 12.1 podría ocasionar errores, dependiendo de la entrada del usuario. Si ejecuta esta aplicación usando la opción **Depurar > Iniciar depuración**, el programa se detiene en la línea en la que ocurre una excepción y muestra el Ayudante de excepciones, que le permite analizar el estado actual del programa y depurarlo. En la sección 12.4.3 hablaremos sobre el Ayudante de excepciones. En el apéndice C hablaremos con detalle sobre la depuración.

En este ejemplo no deseamos depurar la aplicación: sólo queremos ver qué ocurre cuando surgen los errores. Por esta razón, ejecutaremos la aplicación desde una ventana de Símbolo del sistema. Seleccione **Inicio > Todos los programas > Accesorios > Símbolo del sistema** para abrir una ventana Símbolo del sistema y después utilice el comando `cd` para cambiar al directorio debug de la aplicación. Por ejemplo, si esta aplicación reside en el directorio `C:\ejemplos\cap12\Fig12_01\DivisionEntreCeroSinManejoExcepciones` en su sistema, debería escribir:

```
cd /d C:\ejemplos\cap12\Fig12_01\DivisionEntreCeroSinManejoExcepciones\bin\Debug
```

en el Símbolo del sistema, ahora oprima *Intro* para cambiar al directorio debug de la aplicación. Para ejecutar la aplicación, escriba:

```
DivisionEntreCeroSinManejoExcepciones.exe
```


en el Símbolo del sistema, y después oprima *Intro*. Si surge un error durante la ejecución, se muestra un cuadro de diálogo indicando que la aplicación encontró un problema y necesita cerrarse. El cuadro de diálogo también le pregunta si desea enviar información acerca de este error a Microsoft. Como crearemos este error para fines de demostración, haga clic en **No enviar**. [Nota: en algunos sistemas se muestra un cuadro de diálogo **Depuración Just-In-Time**. Si esto ocurre, simplemente haga clic en el botón **No** para cerrar el cuadro de diálogo.] En este punto aparecerá un mensaje en el Símbolo del sistema, en el que se describe el problema. En la figura 12.1 aplicamos formato a los mensajes de error para mejorar la legibilidad. [Nota: al seleccionar **Depurar > Iniciar sin depurar** (o <Ctrl> F5) para ejecutar la aplicación desde Visual Studio, se ejecuta la denominada versión de liberación de la aplicación. Los mensajes de error producidos por esta versión de la aplicación pueden ser distintos a los que se muestran en la figura 12.1, debido a las optimizaciones que realiza el compilador para crear la versión de liberación de una aplicación.]

Análisis de los resultados

La primera ejecución de ejemplo en la figura 12.1 muestra una división exitosa. En la segunda ejecución de ejemplo, el usuario introduce 0 como el denominador. Observe que se muestran varias líneas de información, en respuesta a la entrada inválida. Esta información (conocida como *rastreo de pila*) incluye el nombre de la excepción (System.DivideByZeroException) en un mensaje descriptivo que indica el problema que ocurrió, y la ruta de ejecución que condujo a la excepción, método por método. Esta información le ayuda a depurar un programa. La primera línea del mensaje de error especifica que ha ocurrido una excepción DivideByZeroException. Cuando ocurre la *división entre cero* en la aritmética de enteros, el CLR lanza una excepción **DivideByZeroException** (espacio de nombres System). El texto después de la excepción, “Intento de dividir por cero”, indica que esta excepción ocurrió como resultado de un intento de dividir entre cero. La división entre cero no se permite en la aritmética de enteros. [Nota: se permite la división entre cero con valores de punto flotante. Dicho cálculo produce el valor de infinito, el cual se representa mediante la constante **Double.PositiveInfinity** o la constante **Double.NegativeInfinity**, dependiendo de si el numerador es positivo o negativo. Estos valores se muestran como Infinity o -Infinity. Si tanto el numerador como el denominador son cero, el resultado del cálculo es la constante **Double.NaN** (“no es un número”), el cual se devuelve cuando el resultado de un cálculo es indefinido.]

Cada línea “en” en el rastreo de pila indica una línea de código en el método específico que se estaba ejecutando cuando ocurrió la excepción. La línea “en” contiene el espacio de nombres, el nombre de la clase y el nombre del método en el cual ocurrió la excepción (DivisionEntreCeroSinManejoExcepciones.Main), la ubicación y el nombre del archivo en el que reside el código (C:\MisProyectos\Cap12\fig12_01\DivisionEntreCeroSinManejoExcepciones.cs: línea 16) y la línea de código en donde ocurrió la excepción. En este caso, el rastreo de pila indica que la excepción DivideByZeroException ocurrió cuando el programa estaba ejecutando la línea 16 del método Main. La primera línea “en” en el rastreo de pila indica el *punto de lanzamiento* de la excepción: el punto inicial en que ocurrió la excepción (es decir, la línea 16 en Main). Esta información facilita al programador la tarea de ver en dónde se originó la excepción, y qué llamadas a métodos se hicieron para llegar hasta ese punto en el programa.

Ahora veamos un rastreo de pila más detallado. En la tercera ejecución de ejemplo, el usuario introduce la cadena “ho1a” como el denominador. Esto produce una excepción **FormatException** y se muestra otro rastreo de pila. En nuestros primeros ejemplos que leían valores numéricos del usuario, se asumía que éste introduciría un valor entero. No obstante, un usuario podría introducir por error un valor no entero. Por ejemplo, una excepción **FormatException** (espacio de nombres System) ocurre cuando el método ToInt32 de Convert recibe una cadena que no representa un entero válido. Empezando desde la última línea “en” en el rastreo de pila, podemos ver que la excepción se detectó en la línea 13 el método Main. El rastreo de pila también muestra los demás métodos que condujeron a la excepción que se lanzó: Convert.ToInt32, Number.ParseInt32 y Number.StringToNumber. Para realizar su tarea, Convert.ToInt32 llama al método Number.ParseInt32, quien a su vez llama a Number.StringToNumber. El punto de lanzamiento ocurre en Number.StringToNumber, como se indica mediante la primera línea “en” en el rastreo de pila.

Observe que en las ejecuciones de ejemplo de la figura 12.1, el programa también termina cuando ocurren las excepciones y se muestran los rastreos de pila. Esto no siempre ocurre; en ocasiones un programa puede seguir ejecutándose aun cuando se haya producido una excepción, y se haya impreso un rastreo de pila. En tales casos, la aplicación puede producir resultados erróneos. La siguiente sección demuestra cómo manejar excepciones para permitir que el programa se ejecute hasta terminar en forma normal.

12.4 Ejemplo: manejo de las excepciones

DivideByZeroException y FormatException

Consideraremos un ejemplo simple de manejo de excepciones. La aplicación en la figura 12.2 utiliza el manejo de excepciones para procesar cualquier excepción `DivideByZeroException` y `FormatException` que pueda surgir. La aplicación muestra dos objetos `TextBox` en los que el usuario puede escribir enteros. Cuando el usuario oprime **Haga clic para dividir**, el programa invoca al manejador de eventos `DividirButton_Click` (líneas 17-48), el cual obtiene la entrada del usuario, convierte los valores de entrada al tipo `int` y divide el primer número (numerador) entre el segundo número (denominador). Suponiendo que el usuario proporcione enteros como entrada y no especifique 0 como el denominador para la división, `DividirButton_Click` muestra el resultado de la división en `SalidaLabel`. No obstante, si el usuario introduce un valor no entero o 0 como el denominador, ocurre una excepción. Este programa demuestra cómo *atrapar* y *manejar* (es decir, tratar con) dichas excepciones; en este caso, muestra un mensaje de error y permite al usuario que introduzca otro conjunto de valores.

```

1  // Fig. 12.2: PruebaDivisionEntreCero.cs
2  // Manejadores de excepciones para FormatException y DivideByZeroException.
3  using System;
4  using System.Windows.Forms;
5
6  namespace PruebaDivisionEntreCero
7  {
8      public partial class DivisionEntreCeroForm : Form
9      {
10         public DivisionEntreCeroForm()
11         {
12             InitializeComponent();
13         } // fin del constructor
14
15         // obtiene 2 enteros del usuario
16         // y divide el numerador entre el denominador
17         private void DividirButton_Click( object sender, EventArgs e )
18         {
19             SalidaLabel.Text = ""; // borra control Label EtiquetaSalida
20
21             // extrae entrada del usuario y calcula el cociente
22             try
23             {
24                 // Convert.ToInt32 genera excepción FormatException
25                 // si el argumento no es un entero
26                 int numerador = Convert.ToInt32( NumeradorTextBox.Text );
27                 int denominador = Convert.ToInt32( DenominadorTextBox.Text );
28
29                 // la división genera una excepción DivideByZeroException
30                 // si el denominador es 0
31                 int resultado = numerador / denominador;
32
33                 // muestra el resultado en SalidaLabel
34                 SalidaLabel.Text = resultado.ToString();
35             } // fin de try
36             catch ( FormatException )
37             {
38                 MessageBox.Show( "Debe escribir dos enteros.",
39                                 "Formato de número inválido", MessageBoxButtons.OK,
40                                 MessageBoxIcon.Error );
41             } // fin de catch
42             catch ( DivideByZeroException divideByZeroExceptionParameter )

```

Figura 12.2 | Manejadores de excepciones para `FormatException` y `DivideByZeroException`. (Parte I de 2).

```

43 {
44     MessageBox.Show( divideByZeroExceptionParameter.Message,
45         "Intento de división por cero", MessageBoxButtons.OK,
46         MessageBoxIcon.Error );
47 } // fin de catch
48 } // fin del método DividirButton_Click
49 } // fin de la clase FormDivisionEntreCero
50 } // fin del espacio de nombres PruebaDivisionEntreCero

```

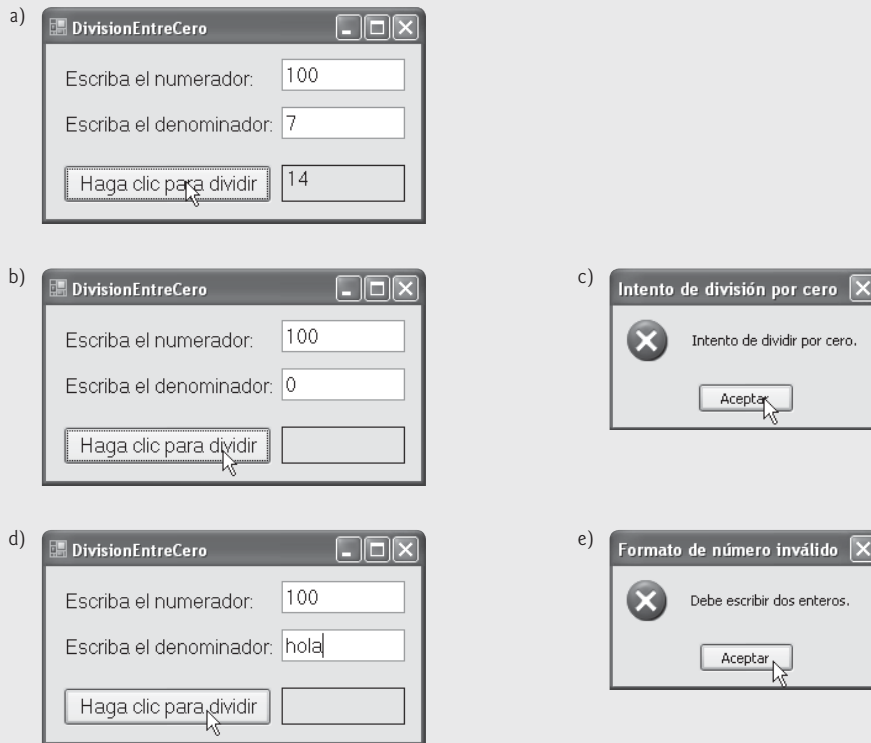


Figura 12.2 | Manejadores de excepciones para `FormatException` y `DivideByZeroException`. (Parte 2 de 2).

Antes de hablar sobre los detalles del programa, consideraremos las ventanas de resultados en la figura 12.2. La ventana en la figura 12.2(a) muestra un cálculo exitoso, en el que el usuario introduce el numerador 100 y el denominador 7. Observe que el resultado (14) es un `int`, ya que la división de enteros siempre produce un resultado `int`. Las siguientes dos ventanas, las figuras 12.2(b) y 12.2(c), demuestran el resultado de un intento de dividir entre cero. En la aritmética de enteros, el CLR evalúa la condición de división entre cero y genera una excepción `DivideByZeroException` si el denominador es cero. El programa detecta la excepción y muestra el cuadro de diálogo con el mensaje de error en la figura 12.2(c), indicando el intento de dividir entre cero. Las últimas dos ventanas de resultados, las figuras 12.2(d) y 12.2(e), ilustran el resultado de introducir un valor no `int`; en este caso, el usuario introduce “hola” en el segundo objeto `TextBox`, como se muestra en la figura 12.2(d). Cuando el usuario hace clic en **Haga clic para dividir**, el programa intenta convertir los objetos `string` de entrada en valores `int` mediante el uso del método `Convert.ToInt32` (líneas 26-27). Si un argumento que se pasa a `Convert.ToInt32` no puede convertirse en valor `int`, el método lanza una excepción `FormatException`. El programa atrapa la excepción y muestra el cuadro de diálogo con el mensaje de error en la figura 12.2(e), indicando que el usuario debe introducir dos valores `int`. Observe que no incluimos el nombre de un parámetro para la instrucción `catch` en la línea 36. En el bloque del `catch` no utilizamos información del objeto `FormatException`. Al omitir el nombre del parámetro evitamos que el compilador genere una advertencia en la que se indique que declaramos una variable, pero no la utilizamos en el bloque `catch`.

12.4.1 Encerrar código en un bloque try

Ahora consideraremos las interacciones del usuario y el flujo de control que producen los resultados que se muestran en las ventanas de resultados de ejemplo. El usuario introduce valores en los objetos `TextBox` que representan el numerador y el denominador, y después oprime el botón **Haga clic para dividir**. En este punto, el programa invoca al método `DividirButton_Click`. La línea 19 asigna la cadena vacía a `SalidaLabel` para borrar cualquier resultado anterior y prepararse para un nuevo cálculo. Las líneas 22-35 definen un **bloque try**, el cual encierra el código que podría lanzar excepciones, así como el código que se omite cuando ocurre una excepción. Por ejemplo, el programa no debería mostrar un nuevo resultado en `SalidaLabel` (línea 34), a menos que el cálculo en la línea 31 se complete con éxito.

Las dos instrucciones que leen los valores `int` de los objetos `TextBox` (líneas 26-27) llaman al método `Convert.ToInt32` para convertir objetos `string` en valores `int`. Este método lanza una excepción `FormatException` si no puede convertir su argumento `string` en un `int`. Si las líneas 26-27 convierten los valores en forma apropiada (es decir, que no ocurran excepciones), entonces la línea 31 divide el numerador entre el denominador y asigna el resultado a la variable `resultado`. Si `denominador` es 0, la línea 31 hace que el CLR lance una excepción `DivideByZeroException`. Si la línea 31 no hace que se lance una excepción, entonces la línea 34 muestra el resultado de la división.

12.4.2 Atrapar excepciones

El código para el manejo de excepciones aparece en un **bloque catch**. En general, cuando ocurre una excepción en un bloque `try`, un bloque `catch` correspondiente atrapa la excepción y la maneja. En este ejemplo, el bloque `try` va seguido de dos bloques `catch`: uno que maneja una excepción `FormatException` (líneas 36-41) y otro que maneja una excepción `DivideByZeroException` (líneas 42-47). Un bloque `catch` especifica un parámetro de excepción que representa la excepción que puede manejar ese bloque `catch`. El bloque `catch` puede utilizar el identificador del parámetro (que el programador elige) para interactuar con un objeto de excepción atrapada. Si no hay necesidad de usar el objeto de excepción en el bloque `catch`, puede omitirse el identificador del parámetro de la excepción. El tipo del parámetro del `catch` es el tipo de la excepción que maneja ese bloque `catch`. De manera opcional, los programadores pueden incluir un bloque `catch` que no especifique un tipo de excepción o un identificador; dicho bloque `catch` (conocido como **cláusula catch general**) atrapa todos los tipos de excepciones. Debe haber por lo menos un bloque `catch` y/o un **bloque finally** (que veremos en la sección 12.6) inmediatamente después de un bloque `try`.

En la figura 12.2, el primer bloque `catch` atrapa excepciones `FormatException` (lanzadas por el método `Convert.ToInt32`), y el segundo bloque `catch` atrapa excepciones `DivideByZeroException` (lanzadas por el CLR). Si ocurre una excepción, el programa ejecuta sólo el primer bloque `catch` que concuerde. Ambos manejadores de excepciones en este ejemplo muestran un cuadro de diálogo con un mensaje de error. Una vez que termina cualquiera de los bloques `catch`, el control del programa continúa con la primera instrucción después del último bloque `catch` (en este ejemplo, el final del método). Pronto veremos con más detalle la forma en que funciona este flujo de control en el manejo de excepciones.

12.4.3 Excepciones no atrapadas

Una **excepción no atrapada** es una excepción para la que no hay un bloque `catch` relacionado. Usted vio los resultados de las excepciones no atrapadas en el segundo y tercer conjunto de resultados de la figura 12.1. Recuerde que cuando ocurren excepciones en ese ejemplo, la aplicación termina en forma anticipada (después de mostrar el rastreo de pila de la excepción). El resultado de una excepción no atrapada depende de la forma en que se ejecuta el programa; la figura 12.1 demostró los resultados de una excepción no atrapada cuando se ejecuta una aplicación en un Símbolo del sistema. Si usted ejecuta la aplicación desde Visual Studio con depuración y el entorno en tiempo de ejecución detecta una excepción no atrapada, la aplicación se pausa y aparece una ventana llamada *Ayudante de excepciones*, la cual indica en dónde ocurrió la excepción, el tipo de la excepción y vínculos a información útil acerca de cómo manejar la excepción. La figura 12.3 muestra el Ayudante de excepciones que aparece si el usuario intenta dividir entre cero, en la aplicación de la figura 12.1.

12.4.4 Modelo de terminación del manejo de excepciones

Cuando un método que se llama en un programa o el CLR detectan un problema, el método o el CLR lanzan una excepción. Recuerde que el punto en el programa en el cual ocurre una excepción se conoce como el punto

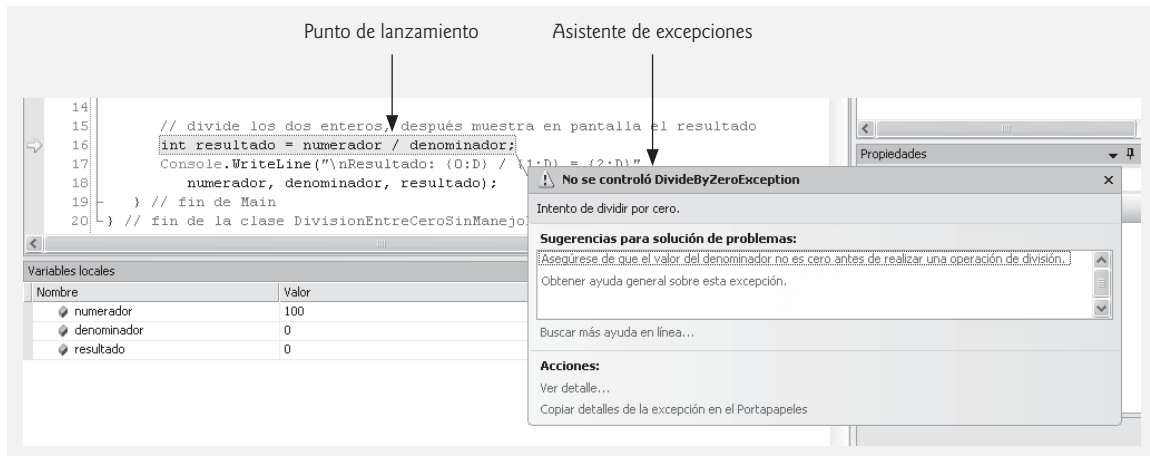


Figura 12.3 | Ayudante de excepciones.

de lanzamiento; ésta es una ubicación importante para fines de depuración (como veremos en la sección 12.7). Si ocurre una excepción en un bloque try (por ejemplo, cuando se lanza una *FormatException* como resultado del código en la línea 27 de la figura 12.2), el bloque try termina de inmediato y el control del programa se transfiere al primero de los siguientes bloques catch en los que el tipo del parámetro de la excepción concuerde con el tipo de la excepción que se lanzó. En la figura 12.2, el primer bloque catch atrapa excepciones *FormatException* (que ocurren cuando se introduce un tipo inválido); el segundo bloque catch atrapa excepciones *DivideByZero* (que ocurren si hay un intento de dividir entre cero). Una vez que se maneja la excepción, el control del programa no regresa al punto de lanzamiento, ya que el bloque try ha expirado (lo cual también hace que cualquiera de sus variables locales quede fuera de alcance). En vez de ello, el control continúa después del último bloque catch. Esto se conoce como el *modelo de terminación del manejo de excepciones*. [Nota: algunos lenguajes utilizan el *modelo de reanudación del manejo de excepciones*, en el cual después de manejar una excepción, el control se reanuda justo después del punto de lanzamiento.]



Error común de programación 12.1

Pueden ocurrir errores lógicos si asumimos que, después de manejar una excepción, el control regresará a la primera instrucción después del punto de lanzamiento.

Si no ocurre una excepción en el bloque try, el programa de la figura 12.2 completa con éxito el bloque try, ignorando los bloques catch en las líneas 36-41 y 42-47, y pasa a la línea 47. Después, el programa ejecuta la primera instrucción después de los bloques try y catch. En este ejemplo, el programa llega al final del manejador de eventos *DividirButton_Click* (línea 48), por lo que el método termina y el programa espera la siguiente interacción del usuario.

En conjunto, el bloque try y sus correspondientes bloques catch y finally forman una *instrucción try*. Es importante no confundir los términos “bloque try” e “instrucción try”; el término “bloque try” se refiere al bloque de código que va después de la palabra clave try (pero antes de cualquier bloque catch o finally), mientras que el término “instrucción try” incluye todo el código desde la palabra clave de apertura try, hasta el final del último bloque catch o finally. Esto incluye el bloque try, así como cualquier bloque catch y finally asociado.

Al igual que con cualquier otro bloque de código, cuando termina un bloque try, las variables locales definidas en el bloque quedan fuera de alcance. Si un bloque try termina debido a una excepción, el CLR busca el primer bloque catch que pueda procesar el tipo de excepción que ocurrió. Para localizar el catch que concuerde, el CLR compara el tipo de la excepción lanzada con el tipo de parámetro de catch. Se produce una concordancia si los tipos son idénticos, o si el tipo de la excepción lanzada es una clase derivada del tipo del parámetro catch. Una vez que se relaciona una excepción con un bloque catch, se ejecuta el código en ese bloque y se ignoran los demás bloques catch en la instrucción try.

12.4.5 Flujo de control cuando ocurren las excepciones

En los resultados de ejemplo de la figura 12.2(b), el usuario introduce `ho1a` como el denominador. Cuando se ejecuta la línea 27, `Convert.ToInt32` no puede convertir este `string` en un `int`, por lo que lanza un objeto `FormatException` para indicar que el método no pudo convertir el `string` en `int`. Cuando ocurre la excepción, el bloque `try` expira (termina). Después, el CLR trata de localizar un bloque `catch` que concuerde. Se produce una concordancia con el bloque `catch` en la línea 36, por lo que se ejecuta el manejador de excepciones y el programa ignora todos los demás manejadores de excepciones que van después del bloque `try`.



Error común de programación 12.2

Especificar una lista de parámetros separada por comas en un bloque `catch` es un error de sintaxis. Un bloque `catch` puede tener a lo más un parámetro.

En los resultados de ejemplo de la figura 12.2(d), el usuario introduce un `0` como el denominador. Cuando se ejecuta la división en la línea 31, se produce una excepción `DivideByZeroException`. Una vez más el bloque `try` termina y el programa trata de localizar un bloque `catch` que concuerde. En este caso, el primer bloque `catch` no concuerda; el tipo de excepción en la declaración del manejador del `catch` no es el mismo que el tipo de la excepción lanzada, y `FormatException` no es una clase base de `DivideByZeroException`. Por lo tanto, el programa continúa buscando un bloque `catch` que concuerde, y lo encuentra en la línea 42. La línea 44 muestra el valor de la propiedad **Message** de la clase `Exception`, la cual contiene el mensaje de error. Observe que nuestro programa nunca “establece” este atributo del mensaje de error. Esto lo hace el CLR cuando crea el objeto excepción.

12.5 Jerarquía `Exception` de .NET

En C#, el mecanismo de manejo de excepciones sólo permite lanzar y atrapar objetos de la clase **Exception** (espacio de nombres `System`) y sus clases derivadas. No obstante, hay que tener en cuenta que los programas en C# pueden interactuar con componentes de software escritos en otros lenguajes .NET (como C++) que no restringen los tipos de las excepciones. En estos casos, es posible utilizar la cláusula `catch` general para atrapar dichas excepciones.

En esta sección veremos las generalidades acerca de varias de las clases de excepciones del .NET Framework, y nos enfocaremos exclusivamente en las que se derivan de la clase `Exception`. Además hablaremos acerca de cómo determinar si un método específico va a lanzar excepciones.

12.5.1 Las clases `ApplicationException` y `SystemException`

La clase **Exception** del espacio de nombres `System` es la clase base de la jerarquía de clases de excepciones del .NET Framework. Dos de las clases más importantes que se derivan de `Exception` son **ApplicationException** y **SystemException**. `ApplicationException` es una clase base que los programadores pueden extender para crear clases de excepciones específicas para sus aplicaciones. En la sección 12.8 le mostraremos cómo crear clases de excepciones definidas por el usuario. Los programas pueden recuperarse de la mayoría de las excepciones `ApplicationException` y continuar su ejecución.

El CLR genera excepciones `SystemException`, que pueden ocurrir en cualquier punto durante la ejecución de un programa. Podemos evitar muchas de estas excepciones si codificamos las aplicaciones en forma apropiada. Por ejemplo, si un programa trata de acceder a un *subíndice de arreglo fuera de rango*, el CLR lanza una excepción de tipo **IndexOutOfRangeException** (una clase derivada de `SystemException`). De manera similar, una excepción ocurre cuando un programa utiliza la referencia a un objeto para manipular un objeto que aún no existe (es decir, la referencia tiene un valor de `null`). Si tratamos de utilizar una referencia `null` se produce una excepción **NullReferenceException** (otra clase derivada de `SystemException`). Anteriormente en este capítulo vimos que una excepción `DivideByZeroException` ocurre en la división de enteros, cuando un programa trata de dividir entre cero.

Otros tipos de excepciones `SystemException` que lanza el CLR son: **OutOfMemoryException**, **StackOverflowException** y **ExecutionEngineException**. Éstas se lanzan cuando algo sale mal y el CLR se vuelve inestable. En ciertos casos, incluso dichas excepciones no pueden atraparse. En general, es mejor anotar en el registro esas excepciones y después terminar la aplicación.

Un beneficio de la jerarquía de clases de excepciones es que un bloque `catch` puede atrapar excepciones de un tipo específico o (debido a la relación *es un* de la herencia) puede utilizar un tipo de clase base para atrapar

excepciones, en una jerarquía de tipos de excepciones relacionadas. Por ejemplo, en la sección 12.4.2 vimos el bloque `catch` sin parámetro, que atrapa excepciones de todos los tipos (incluyendo los que no se derivan de `Exception`). Un bloque `catch` que especifica un parámetro de tipo `Exception` puede atrapar todas las excepciones que se derivan de `Exception`, ya que ésta es la clase base de todas las clases de excepciones. La ventaja de este enfoque es que el manejador de excepciones puede acceder a la información de la excepción atrapada mediante el parámetro en el `catch`. En la línea 44 de la figura 12.2 demostramos cómo acceder a la información en una excepción. En la sección 12.7 veremos más acerca del acceso a la información de una excepción.

El uso de la herencia con las excepciones permite a un bloque `catch` atrapar excepciones relacionadas, mediante el uso de una notación concisa. Un conjunto de manejadores de excepciones podría atrapar a cada tipo de excepción de clase derivada en forma individual, pero es más conciso atrapar el tipo de excepción de clase base. No obstante, esta técnica tiene sentido sólo si el comportamiento del manejo es el mismo para una clase base que para todas las clases derivadas. De no ser así, hay que atrapar cada excepción de clase derivada en forma individual.



Error común de programación 12.3

Si un bloque `catch` que atrapa una excepción de clase base se coloca antes de un bloque `catch` para cualquiera de los tipos de clases derivadas de esa clase, se produce un error de compilación. Si se permitiera esto, el bloque `catch` de la clase base atraparía todas las excepciones de la clase base y las clases derivadas, y por lo tanto el manejador de excepciones de las clases derivadas nunca se ejecutaría.

12.5.2 Determinar cuáles excepciones debe lanzar un método

¿Cómo determinamos que podría ocurrir una excepción en un programa? Para los métodos contenidos en las clases del .NET Framework, lea las descripciones detalladas de los métodos en la documentación en línea. Si un método lanza una excepción, su descripción contiene una sección llamada **Exceptions**, la cual especifica los tipos de excepciones que lanza ese método y describe en forma breve las posibles causas de las excepciones. Por ejemplo, busque “método `Convert.ToInt32`” en el **Índice** de la documentación en línea de Visual Studio (use el filtro **.NET Framework**). Seleccione el documento titulado **Convert.ToInt32 (Método) (System)**. En el documento que describe al método, haga clic en el vínculo **Convert.ToInt32(String)**. En el documento que aparezca, la sección **Exceptions** (cerca de la parte inferior del documento) indica que el método `Convert.ToInt32` lanza dos tipos de excepciones (`FormatException` y `OverflowException`) y describe la razón por la cual se podría producir cada una de ellas.



Observación de ingeniería de software 12.1

Si un método lanza excepciones, las instrucciones que invocan al método en forma directa o indirecta deberían colocarse en bloques `try`, y esas excepciones deberían atraparse y manejarse.

Es más difícil determinar cuando el CLR lanza excepciones. Dicha información aparece en la *Especificación del lenguaje C#*. Este documento define la sintaxis de C# y especifica los casos en los que se lanzan excepciones. La figura 12.2 demostró que el CLR lanza una excepción `DivideByZeroException` en la aritmética de enteros, cuando un programa trata de dividir entre cero. La sección 7.7.2 de la especificación del lenguaje (14.7.2 en la versión ECMA) habla sobre el operador de división y cuándo ocurren las excepciones `DivideByZeroException`.

12.6 El bloque **finally**

Con frecuencia, los programas solicitan y liberan recursos en forma dinámica (es decir, en tiempo de ejecución). Por ejemplo, un programa que lee un archivo del disco primero hace una solicitud para abrir el archivo (como veremos en el capítulo 18, Archivos y flujos). Si esa solicitud tiene éxito, el programa lee el contenido del archivo. Por lo general, los sistemas operativos evitan que más de un programa manipule un archivo al mismo tiempo. Por lo tanto, cuando un programa termina de procesar un archivo, éste debe cerrarlo (es decir, liberar el recurso) para que otros programas puedan utilizar el archivo. Si no se cierra, ocurre una *fuga de recursos*. En tal caso, el recurso del archivo no está disponible para otros programas, posiblemente debido a que un programa que utiliza el archivo no lo ha cerrado.

En lenguajes de programación como C y C++, en los que el programador (y no el lenguaje) es responsable de la administración de la memoria dinámica, el tipo más común de fuga de recursos es una *fuga de memoria*.

Este tipo de fuga ocurre cuando un programa asigna memoria (como hacen los programadores en C# mediante la palabra clave `new`), pero no la desasigna cuando ya no se necesita. Por lo general esto no representa un problema en C#, ya que el CLR se encarga de la recolección de basura para la memoria que ya no necesita un programa en ejecución (sección 9.10). No obstante, pueden ocurrir otros tipos de fugas de recursos (como los archivos sin cerrar).



Tip de prevención de errores 12.2

El CLR no elimina por completo las fugas de memoria. El CLR no hará la recolección de basura sobre un objeto sino hasta que el programa no contenga más referencias a ese objeto. Por ende, pueden ocurrir fugas de memoria si los programadores mantienen referencias a objetos indeseados, sin darse cuenta.

Desplazar el código de liberación de recursos a un bloque `finally`

Por lo general, las excepciones ocurren cuando se procesan recursos que requieren una liberación explícita. Por ejemplo, un programa que procesa un archivo podría recibir excepciones `IOException` durante el procesamiento. Por esta razón, el código para procesar archivos aparece normalmente en un bloque `try`. Sin importar si un programa experimenta excepciones mientras procesa un archivo, debe cerrarlo cuando ya no lo necesita. Suponga que un programa coloca todo el código de solicitud y liberación de recursos en un bloque `try`. Si no ocurren excepciones, el bloque `try` se ejecuta normalmente y libera los recursos después de utilizarlos. No obstante, si ocurre una excepción, el bloque `try` tal vez expire antes de que el código de liberación de recursos pueda ejecutarse. Podríamos duplicar todo el código de liberación de recursos en cada uno de los bloques `catch`, pero esto haría que el código fuera más difícil de modificar y mantener. También podríamos colocar el código de liberación de recursos después de la instrucción `try`; pero si el bloque `try` termina debido a una instrucción de retorno, el código después de la instrucción `try` nunca se ejecutaría.

Para lidiar con estos problemas, el mecanismo de manejo de excepciones de C# cuenta con el bloque `finally`, con el cual se garantiza que se ejecutará sin importar que el bloque `try` se ejecute con éxito o que ocurra una excepción. Esto convierte al bloque `finally` en una ubicación ideal en la cual colocar el código de liberación de recursos, para los recursos que se adquieren y se manipulan en el bloque `try` correspondiente. Si el bloque `try` se ejecuta con éxito, el bloque `finally` se ejecuta inmediatamente después de que termina el bloque `try`. Si ocurre una excepción en el bloque `try`, el bloque `finally` se ejecuta inmediatamente después de que termina un bloque `catch`. Si la excepción no se atrapa mediante un bloque `catch` asociado con el bloque `try`, o si un bloque `catch` asociado con el bloque `try` lanza una excepción por sí mismo, el bloque `finally` se ejecuta antes que se procese la excepción por el siguiente bloque `try` circundante (si hay uno). Al colocar el código de liberación de recursos en un bloque `finally`, aseguramos que, aun si el programa termina debido a una excepción no atrapada, el recurso se desasignará. Observe que no se puede acceder a las variables locales de un bloque `try` en el bloque `finally` correspondiente. Por esta razón, las variables a las que se debe acceder tanto en un bloque `try` como en su bloque `finally` correspondiente deben declararse antes del bloque `try`.



Tip de prevención de errores 12.3

Por lo general, un bloque `finally` contiene código para liberar los recursos adquiridos en el bloque `try` correspondiente, lo cual hace del bloque `finally` un mecanismo efectivo para eliminar fugas de recursos.



Tip de rendimiento 12.1

Como regla, los recursos deben liberarse tan pronto como ya no sean necesarios en un programa. Esto los hace disponibles para su reutilización en el menor tiempo posible.

Si uno o más bloques `catch` siguen después de un bloque `try`, el bloque `finally` es opcional. No obstante, si ningún bloque `catch` sigue después de un bloque `try`, debe aparecer un bloque `finally` justo después del bloque `try`. Si uno o más bloques `catch` siguen después de un bloque `try`, el bloque `finally` (si hay uno) aparece después del último bloque `catch`. Sólo puede haber espacio en blanco y comentarios para separar los bloques en una instrucción `try`.



Error común de programación 12.4

*Colocar el bloque **finally** antes de un bloque **catch** es un error de sintaxis.*

Demostración del bloque **finally**

La aplicación en la figura 12.4 demuestra que el bloque **finally** siempre se ejecuta, sin importar que ocurra o no una excepción en el bloque **try** correspondiente. El programa consiste en el método **Main** (líneas 8-47) y otros cuatro métodos que invoca **Main** para demostrar el bloque **finally**. Estos métodos son **NoLanzaExcepcion** (líneas 50-67), **LanzaExcepcionConCatch** (líneas 70-89), **LanzaExcepcionSinCatch** (líneas 92-108) y **LanzaExcepcionCatchLanzaDeNuevo** (líneas 111-136).

```

1  // Fig. 12.4: UsoDeExcepciones.cs
2  // Uso de los bloques finally.
3  // Demuestra que un bloque finally siempre se ejecuta.
4  using System;
5
6  class UsoDeExcepciones
7  {
8      static void Main()
9      {
10         // Caso 1: No ocurren excepciones en el método que se llamó
11         Console.WriteLine( "Llamando a NoLanzaExcepcion" );
12         NoLanzaExcepcion();
13
14         // Caso 2: Ocurre una excepción y se atrapa en el método que se llamó
15         Console.WriteLine( "\nLlamando a LanzaExcepcionConCatch" );
16         LanzaExcepcionConCatch();
17
18         // Caso 3: Ocurre una excepción, pero no se atrapa en el método que se llamó,
19         // ya que no hay bloque catch.
20         Console.WriteLine( "\nLlamando a LanzaExcepcionSinCatch" );
21
22         // llama a LanzaExcepcionSinCatch
23         try
24         {
25             LanzaExcepcionSinCatch();
26         } // fin de try
27         catch
28         {
29             Console.WriteLine( "Atrapó la excepción de " +
30                 "LanzaExcepcionSinCatch en Main" );
31         } // fin de catch
32
33         // Caso 4: Ocurre una excepción y se atrapa en el método que se llamó,
34         // después se vuelve a lanzar al que hizo la llamada.
35         Console.WriteLine( "\nLlamando a LanzaExcepcionCatchLanzaDeNuevo" );
36
37         // llama a LanzaExcepcionCatchLanzaDeNuevo
38         try
39         {
40             LanzaExcepcionCatchLanzaDeNuevo();
41         } // fin de try
42         catch
43         {
44             Console.WriteLine( "Atrapó la excepción de " +
45                 "LanzaExcepcionCatchLanzaDeNuevo en Main" );
46         } // fin de catch

```

Figura 12.4 | Los bloques **finally** siempre se ejecutan, sin importar que ocurra o no una excepción. (Parte I de 3).


```

47     } // fin del método Main
48
49     // no se lanzan excepciones
50     static void NoLanzaExcepcion()
51     {
52         // el bloque try no lanza excepciones
53         try
54         {
55             Console.WriteLine( "En NoLanzaExcepcion" );
56         } // fin de try
57         catch
58         {
59             Console.WriteLine( "Este bloque catch nunca se ejecuta" );
60         } // fin de catch
61         finally
62         {
63             Console.WriteLine( "se ejecutó finally en NoLanzaExcepcion" );
64         } // fin de finally
65
66         Console.WriteLine( "Fin de NoLanzaExcepcion" );
67     } // fin del método NoLanzaExcepcion
68
69     // lanza la excepción y la atrapa en forma local
70     static void LanzaExcepcionConCatch()
71     {
72         // el bloque try lanza la excepción
73         try
74         {
75             Console.WriteLine( "En LanzaExcepcionConCatch" );
76             throw new Exception( "Excepción en LanzaExcepcionConCatch" );
77         } // fin de try
78         catch (Exception parametroExcepcion)
79         {
80             Console.WriteLine( "Mensaje: " + parametroExcepcion.Message );
81         } // fin de catch
82         finally
83         {
84             Console.WriteLine(
85                 "se ejecutó finally en LanzaExcepcionConCatch" );
86         } // fin de finally
87
88         Console.WriteLine( "Fin de LanzaExcepcionConCatch" );
89     } // fin del método LanzaExcepcionConCatch
90
91     // lanza la excepción y no la atrapa en forma local
92     static void LanzaExcepcionSinCatch()
93     {
94         // lanza la excepción, pero no la atrapa
95         try
96         {
97             Console.WriteLine( "En LanzaExcepcionSinCatch" );
98             throw new Exception( "Excepción en LanzaExcepcionSinCatch" );
99         } // fin de try
100        finally
101        {
102            Console.WriteLine( "se ejecutó finally en " +
103                "LanzaExcepcionSinCatch" );
104        } // fin de finally

```

Figura 12.4 | Los bloques finally siempre se ejecutan, sin importar que ocurra o no una excepción. (Parte 2 de 3).

```

105
106     // código inalcanzable; error lógico
107     Console.WriteLine( "Fin de LanzaExcepcionSinCatch" );
108 } // fin del método LanzaExcepcionSinCatch
109
110 // lanza la excepción, la atrapa y la vuelve a lanzar
111 static void LanzaExcepcionCatchLanzaDeNuevo()
112 {
113     // el bloque try lanza la excepción
114     try
115     {
116         Console.WriteLine( "En LanzaExcepcionCatchLanzaDeNuevo" );
117         throw new Exception( "Excepción en LanzaExcepcionCatchLanzaDeNuevo" );
118     } // fin de try
119     catch ( Exception parametroExcepcion )
120     {
121         Console.WriteLine( "Mensaje: " + parametroExcepcion.Message );
122
123         // vuelve a lanzar la excepción para procesarla después
124         throw;
125
126         // código inalcanzable; error lógico
127     } // fin de catch
128     finally
129     {
130         Console.WriteLine( "se ejecutó finally en " +
131             "LanzaExcepcionCatchLanzaDeNuevo" );
132     } // fin de finally
133
134     // cualquier código que se coloque aquí nunca se ejecutará
135     Console.WriteLine( "Fin de LanzaExcepcionCatchLanzaDeNuevo" );
136 } // fin del método LanzaExcepcionCatchLanzaDeNuevo
137 } // fin de la clase UsoDeExcepciones

```

```

Llamando a NoLanzaExcepcion
En NoLanzaExcepcion
se ejecutó finally en NoLanzaExcepcion
Fin de NoLanzaExcepcion

Llamando a LanzaExcepcionConCatch
En LanzaExcepcionConCatch
Mensaje: Excepción en LanzaExcepcionConCatch
se ejecutó finally en LanzaExcepcionConCatch
Fin de LanzaExcepcionConCatch

Llamando a LanzaExcepcionSinCatch
En LanzaExcepcionSinCatch
se ejecutó finally en LanzaExcepcionSinCatch
Atrapó la excepción de LanzaExcepcionSinCatch en Main

Llamando a LanzaExcepcionCatchLanzaDeNuevo
En LanzaExcepcionCatchLanzaDeNuevo
Mensaje: Excepción en LanzaExcepcionCatchLanzaDeNuevo
se ejecutó finally en LanzaExcepcionCatchLanzaDeNuevo
Atrapó la excepción de LanzaExcepcionCatchLanzaDeNuevo en Main

```

Figura 12.4 | Los bloques **finally** siempre se ejecutan, sin importar que ocurra o no una excepción. (Parte 3 de 3).

La línea 12 de Main invoca al método `NoLanzaExcepcion`. El bloque `try` para este método imprime un mensaje en pantalla (línea 55). Como el bloque `try` no lanza excepciones, el control del programa ignora el bloque `catch` (líneas 57-60) y ejecuta el bloque `finally` (líneas 61-64), el cual imprime un mensaje en pantalla. En

este punto, el control del programa continúa con la primera instrucción que sigue después del cierre del bloque `finally` (línea 66), la cual imprime un mensaje en pantalla indicando que se ha llegado al final del método. Después, el control del programa regresa a `Main`.

Lanzar excepciones mediante el uso de la instrucción `throw`

La línea 16 de `Main` invoca al método `LanzarExcepcionConCatch` (líneas 70-89), el cual empieza en su bloque `try` (líneas 73-77) imprimiendo un mensaje. A continuación, el bloque `try` crea un objeto `Exception` y utiliza una **instrucción `throw`** para lanzar el objeto excepción (línea 76). Al ejecutar la instrucción `throw` se indica que ocurrió una excepción. Hasta ahora sólo hemos atrapado excepciones que lanzan los métodos a los que se llama. Con la instrucción `throw` puede lanzar excepciones. Al igual que con las excepciones que lanzan los métodos de la FCL y el CLR, esto indica a las aplicaciones cliente que ocurrió un error. Una instrucción `throw` especifica un objeto a lanzar. El operando de una instrucción `throw` puede ser de tipo `Exception`, o de cualquier tipo derivado de la clase `Exception`.



Error común de programación 12.5

Si el argumento de una instrucción `throw` (un objeto excepción) no es de la clase `Exception` o de una de sus clases derivadas, se produce un error de compilación.

El objeto `string` que se pasa al constructor se convierte en el mensaje de error del objeto excepción. Cuando se ejecuta una instrucción `throw` en un bloque `try`, el bloque `try` expira de inmediato y el control del programa continúa con el primer bloque `catch` que concuerde (líneas 78-81), que siga después del bloque `try`. En este ejemplo, el tipo lanzado (`Exception`) concuerda con el tipo especificado en el bloque `catch`, por lo que la línea 80 imprime un mensaje en pantalla indicando que ocurrió la excepción. Después, el bloque `finally` (líneas 82-86) se ejecuta e imprime un mensaje en pantalla. En este punto, el control del programa continúa con la primera instrucción que va después del cierre del bloque `finally` (línea 88), el cual imprime un mensaje en pantalla que indica que se ha llegado al final del archivo. Después, el control del programa regresa a `Main`. En la línea 80, observe que utilizamos la propiedad `Message` del objeto excepción para extraer el mensaje de error asociado con la excepción (es decir, el mensaje que se pasa al constructor de `Exception`). La sección 12.7 habla sobre varias propiedades de la clase `Exception`.

Las líneas 23-31 de `Main` definen una instrucción `try`, en la que `Main` invoca al método `LanzaExcepcionSinCatch` (líneas 92-108). El bloque `try` permite que `Main` atrape cualquier excepción lanzada por `LanzaExcepcionSinCatch`. El bloque `try` en las líneas 95-99 de `LanzaExcepcionSinCatch` empieza imprimiendo un mensaje en pantalla. Después, el bloque `try` lanza una excepción tipo `Exception` (línea 98) y expira de inmediato.

Por lo general, el control del programa continuaría en el primer bloque `catch` después de este bloque `try`. Sin embargo, este bloque `try` no tiene bloques `catch`. Por lo tanto, la excepción no se atrapa en el método `LanzaExcepcionSinCatch`. El control del programa se reanuda en el bloque `finally` (líneas 100-104), el cual imprime un mensaje en pantalla. En este punto, el control del programa regresa a `Main`; cualquier instrucción que aparezca después del bloque `finally` (por ejemplo, la línea 107) no se ejecuta. En este ejemplo, dichas instrucciones podrían ocasionar errores lógicos, ya que la excepción que se lanza en la línea 98 no se atrapa. En `Main`, el bloque `catch` en las líneas 27-31 atrapa la excepción y muestra un mensaje en pantalla, indicando que la excepción se atrapó en `Main`.

Volver a lanzar excepciones

Las líneas 38-46 de `Main` definen una instrucción `try` en la que `Main` invoca al método `LanzaExcepcionCatchLanzaDeNuevo` (líneas 111-136). La instrucción `try` permite que `Main` atrape cualquier excepción que lance `LanzaExcepcionCatchLanzaDeNuevo`. La instrucción `try` en las líneas 114-132 de `LanzaExcepcionCatchLanzaDeNuevo` empieza imprimiendo un mensaje en pantalla. A continuación, el bloque `try` lanza una excepción tipo `Exception` (línea 117). El bloque `try` expira de inmediato y el control del programa continúa en el primer bloque `catch` (líneas 119-127) que va después del bloque `try`. En este ejemplo, el tipo que se lanza (`Exception`) concuerda con el tipo especificado en el bloque `catch`, por lo que la línea 121 imprime un mensaje de salida que indica en dónde ocurrió la excepción. La línea 124 utiliza la instrucción `throw` para **volver a lanzar** la excepción. Esto indica que el bloque `catch` realizó un procesamiento parcial de la excepción y ahora la está pasando de vuelta al método que hizo la llamada (en este caso, `Main`) para que la siga procesando.

También se puede volver a lanzar una excepción con una versión de la instrucción **throw** que recibe un operando, el cual es la referencia a la excepción que se atrapó. No obstante, es importante tener en cuenta que esta forma de instrucción **throw** restablece el punto de lanzamiento, por lo que se pierde la información original del rastreo de pila del punto de lanzamiento. La sección 12.7 demuestra el uso de una instrucción **throw** con un operando proveniente de un bloque **catch**. En esa sección veremos que, después de atrapar una excepción, se puede crear y lanzar un tipo distinto de objeto excepción del bloque **catch** y se puede incluir la excepción original como parte del nuevo objeto excepción. Los diseñadores de biblioteca de clases hacen esto con frecuencia, para personalizar los tipos de excepciones que se lanzan de los métodos en sus bibliotecas de clases, o para proporcionar información de depuración adicional.

El manejo de excepciones en el método `LanzaExcepcionCatchLanzaDeNuevo` no se completa, ya que el programa no puede ejecutar el código en el bloque **catch** que se coloca después de la invocación de la instrucción **throw** en la línea 124. Por lo tanto, el método `LanzaExcepcionCatchLanzaDeNuevo` termina y devuelve el control a `Main`. Una vez más, el bloque **finally** (líneas 128-132) se ejecuta e imprime un mensaje en pantalla, antes de que regrese el control a `Main`. Cuando el control regresa a `Main`, el bloque **catch** en las líneas 42-46 atrapa la excepción y muestra un mensaje que indica que se atrapó la excepción. Después, el programa termina.

Regresar después de un bloque *finally*

Observe que la siguiente instrucción a ejecutar después de que termina un bloque **finally** depende del estado del manejo de la excepción. Si el bloque **try** se completa con éxito, o si un bloque **catch** atrapa y maneja una excepción, el programa continúa su ejecución con la siguiente instrucción después del bloque **finally**. No obstante, si no se atrapa una excepción, o si un bloque **catch** vuelve a lanzar una excepción, el control del programa continúa en el siguiente bloque **try** circundante. El bloque **try** circundante podría estar en el método que hizo la llamada o en uno de los que lo llamaron. También es posible anidar una instrucción **try** en un bloque **try**; en tal caso, los bloques **catch** de la instrucción **try** externa procesarían las excepciones que no se atraparan en la instrucción **try** interna. Si se ejecuta un bloque **try** y tiene un bloque **finally** correspondiente, el bloque **finally** se ejecuta incluso aunque termine el bloque **try**, debido a una instrucción **return**. El retorno ocurre después de la ejecución del bloque **finally**.



Error común de programación 12.6

*Lanzar una excepción desde un bloque **finally** puede ser peligroso. Si una excepción sin atrapar está esperando ser procesada cuando se ejecute el bloque **finally**, y el bloque **finally** lanza una nueva excepción que no se atrapa en el bloque **finally**, se pierde la primera excepción y la nueva excepción se pasa al siguiente bloque **try** circundante.*



Tip de prevención de errores 12.4

*Cuando coloque código que puede lanzar una excepción en un bloque **finally**, siempre encierre el código en la instrucción **try** que atrapa los tipos de excepciones apropiados. Esto evita la pérdida de cualquier excepción no atrapada y vuelta a lanzar, que ocurra antes de que se ejecute el bloque **finally**.*



Observación de ingeniería de software 12.2

*No coloque bloques **try** alrededor de cada instrucción que pueda lanzar una excepción, ya que esto puede dificultar la legibilidad de sus programas. Es mejor colocar un bloque **try** alrededor de una porción considerable de código, y colocar después este bloque **try** con bloques **catch** que manejen cada una de las posibles excepciones. Luego coloque un solo bloque **finally** después de los bloques **catch**. Se deben utilizar bloques **try** separados cuando es importante diferenciar entre varias instrucciones que puedan lanzar el mismo tipo de excepción.*

using

Anteriormente en esta sección vimos que el código de liberación de recursos debe colocarse en un bloque **finally** para asegurar que se libere un recurso, sin importar si hubo excepciones cuando se utilizó el recurso en el bloque **try** correspondiente. La instrucción **using**, una instrucción alternativa (que no debe confundirse con la directiva **using** para usar espacios de nombres), simplifica la escritura del código mediante el cual se obtiene un recurso, se utiliza el recurso en un bloque **try** y se libera el recurso en un bloque **finally** correspondiente. Por ejemplo, una aplicación de procesamiento de archivos (capítulo 18) podría procesar un archivo con una instrucción **using** para asegurar que el archivo se cierre en forma apropiada cuando ya no se necesite. El recurso debe ser un objeto

que implementa la interfaz `IDisposable` y, por lo tanto, tiene un método `Dispose`. La forma general de una instrucción `using` sería:

```
using ( ObjetoEjemplo objetoEjemplo = new ObjetoEjemplo() )
{
    objetoEjemplo.UnMetodo();
}
```

en donde `ObjetoEjemplo` es una clase que implementa a la interfaz `IDisposable`. Este código crea un objeto de tipo `ObjetoEjemplo` y lo utiliza en una instrucción; después llama a su método `Dispose` para liberar cualquier recurso utilizado por el objeto. La instrucción `using` coloca en forma implícita el código de su cuerpo en un bloque `try` con su correspondiente bloque `finally`, el cual llama al método `Dispose` del objeto. Por ejemplo, el código anterior es equivalente a:

```
{
    ObjetoEjemplo objetoEjemplo = new ObjetoEjemplo();
    try
    {
        objetoEjemplo.UnMetodo();
    }
    finally
    {
        if (objetoEjemplo != null )
            ( ( IDisposable ) objetoEjemplo ).Dispose();
    }
}
```

Observe que la instrucción `if` asegura que `objetoEjemplo` sigue referenciando a un objeto; de no ser así, podría ocurrir una excepción `NullReferenceException`. En la sección 8.13 de la *Especificación del lenguaje C#* (sección 15.13 en la versión ECMA) podrá leer más acerca del uso de la instrucción `using`.

12.7 Propiedades de `Exception`

Como vimos en la sección 12.5, los tipos de excepciones se derivan de la clase `Exception`, la cual tiene varias propiedades. Éstas se utilizan con frecuencia para formular mensajes de error que indican una excepción atrapada. Dos propiedades importantes son `Message` y **`StackTrace`**. La propiedad `Message` almacena el mensaje de error asociado con un objeto `Exception`. Este mensaje puede ser un mensaje predeterminado asociado con el tipo de excepción, o puede ser un mensaje personalizado que se pasa al constructor de un objeto `Exception` cuando éste se lanza. La propiedad `StackTrace` contiene un objeto `string` que representa la *pila de llamadas a los métodos*. Recuerde que el entorno en tiempo de ejecución mantiene en todo momento una lista de llamadas abiertas a métodos, que se han realizado pero que no han regresado aún. La propiedad `StackTrace` representa la serie de métodos que no han terminado de procesarse al momento en que ocurre la excepción.



Tip de prevención de errores 12.5

Un rastreo de pila muestra la pila de llamadas a métodos completa, al momento en que ocurrió una excepción. Esto permite al programador ver la serie de llamadas a métodos que condujeron a la excepción. La información en el rastreo de pila incluye los nombres de los métodos en la pila de llamadas al momento de la excepción, los nombres de las clases en las que están definidos los métodos y los nombres de los espacios de nombres en los que están definidas las clases. Si el archivo de base de datos del programa (PDB) que contiene la información de depuración para el método está disponible, el rastreo de pila también incluye los números de línea; el primer número de línea indica el punto de lanzamiento y los subsiguientes números de línea indican las ubicaciones desde las que se llamaron los métodos en el rastreo de pila. El IDE crea los archivos PDB para mantener la información de depuración para los proyectos que cree.

Propiedad `InnerException`

Otra propiedad que utilizan con frecuencia los programadores de bibliotecas de clases es **`InnerException`**. Por lo general, los programadores de bibliotecas de clases “envuelven” los objetos de excepciones que se atrapan en su código, para que entonces puedan lanzar nuevos tipos de excepciones que sean específicos para sus bibliotecas.

Por ejemplo, un programador que implemente un sistema de contabilidad podría tener cierto código de procesamiento de números de cuenta, en el cual los números de cuenta se introduzcan como objetos `string`, pero se representen como objetos `int` en el código. Recuerde que un programa puede convertir valores `string` en valores `int` mediante `Convert.ToInt32`, el cual lanza una excepción `FormatException` cuando encuentra un formato de número inválido. Cuando se produce un formato de número de cuenta inválido, el programador del sistema de contabilidad tal vez desee emplear un mensaje de error distinto al mensaje predeterminado suministrado por `FormatException`, o tal vez desee indicar un nuevo tipo de excepción, como `NumeroCuentaInvalidoFormatException`. En tales casos, el programador proporcionaría código para atrapar la excepción `FormatException` y después crearía un tipo apropiado de objeto `Exception` en el bloque `catch`, y pasaría la excepción original como uno de los argumentos para el constructor. El objeto excepción original se convierte en la excepción interior (`InnerException`) del nuevo objeto excepción. Cuando ocurre una excepción `NumeroCuentaInvalidoFormatException` en el código que utiliza la biblioteca del sistema de contabilidad, el bloque `catch` que atrapa la excepción puede obtener una referencia a la excepción original, a través de la propiedad `InnerException`. Por ende, la excepción indica que el usuario específico un número de cuenta inválido y que el problema era un formato de número inválido. Si la propiedad `InnerException` es `null`, esto indica que la excepción no se produjo por otra excepción.

Otras propiedades de Exception

La clase `Exception` tiene otras propiedades, incluyendo **`HelpLink`**, **`Source`** y **`TargetSite`**. La propiedad `HelpLink` especifica la ubicación del archivo de ayuda que describe el problema que ocurrió. Esta propiedad es `null` si no existe dicho archivo. La propiedad `Source` especifica el nombre de la aplicación en la que ocurrió la excepción. La propiedad `TargetSite` especifica el método en donde se originó la excepción.

Demostración de las propiedades de Exception y la limpieza de la pila

Nuestro siguiente ejemplo (figura 12.5) demuestra las propiedades `Message`, `StackTrace` e `InnerException`, y el método `ToString` de la clase `Exception`. Además, el ejemplo introduce el concepto de *limpieza de la pila*: cuando se lanza una excepción pero no se atrapa en un alcance específico, la pila de llamadas a métodos se “limpia” y se hace un intento de atrapar (`catch`) la excepción en el siguiente bloque `try` exterior. Llevaremos el registro de los métodos en la pila de llamadas a medida que hablemos sobre la propiedad `StackTrace` y el mecanismo de limpieza de la pila. Para ver el rastreo de pila apropiado, debe ejecutar este programa utilizando pasos similares a los que presentamos en la sección 12.3.

```

1  // Fig. 12.5: Propiedades.cs
2  // Limpieza de la pila y las propiedades de la clase Exception.
3  // Demuestra el uso de las propiedades Message, StackTrace e InnerException.
4  using System;
5
6  class Propiedades
7  {
8      static void Main()
9      {
10         // llama a Metodo1; cualquier Exception que se genere
11         // se atrapa en el bloque catch que le sigue
12         try
13         {
14             Metodo1();
15         } // fin de try
16         catch ( Exception parametroExcepcion )
17         {
18             // imprime en pantalla la representación string de Exception, después imprime
19             // en pantalla las propiedades InnerException, Message y StackTrace
20             Console.WriteLine( "parametroExcepcion.ToString: \n{0}\n",
21                               parametroExcepcion.ToString() );

```

Figura 12.5 | Las propiedades de `Exception` y la limpieza de la pila. (Parte 1 de 3).

```

22         Console.WriteLine( "parametroExcepcion.Message: \n{0}\n",
23                             parametroExcepcion.Message );
24         Console.WriteLine( "parametroExcepcion.StackTrace: \n{0}\n",
25                             parametroExcepcion.StackTrace );
26         Console.WriteLine( "parametroExcepcion.InnerException: \n{0}\n",
27                             parametroExcepcion.InnerException.ToString() );
28     } // fin de catch
29 } // fin del método Main
30
31 // llama a Metodo2
32 static void Metodo1()
33 {
34     Metodo2();
35 } // fin del método Metodo1
36
37 // llama a Metodo3
38 static void Metodo2()
39 {
40     Metodo3();
41 } // fin del método Metodo2
42
43 // lanza una Exception que contiene una InnerException
44 static void Metodo3()
45 {
46     // trata de convertir string en int
47     try
48     {
49         Convert.ToInt32( "No es un entero" );
50     } // fin de try
51     catch ( FormatException parametroFormatException )
52     {
53         // envuelve a FormatException en nueva Exception
54         throw new Exception( "Ocurrió una excepción en Metodo3",
55                             parametroFormatException );
56     } // fin de catch
57 } // fin del método Metodo3
58 } // fin de la clase Propiedades

```

```

parametroExcepcion.ToString:
System.Exception: Ocurrió una excepción en Metodo3 --->
  System.FormatException: La cadena de entrada no tiene el formato correcto.
    en System.Number.StringToNumber(String str, NumberStyles options,
    NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
    en System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
    en System.Convert.ToInt32(String value)
    en Propiedades.Metodo3() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 49
--- Fin del seguimiento de la pila de la excepción interna ---
    en Propiedades.Metodo3() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 54
    en Propiedades.Metodo2() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 40
    en Propiedades.Metodo1() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 34
    en Propiedades.Main() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 14

```

Figura 12.5 | Las propiedades de Exception y la limpieza de la pila. (Parte 2 de 3).


```

parametroExcepcion.Message:
Ocurrió una excepción en Metodo3

parametroExcepcion.StackTrace:
  en Propiedades.Metodo3() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 54
  en Propiedades.Metodo2() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 40
  en Propiedades.Metodo1() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 34
  en Propiedades.Main() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 14

parametroExcepcion.InnerException:
System.FormatException: La cadena de entrada no tiene el formato correcto.
  en System.Number.StringToNumber(String str, NumberStyles options,
    NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
  en System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
  en System.Convert.ToInt32(String value)
  en Propiedades.Metodo3() en C:\ejemplos\cap12\Fig12_04\Propiedades\
    Propiedades.cs:línea 49

```

Figura 12.5 | Las propiedades de Exception y la limpieza de la pila. (Parte 3 de 3).

La ejecución del programa empieza con Main, que se convierte en el primer método en la pila de llamadas a métodos. La línea 14 del bloque try en Main invoca al Metodo1 (declarado en las líneas 32-35), el cual se convierte en el segundo método en la pila. Si Metodo1 lanza una excepción, el bloque catch en las líneas 16-28 maneja la excepción e imprime en pantalla información acerca de la excepción que ocurrió. La línea 34 de Metodo1 invoca a Metodo2 (líneas 38-41), el cual se convierte en el tercer método en la pila. Después la línea 40 de Metodo2 invoca a Metodo3 (líneas 44-57), el cual se convierte en el cuarto método en la pila.

En este punto, la pila de llamadas a métodos (de arriba-abajo) para el programa es:

```

Metodo3
Metodo2
Metodo1
Main

```

El método que se llamó al último (Metodo3) aparece en la parte superior de la pila; el primer método que se llamó (Main) aparece en la parte inferior. La instrucción try (líneas 47-56) en Metodo3 invoca al método Convert.ToInt32 (línea 49), el cual trata de convertir un valor string en un valor int. En este punto, Convert.ToInt32 se convierte en el método 50, y final, de la pila.

Lanzar una Exception con una InnerException

Debido a que el argumento para Convert.ToInt32 no está en formato int, la línea 49 lanza una excepción FormatException que se atrapa en la línea 51 de Metodo3. La excepción termina la llamada a Convert.ToInt32, por lo que el método se elimina (o limpia) de la pila de llamadas a métodos. Después, el bloque catch en Metodo3 crea y lanza un objeto Exception. El primer argumento para el constructor de Exception es el mensaje de error personalizado para nuestro ejemplo, “Ocurrió una excepción en Metodo3”. El segundo argumento es InnerException: la excepción FormatException que se atrapó. El valor de StackTrace para este nuevo objeto excepción refleja el punto en el cual se lanzó la excepción (líneas 54-55). Ahora Metodo3 termina, ya que la excepción que se lanzó en el bloque catch no se atrapó en el cuerpo del método. Por ende, el control regresa a la instrucción que invocó a Metodo3 en el método anterior en la pila de llamadas (Metodo2). Esto elimina, o *limpia*, a Metodo3 de la pila de llamadas a métodos.

Cuando el control regresa a la línea 40 en Metodo2, el CLR determina que la línea 40 no está dentro de un bloque try. Por lo tanto, la excepción no se puede atrapar en Metodo2 y éste termina. Esto limpia a Metodo2 de la pila de llamadas y se regresa el control a la línea 28 en Metodo1.

Aquí podemos ver otra vez que la línea 34 no está dentro de un bloque try, por lo que Metodo1 no puede atrapar la excepción. El método termina y se limpia de la pila de llamadas, regresando el control a la línea 14 en Main, que *está* ubicado dentro de un bloque try. El bloque try en Main expira y el bloque catch (líneas 16-28) atrapa la excepción. El bloque catch utiliza el método ToString y las propiedades Message, StackTrace e InnerException para crear la salida. Observe que la limpieza de la pila continúa hasta que un bloque catch atrape la excepción, o cuando termine el programa.

Mostrar información acerca de la excepción *Exception*

El primer bloque de salida (le dimos formato otra vez, para mejorar la legibilidad) en la figura 12.5 contiene la representación string de la excepción, que devuelve el método ToString. El objeto string empieza con el nombre de la clase de la excepción, seguido del valor de la propiedad Message. Los siguientes cuatro elementos presentan el rastreo de pila del objeto InnerException. El resto del bloque de resultado muestra la propiedad StackTrace para la excepción que se lanza en Metodo3. Observe que StackTrace representa el estado de la pila de llamadas a métodos en el punto de lanzamiento de la excepción, en vez de que sea el punto en donde se atraparé la excepción, en un momento dado. Cada línea de StackTrace que empieza con “en” representa un método en la pila de llamadas. Estas líneas indican el método en el que ocurrió la excepción, el archivo en el cual reside el método y el número de línea del punto de lanzamiento en el archivo. Observe que la información de la excepción interior incluye el rastreo de pila de esta excepción.



Tip de prevención de errores 12.6

Al atrapar y volver a lanzar una excepción, debe proporcionar información de depuración adicional en la excepción que se volvió a lanzar. Para ello, cree un objeto Exception que contenga información de depuración más específica, y después pase la excepción original que se atrapó al constructor del nuevo objeto excepción, para inicializar la propiedad InnerException.

El siguiente bloque de resultados (dos líneas) sólo muestra en pantalla el valor de la propiedad Message (Ocurrió una excepción en Metodo3) de la excepción que se lanzó en Metodo3.

El tercer bloque de resultados muestra en pantalla la propiedad StackTrace de la excepción que se lanzó en Metodo3. Observe que esta propiedad StackTrace contiene el rastreo de pila, empezando desde la línea 54 en Metodo3, ya que es el punto en el que se creó y se lanzó el objeto Exception. El rastreo de pila siempre empieza desde el punto de lanzamiento de la excepción.

Para terminar, el último bloque de resultados muestra en pantalla la representación string de la propiedad InnerException, la cual incluye el espacio de nombres y el nombre de clase del objeto excepción, así como las propiedades Message y StackTrace.

12.8 Clases de excepciones definidas por el usuario

En muchos casos, puede utilizar las clases de excepciones existentes de la Biblioteca de clases del .NET Framework para indicar las excepciones que ocurren en sus programas. No obstante, en algunos casos tal vez sea conveniente crear nuevas clases de excepciones que sean específicas para los problemas que ocurran en sus programas. Las *clases de excepciones definidas por el usuario* deben derivarse de manera directa o indirecta de la clase ApplicationException, del espacio de nombres System.



Buena práctica de programación 12.1

Asociar cada tipo de falla con una clase de excepción con un nombre apropiado mejora la legibilidad del programa.



Observación de ingeniería de software 12.3

Antes de crear una clase de excepción definida por el usuario, investigue las excepciones existentes en la Biblioteca de clases del .NET Framework para determinar si ya existe un tipo apropiado de excepción.

Las figuras 12.6 y 12.7 demuestran el uso de una clase de excepción definida por el usuario. La clase NumeroNegativoException (figura 12.6) es una clase de excepción definida por el usuario, la cual representa excepciones que ocurren cuando un programa realiza una operación ilegal con un número negativo, como tratar de calcular su raíz cuadrada.

```

1  // Fig. 12.6: NumeroNegativoException.cs
2  // NumeroNegativoException representa las excepciones producidas por
3  // operaciones ilegales con números negativos.
4  using System;
5
6  namespace PruebaRaizCuadrada
7  {
8      class NumeroNegativoException : ApplicationException
9      {
10         // constructor predeterminado
11         public NumeroNegativoException()
12             : base( "Operación ilegal para un número negativo" )
13         {
14             // cuerpo vacío
15         } // fin del constructor predeterminado
16
17         // constructor para personalizar el mensaje de error
18         public NumeroNegativoException( string valorMensaje )
19             : base( valorMensaje )
20         {
21             // cuerpo vacío
22         } // fin del constructor con un argumento
23
24         // constructor para personalizar el mensaje de error
25         // de la excepción y especificar el objeto InnerException
26         public NumeroNegativoException( string valorMensaje,
27             Exception exterior )
28             : base( valorMensaje, exterior )
29         {
30             // cuerpo vacío
31         } // fin del constructor con dos argumentos
32     } // fin de la clase NumeroNegativoException
33 } // fin del espacio de nombres PruebaRaizCuadrada

```

Figura 12.6 | La clase derivada de `ApplicationException`, que se lanza cuando un programa realiza una operación ilegal con un número negativo.

De acuerdo con las “Mejores prácticas para el manejo de excepciones [C#]”, las excepciones definidas por el usuario deben extender la clase `ApplicationException`, deben tener un nombre de clase que termine con “Exception” y deben definir tres constructores: un constructor sin parámetros; un constructor que reciba un argumento `string` (el mensaje de error); y un constructor que reciba un argumento `string` y un argumento `Exception` (el mensaje de error y el objeto excepción interior). Si define estos tres constructores, su clase de excepción será más flexible, lo que permitirá que otros programadores puedan usarla y extenderla con facilidad.

Las excepciones del tipo `NumeroNegativoException` ocurren con más frecuencia durante las operaciones aritméticas, por lo que parece lógico derivar la clase `NumeroNegativoException` de la clase `ArithmeticException`. No obstante, la clase `ArithmeticException` se deriva de la clase `SystemException`: la categoría de excepciones que lanza el CLR. Recuerde que las clases de excepciones definidas por el usuario deben heredar de `ApplicationException`, en vez de `SystemException`.

La clase `RaizCuadradaForm` (figura 12.7) demuestra nuestra clase de excepción definida por el usuario. La aplicación permite al usuario introducir un valor numérico y después invoca al método `RaizCuadrada` (líneas 17-25) para calcular la raíz cuadrada de ese valor. Para realizar este cálculo, `RaizCuadrada` invoca al método `Sqrt` de la clase `Math`, el cual recibe un valor `double` como argumento. Por lo general, si el argumento es negativo, el método `Sqrt` devuelve `NaN`. En este programa nos gustaría evitar que el usuario calculara la raíz cuadrada de un número negativo. Si el valor numérico que introduce el usuario es negativo, el método `RaizCuadrada` lanza una excepción `NumeroNegativoException` (líneas 21-22). De no ser así, `RaizCuadrada` invoca al método `Sqrt` de la clase `Math` para calcular la raíz cuadrada (línea 24).

Cuando el usuario introduce un valor y hace clic en el botón **Raíz cuadrada**, el programa invoca al manejador de eventos `RaízCuadradaButton_Click` (líneas 28-53). La instrucción `try` (líneas 33-52) intenta invocar a `RaízCuadrada` usando el valor introducido por el usuario. Si la entrada del usuario no es un número válido, se produce una excepción `FormatException` y el bloque `catch` en las líneas 40-45 procesa la excepción. Si el usuario introduce un número negativo, el método `RaízCuadrada` lanza una excepción `NumeroNegativoException` (líneas 21-22); el bloque `catch` en las líneas 46-52 atrapa y maneja este tipo de excepción.

```

1  // Fig. 12.7: PruebaRaízCuadrada.cs
2  // Demostración de una clase de excepción definida por el usuario.
3  using System;
4  using System.Windows.Forms;
5
6  namespace PruebaRaízCuadrada
7  {
8      public partial class RaízCuadradaForm : Form
9      {
10         public RaízCuadradaForm()
11         {
12             InitializeComponent();
13         } // fin del constructor
14
15         // calcula la raíz cuadrada del parámetro; lanza excepción
16         // NumeroNegativoException si el parámetro es negativo
17         public double RaízCuadrada( double valor )
18         {
19             // si el operando es negativo, lanza NumeroNegativoException
20             if ( valor < 0 )
21                 throw new NumeroNegativoException(
22                     "No se permite la raíz cuadrada de un número negativo" );
23             else
24                 return Math.Sqrt( valor ); // calcula raíz cuadrada
25         } // fin del método RaízCuadrada
26
27         // obtiene la entrada del usuario, convierte en double, calcula la raíz cuadrada
28         private void RaízCuadradaButton_Click( object sender, EventArgs e )
29         {
30             SalidaLabel.Text = ""; // borra SalidaLabel
31
32             // atrapa cualquier excepción NumeroNegativoException que se lance
33             try
34             {
35                 double resultado =
36                     RaízCuadrada( Convert.ToDouble( EntradaTextBox.Text ) );
37
38                 SalidaLabel.Text = resultado.ToString();
39             } // fin de try
40             catch ( FormatException parametroFormatException )
41             {
42                 MessageBox.Show( parametroFormatException.Message,
43                     "Formato de número inválido", MessageBoxButtons.OK,
44                     MessageBoxIcon.Error );
45             } // fin de catch
46             catch ( NumeroNegativoException
47                 parametroNumeroNegativoException )
48             {
49                 MessageBox.Show( parametroNumeroNegativoException.Message,

```

Figura 12.7 | La clase `RaízCuadradaForm` lanza una excepción si ocurre un error al calcular la raíz cuadrada. (Parte 1 de 2).

```

50         "Operación inválida", MessageBoxButtons.OK,
51         MessageBoxIcon.Error );
52     } // fin de catch
53 } // fin del método RaizCuadradaButton_Click
54 } // fin de la clase RaizCuadradaForm
55 } // fin del espacio de nombres PruebaRaizCuadrada

```

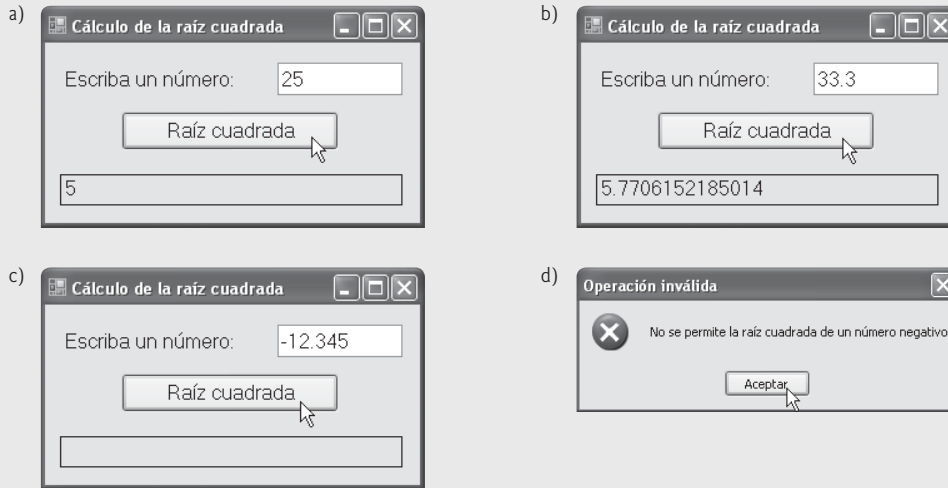


Figura 12.7 | La clase `RaizCuadradaForm` lanza una excepción si ocurre un error al calcular la raíz cuadrada. (Parte 2 de 2).

12.9 Conclusión

En este capítulo aprendió a utilizar el manejo de excepciones para lidiar con los errores en una aplicación. Demostramos que el manejo de las excepciones le permite eliminar el código para manejar errores de la “línea principal” de ejecución del programa. Vio el manejo de excepciones en el contexto de un ejemplo de división entre cero. Aprendió a utilizar bloques `try` para encerrar el código que podría lanzar una excepción, y cómo utilizar bloques `catch` para lidiar con las excepciones que puedan surgir. Hablamos sobre el modelo de terminación del manejo de excepciones, en el cual después de manejar una excepción, el control del programa no regresa al punto de lanzamiento. También vimos varias clases importantes de la jerarquía `Exception` de .NET, incluyendo `ApplicationException` (a partir de la cual se derivan las clases de excepciones definidas por el usuario) y `SystemException`. Después aprendió a utilizar el bloque `finally` para liberar recursos en caso de que ocurra o no una excepción, y cómo lanzar y volver a lanzar excepciones mediante la instrucción `throw`. Hablamos además sobre cómo puede usarse la instrucción `using` para automatizar el proceso de liberar un recurso. Más adelante aprendió a obtener información acerca de una excepción, mediante el uso de las propiedades `Message`, `StackTrace` e `InnerException` de la clase `Exception`, junto con el método `ToString`. Aprendió a crear sus propias clases de excepciones. En los siguientes dos capítulos, presentaremos un tratamiento detallado de las interfaces gráficas de usuario. En estos capítulos y durante el resto de este libro, usaremos el manejo de excepciones para que nuestros ejemplos sean más robustos, al mismo tiempo que se demuestran las nuevas características del lenguaje.

13

Conceptos de interfaz gráfica de usuario: parte I

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Los principios de diseño de las interfaces gráficas de usuario (GUIs).
- Crear interfaces gráficas de usuario.
- Procesar eventos que se generen mediante las interacciones del usuario con los controles de la GUI.
- Acerca de los espacios de nombres que contienen las clases para los controles de la interfaz gráfica de usuario y el manejo de los eventos.
- Crear y manipular controles `Button`, `Label`, `RadioButton`, `CheckBox`, `TextBox`, `Panel` y `NumericUpDown`.
- Agregar cuadros de información de la herramienta (`ToolTip`) descriptivos a los controles de la GUI.
- Procesar eventos del ratón y del teclado.

*...los profetas más sabios
se aseguran primero del evento.*

—Horace Walpole

*... El usuario debe sentirse en
control de la computadora;
no al revés. Esto se logra en
aplicaciones que abarcan
tres cualidades: sensibilidad,
permisividad y consistencia.*

—Apple Computer, Inc. 1985

Para verte mejor, querida.

—El Gran Lobo Malo a
Caperucita Roja

- 13.1 Introducción
- 13.2 Formularios Windows Forms
- 13.3 Manejo de eventos
 - 13.3.1 Una GUI simple controlada por eventos
 - 13.3.2 Otro vistazo al código generado por Visual Studio
 - 13.3.3 Delegados y el mecanismo de manejo de eventos
 - 13.3.4 Otras formas de crear manejadores de eventos
 - 13.3.5 Localización de la información de los eventos
- 13.4 Propiedades y distribución de los controles
- 13.5 Controles Label, TextBox y Button
- 13.6 Controles GroupBox y Panel
- 13.7 Controles CheckBox y RadioButton
- 13.8 Controles PictureBox
- 13.9 Controles ToolTip
- 13.10 Control NumericUpDown
- 13.11 Manejo de los eventos del ratón
- 13.12 Manejo de los eventos del teclado
- 13.13 Conclusión

13.1 Introducción

Una interfaz gráfica de usuario (GUI) permite a un usuario interactuar con un programa en forma visual; además, proporciona a un programa una “apariencia visual” única. Al proporcionar distintas aplicaciones en las que los componentes de la interfaz de usuario sean consistentes e intuitivos, los usuarios pueden conocer con mayor rapidez las aplicaciones y así volverse más productivos.



Observación de apariencia visual 13.1

Las interfaces de usuario consistentes permiten aprender nuevas aplicaciones con más rapidez, ya que éstas tienen la misma “apariencia visual”.

Como ejemplo de GUI, considere la figura 13.1, que muestra una ventana del navegador Web Internet Explorer que contiene varios controles. Cerca de la parte superior de la ventana hay una barra de menús que muestra los menús **Archivo**, **Edición**, **Ver**, **Favoritos**, **Herramientas** y **Ayuda**. Debajo del menú hay un conjunto de botones, cada uno de los cuales tiene una tarea definida, como regresar a la página Web anterior, imprimir la página actual o actualizarla. Debajo de estos botones hay un cuadro combinado, en el cual, los usuarios pueden escribir las ubicaciones de los sitios Web que desean visitar. A la izquierda del cuadro combinado hay una etiqueta (**Dirección**) que indica el propósito del cuadro combinado (en este caso, escribir la ubicación de un sitio Web). En la parte derecha e inferior de la ventana hay barras de desplazamiento. Por lo general, estas barras aparecen cuando una ventana contiene más información de la que puede desplegar en su área visible. Las barras de desplazamiento permiten que el usuario vea distintas porciones del contenido de la ventana. Estos controles forman una interfaz amigable al usuario, a través de la cual éste interactúa con el navegador Web Internet Explorer.

Las GUIs se crean a partir de controles de la GUI (conocidos como *componentes* o *widgets* [accesorios de ventana]). Los controles de la GUI son objetos que pueden mostrar información en la pantalla o que permiten a los usuarios interactuar con una aplicación a través del ratón, del teclado o de alguna otra forma de entrada (como los comandos de voz). En la figura 13.2 se listan varios controles comunes de la GUI; en las subsiguientes secciones y en el capítulo 14 hablaremos detalladamente sobre cada uno de estos controles. En el capítulo 14 también se exploran las características y propiedades de controles adicionales de la GUI.

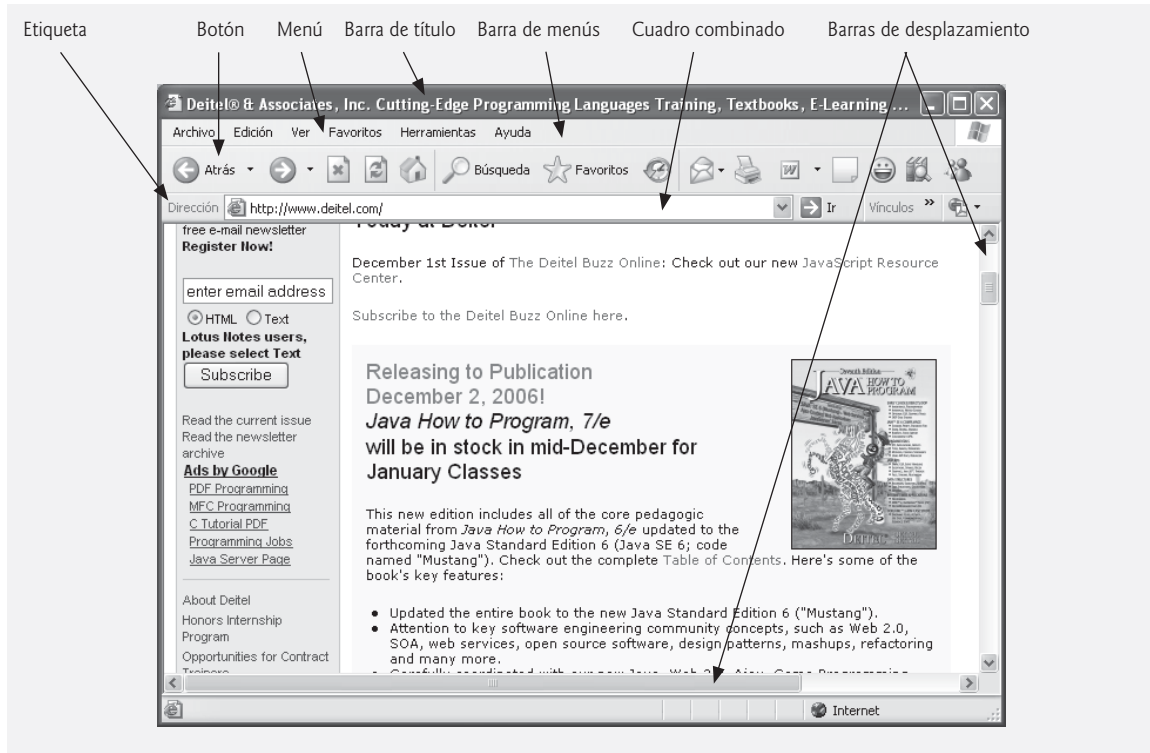


Figura 13.1 | Controles de la GUI en una ventana de Internet Explorer.

Control	Descripción
Label	Muestra imágenes o texto que no puede editarse.
TextBox	Permite al usuario introducir datos mediante el teclado. También puede usarse para mostrar texto que puede o no editarse.
Button	Activa un evento cuando se hace clic con el ratón.
CheckBox	Especifica una opción que puede seleccionarse (activada) o deseleccionarse (desactivada).
ComboBox	Proporciona una lista desplegable de elementos, de los cuales el usuario puede seleccionar uno, haciendo clic en un elemento de la lista o escribiendo dentro de un cuadro.
ListBox	Proporciona una lista de elementos, de los cuales el usuario puede seleccionar uno, haciendo clic en un elemento de la lista. Pueden seleccionarse varios elementos a la vez.
Panel	Un contenedor en el que pueden colocarse y organizarse controles.
NumericUpDown	Permite al usuario seleccionar de un rango de valores de entrada.

Figura 13.2 | Algunos controles básicos de la GUI.

13.2 Formularios Windows Forms

Los *formularios Windows Forms* se utilizan para crear las GUIs para los programas. Un formulario (Form) es un elemento gráfico que aparece en el escritorio de su computadora; puede ser un cuadro de diálogo, una ventana o una *ventana MDI* (*ventana de interfaz de múltiples documentos*), que veremos en el capítulo 14, Conceptos de interfaz gráfica de usuario: parte 2. Un *componente* es una instancia de una clase que implementa a la *interfaz IComponent*, la cual define los comportamientos que deben implementar los componentes, como la forma en que se debe cargar el componente. Un control como Button o Label tiene una representación gráfica en

tiempo de ejecución. Algunos componentes carecen de representaciones gráficas. (Por ejemplo, la clase `Timer` del espacio de nombres `System.Windows.Forms`; vea el capítulo 14). Dichos componentes no son visibles en tiempo de ejecución.

La figura 13.3 muestra los controles y componentes `Windows Forms` del **Cuadro de herramientas** de C#. Los controles y componentes se organizan en categorías, con base en su funcionalidad. Si selecciona la categoría **Todos los formularios Windows Forms** en la parte superior del **Cuadro de herramientas**, podrá ver todos los controles y componentes de las demás fichas en una sola lista (como se muestra en la figura 13.3). En este capítulo y en el siguiente, hablaremos sobre muchos de estos controles y componentes. Para agregar un control o componente a un formulario, seleccione ese control o componente del **Cuadro de herramientas** y arrástrelo en el formulario. Para deseleccionar un control o componente, seleccione el elemento **Puntero** en el **Cuadro de herramientas** (el icono en la parte superior de la lista). Cuando se selecciona el elemento **Puntero**, no se puede agregar de manera accidental un nuevo control al formulario.

Cuando hay varias ventanas en la pantalla, la **ventana activa** es la que está al frente y tiene su barra de título resaltada; por lo general de un color azul más oscuro que el de las otras ventanas en la pantalla. Una ventana se convierte en la ventana activa cuando el usuario hace clic en alguna parte dentro de ella. Se dice que la ventana activa “tiene el *foco*”. Por ejemplo, en Visual Studio la ventana activa es el **Cuadro de herramientas** cuando se selecciona un elemento de ella, o la ventana **Propiedades** cuando estamos editando las propiedades de un control.

Un formulario es un *contenedor* de controles y componentes. Cuando usted arrastra un control o componente del **Cuadro de herramientas** hacia el formulario, Visual Studio genera código que crea una instancia del objeto y establece sus propiedades básicas. Este código se actualiza cuando se modifican las propiedades del control o componente en el IDE. Si se elimina un control o componente del formulario, se elimina el código generado para ese control. El IDE coloca el código generado en un archivo separado, usando clases parciales. Aunque podríamos escribir este código por nuestra cuenta, es mucho más fácil crear y modificar controles y componentes utilizando las ventanas **Cuadro de herramientas** y **Propiedades**, y permitir que Visual Studio se encargue de los detalles. En el

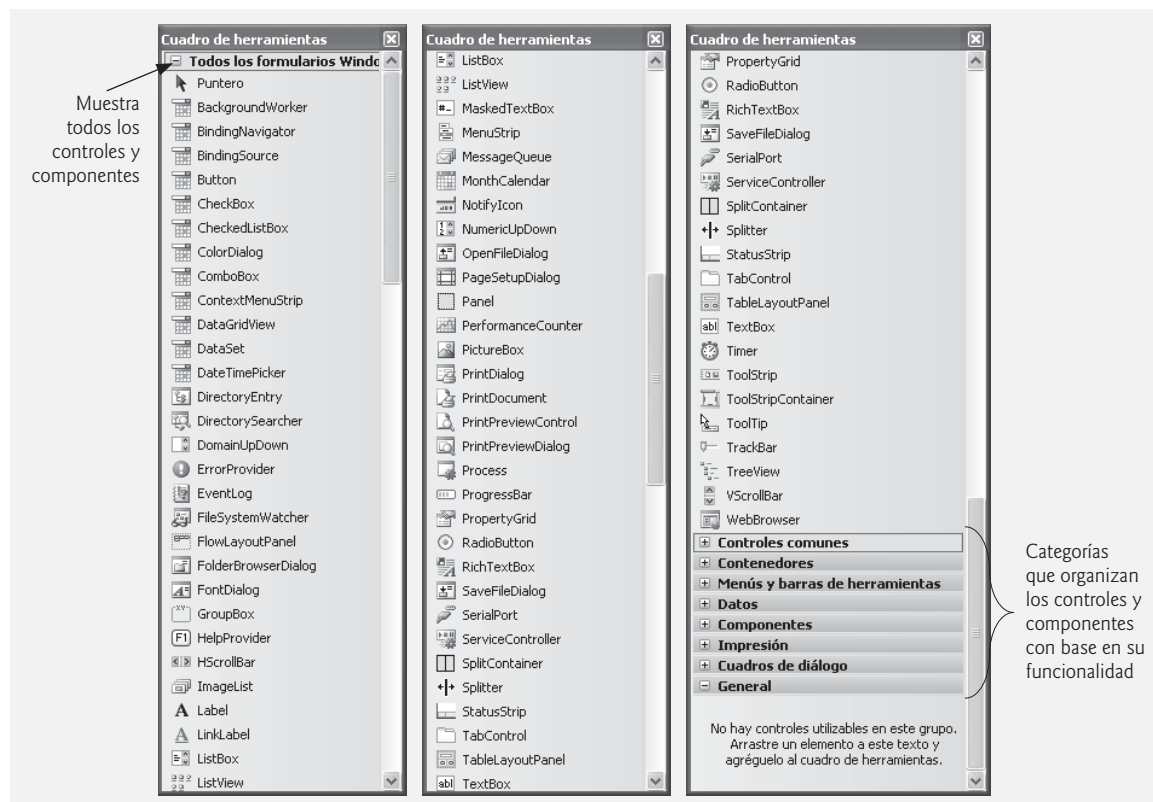


Figura 13.3 | Componentes y controles para formularios `Windows Forms`.

Propiedades, métodos y eventos de Form	Descripción
<i>Propiedades comunes</i>	
AcceptButton	Control Button en el que se hace clic al oprimir <i>Intro</i> .
AutoScroll	Valor boolean que activa o desactiva las barras de desplazamiento, según sea necesario.
CancelButton	Control Button en el que se hace clic al oprimir <i>Esc</i> .
FormBorderStyle	Estilo de borde para el formulario. (Por ejemplo: ninguno, simple o tridimensional).
Font	La fuente de texto que se muestra en el formulario, y la fuente predeterminada para los controles que se agregan al formulario.
Text	El texto en la barra de título del formulario.
<i>Métodos comunes</i>	
Close	Cierra un formulario y libera todos los recursos, como la memoria utilizada para los controles y componentes del formulario. Un formulario cerrado no puede volver a abrirse.
Hide	Oculto un formulario, pero no lo destruye ni libera sus recursos.
Show	Muestra un formulario oculto.
<i>Eventos comunes</i>	
Load	Ocurre antes de que se muestre un formulario al usuario. El manejador para este evento se muestra en el editor de Visual Studio al hacer doble clic en el formulario, en el diseñador de Visual Studio.

Figura 13.4 | Propiedades, métodos y eventos comunes del control Form.

capítulo 2 presentamos los conceptos de la programación visual. En este capítulo y en el siguiente, utilizaremos la programación visual para crear GUIs más robustas.

Cada control o componente que presentamos en este capítulo se encuentra en el espacio de nombres `System.Windows.Forms`. Para crear una aplicación de Windows, por lo general se crea un formulario, se establecen sus propiedades, se agregan controles al formulario, se establecen las propiedades de estos controles y se implementan manejadores de eventos (métodos) que respondan a los eventos generados por los controles. La figura 13.4 lista las propiedades, métodos y eventos comunes del control Form.

Cuando creamos controles y manejadores de eventos, Visual Studio genera la mayor parte del código relacionado con la GUI. En la programación visual, el IDE se encarga de mantener el código relacionado con la GUI y usted escribe los cuerpos de los manejadores de eventos, para indicar qué acciones debe realizar el programa cuando ocurren ciertos eventos.

13.3 Manejo de eventos

Por lo general, un usuario interactúa con la GUI de una aplicación para indicar las tareas que ésta debe realizar. Por ejemplo, cuando usted escribe un correo electrónico en una aplicación para correo electrónico, al hacer clic en el botón **Enviar** indica a la aplicación que debe enviar el correo electrónico a las direcciones de correo electrónico especificadas. Las GUIs son *controladas por eventos*. Cuando el usuario interactúa con un componente de la GUI, la interacción (conocida como *evento*) controla el programa para que realice una tarea. Los eventos comunes (interacciones del usuario) que podrían hacer que una aplicación realice una tarea incluyen el hacer clic en un control Button, escribir en un control TextBox, seleccionar un elemento de un menú, cerrar una ventana y mover el ratón. A un método que realiza una tarea en respuesta a un evento se le conoce como *manejador de eventos*, y al proceso general de responder a los eventos se le conoce como *manejo de eventos*.

13.3.1 Una GUI simple controlada por eventos

El formulario (Form) en la aplicación de la figura 13.5 contiene un control Button en el que puede hacer clic el usuario para mostrar un control MessageBox. Ya hemos creado varios ejemplos de la GUI que ejecutan un manejador de eventos, en respuesta al clic de un control Button. En este ejemplo veremos el código generado en forma automática por Visual Studio con más detalle.

Usando las técnicas que presentamos en capítulos anteriores de este libro, cree un formulario que contenga un botón (Button). Primero, cree una aplicación nueva de Windows y agregue un botón al formulario. En la ventana **Propiedades** para el control Button, escriba en la propiedad (Name) el texto `clicButton` y en la propiedad Text el texto Haga clic. Como podrá observar, utilizamos una convención en la que el nombre de cada variable que creamos para un control, termina con el tipo del control. Por ejemplo, en la variable llamada `clicButton`, “Button” es el tipo del control.

Cuando el usuario hace clic en el control Button en este ejemplo, queremos que la aplicación responda mostrando un cuadro de diálogo MessageBox. Para ello, debemos crear un manejador de eventos para el evento Click del control Button. Para crear este manejador de eventos podemos hacer doble clic en el control Button dentro del formulario, el cual declara el siguiente manejador de eventos vacío en el código del programa:

```
private void clicButton_Click( object sender, EventArgs e )
{

} // fin del método clicButton_Click
```

Por convención, C# nombra al método manejador de eventos así: *nombreControl_nombreEvento*. (Por ejemplo, `clicButton_Click`). El manejador de eventos `clicButton_Click` se ejecuta cuando el usuario hace clic en el control `clicButton`.

Cuando se hace una llamada a un manejador de eventos, éste recibe dos parámetros. El primero (una referencia `object` llamada `sender`) es una referencia al objeto que generó el evento. El segundo es una referencia a un objeto de argumentos de evento de tipo `EventArgs` (o una de sus clases derivadas), que por lo general se llama `e`.

```
1 // Fig. 13.5: EjemploSimpleEventoForm.cs
2 // Uso de Visual Studio para crear manejadores de eventos.
3 using System;
4 using System.Windows.Forms;
5
6 // Formulario que muestra un manejador simple de eventos
7 public partial class EjemploSimpleEventoForm : Form
8 {
9     // constructor predeterminado
10    public EjemploSimpleEventoForm()
11    {
12        InitializeComponent();
13    } // fin del constructor
14
15    // maneja evento de clic del control Button clickButton
16    private void clicButton_Click( object sender, EventArgs e )
17    {
18        MessageBox.Show( "Se hizo clic en el botón." );
19    } // fin del método clicButton_Click
20 } // fin de la clase EjemploSimpleEventoForm
```

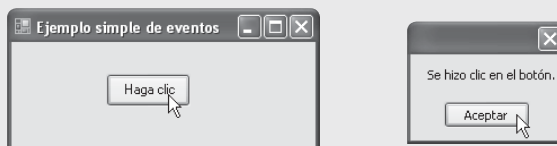


Figura 13.5 | Ejemplo simple de manejo de eventos, usando programación visual.

Este objeto contiene información adicional acerca del evento que ocurrió. EventArgs es la clase base de todas las clases que representan la información de un evento.



Observación de ingeniería de software 13.1

No debe esperar valores de retorno de los manejadores de eventos; éstos están diseñados para ejecutar código con base en una acción, y devuelven el control al programa principal.



Buena práctica de programación 13.1

Use la convención de nomenclatura de manejadores de eventos nombreControl_nombreEvento, de manera que los nombres de los eventos sean significativos. Dichos nombres indican a los usuarios cuál es el evento que maneja un método y para qué control. Esta convención no es obligatoria, pero mejora la legibilidad de su código y facilita su comprensión, modificación y mantenimiento.

Para mostrar un cuadro de diálogo MessageBox en respuesta al evento, inserte la instrucción

```
MessageBox.Show( "Se hizo clic en el botón." );
```

en el cuerpo del manejador del evento. El manejador de evento resultante aparece en las líneas 16-19 de la figura 13.5. Cuando ejecute la aplicación y haga clic en el botón (Button), aparecerá un cuadro de diálogo MessageBox en el que se mostrará el texto "Se hizo clic en el botón".

13.3.2 Otro vistazo al código generado por Visual Studio

Visual Studio genera el código para crear e inicializar la GUI que usted creará en la ventana de diseño de la GUI. Este código generado en forma automática se coloca en el archivo Designer.cs del control Form (EjemploSimpleEventoForm.Designer.cs en este ejemplo). Puede abrir este archivo expandiendo el nodo del archivo con el que se encuentra trabajando en estos momentos (EjemploSimpleEventoForm.cs) y haciendo doble clic en el nombre del archivo que termina con Designer.cs. Las figuras 13.6-13.7 muestran el contenido de este archivo. El IDE contrae el código en las líneas 21-53 de la figura 13.7 de manera predeterminada.

Ahora que hemos estudiado las clases y los objetos con detalle, este código le será más fácil de entender. Como Visual Studio crea y da mantenimiento a este código, por lo general no necesitamos analizarlo. De hecho, no nece-

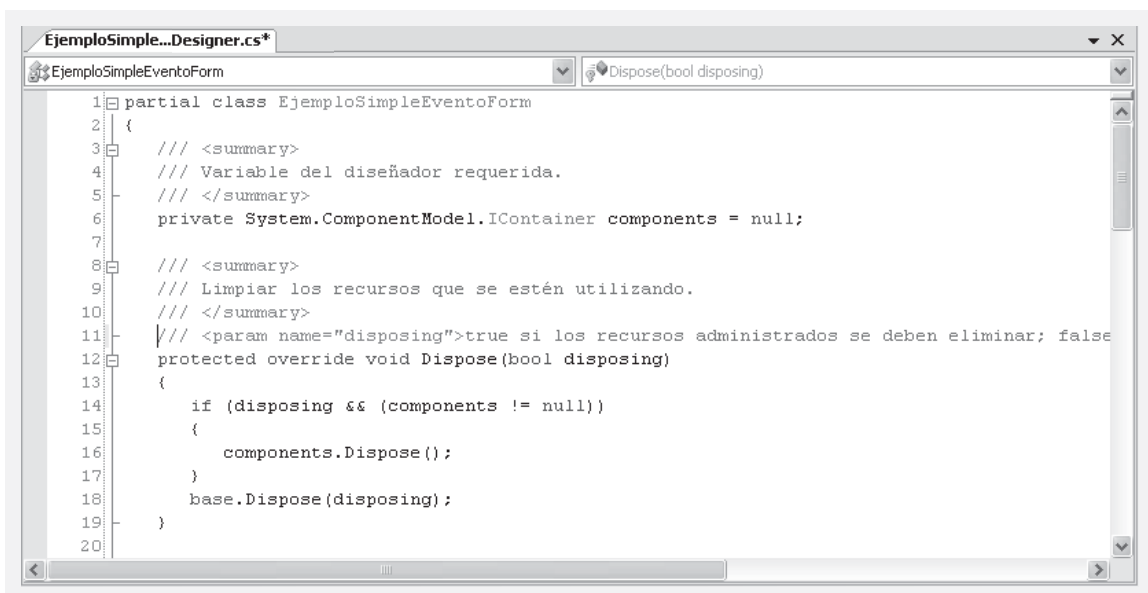
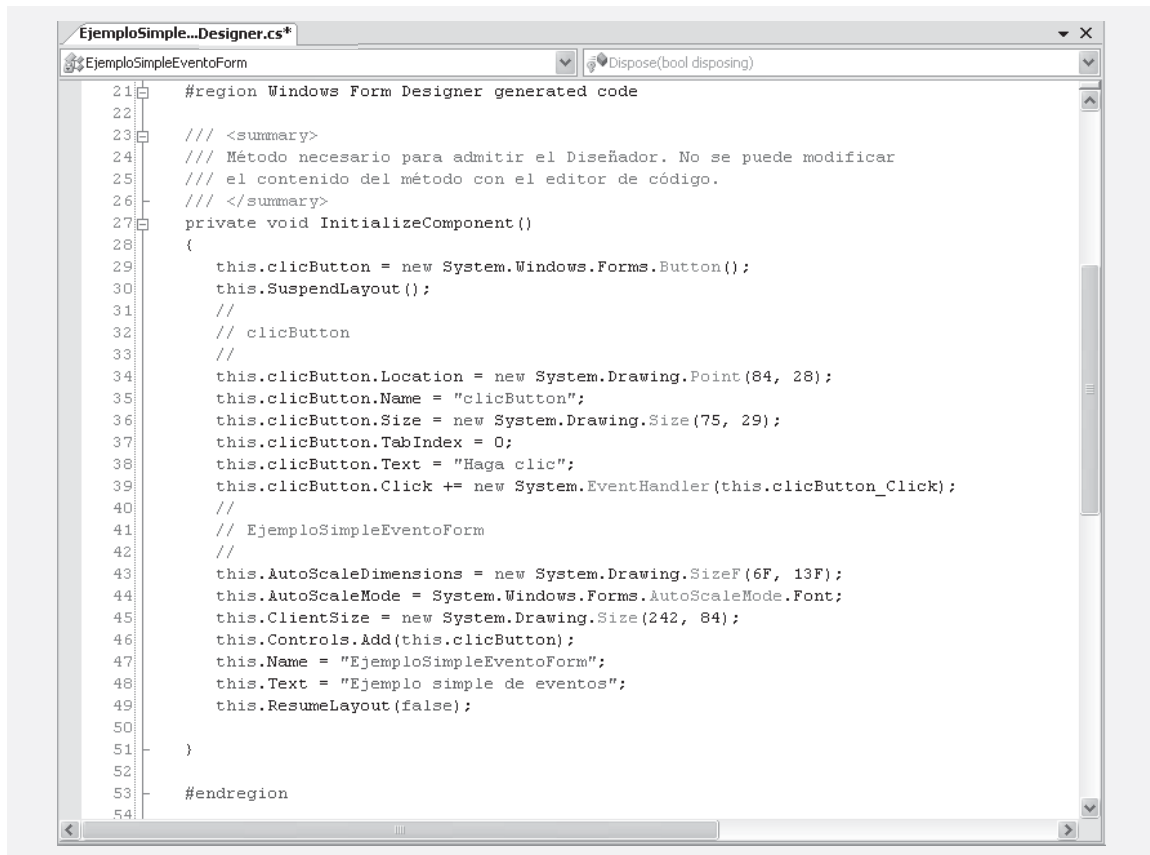


Figura 13.6 | Primera mitad del archivo de código generado por Visual Studio.



```

21 #region Windows Form Designer generated code
22
23 /// <summary>
24 /// Método necesario para admitir el Diseñador. No se puede modificar
25 /// el contenido del método con el editor de código.
26 /// </summary>
27 private void InitializeComponent()
28 {
29     this.clicButton = new System.Windows.Forms.Button();
30     this.SuspendLayout();
31     //
32     // clicButton
33     //
34     this.clicButton.Location = new System.Drawing.Point(84, 28);
35     this.clicButton.Name = "clicButton";
36     this.clicButton.Size = new System.Drawing.Size(75, 29);
37     this.clicButton.TabIndex = 0;
38     this.clicButton.Text = "Haga clic";
39     this.clicButton.Click += new System.EventHandler(this.clicButton_Click);
40     //
41     // EjemploSimpleEventoForm
42     //
43     this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
44     this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
45     this.ClientSize = new System.Drawing.Size(242, 84);
46     this.Controls.Add(this.clicButton);
47     this.Name = "EjemploSimpleEventoForm";
48     this.Text = "Ejemplo simple de eventos";
49     this.ResumeLayout(false);
50
51 }
52
53 #endregion
54

```

Figura 13.7 | Segunda mitad del archivo de código generado por Visual Studio.

sita comprender la mayor parte del código que se muestra aquí para crear aplicaciones con la GUI. No obstante, ahora lo analizaremos más de cerca para ayudarle a comprender cómo funcionan las aplicaciones con GUI.

El código generado en forma automática que define la GUI en realidad forma parte de la clase de Form; en este caso, `EjemploSimpleEventoForm`. La línea 1 de la figura 13.6 utiliza el modificador `partial`, el cual permite dividir esta clase entre varios archivos. La línea 55 contiene la declaración del control Button llamado `clicButton` que creamos en el modo de **Diseño**. Observe que el control se declara como una variable de instancia de la clase `EjemploSimpleEventoForm`. De manera predeterminada, todas las declaraciones de las variables para los controles que se crean mediante la ventana de diseño de C# tienen un modificador de acceso `private`. El código también incluye el método `Dispose` para liberar recursos (líneas 12-19) y el método `InitializeComponent` (líneas 27-51), que contiene el código para crear el control Button, y después establece algunas de las propiedades del control Button y del control Form. Los valores de las propiedades corresponden a los valores establecidos en la ventana **Propiedades** para cada control. Observe que Visual Studio agrega comentarios al código que genera, como en las líneas 31-33. La línea 39 se generó cuando creamos el manejador de eventos para el evento `Click` de Button.

El método `InitializeComponent` se llama cuando se crea el formulario, y establece propiedades tales como el título del formulario, su tamaño, los tamaños de los controles y el texto. Visual Studio también utiliza el código en este método para crear la GUI que vemos en la vista de diseño. Si cambia el código en `InitializeComponent`, podría evitar que Visual Studio desplegara la GUI en forma apropiada.



Tip de prevención de errores 13.1

*El código generado al crear una GUI en el modo de **Diseño** no debe modificarse en forma directa; si lo hace la aplicación podría funcionar en forma incorrecta. Debe modificar las propiedades de los controles a través de la ventana **Propiedades**.*

13.3.3 Delegados y el mecanismo de manejo de eventos

El control que genera un evento se conoce como el *emisor del evento*. Un método manejador de eventos (conocido como el *receptor del evento*) responde a un evento específico que genera un control. Cuando ocurre un evento, el emisor del evento llama al receptor de su evento para realizar una tarea (es decir, para “manejar el evento”).

El mecanismo para manejar eventos de .NET le permite elegir sus propios nombres para los métodos manejadores de eventos. No obstante, cada método manejador de eventos debe declarar los parámetros apropiados para recibir información acerca del evento que maneja. Como usted puede elegir sus propios nombres para los métodos, un emisor de eventos tal como un control Button no puede saber de antemano cuál método responderá a sus eventos. Por lo tanto, necesitamos un mecanismo para indicar cuál método será el receptor de eventos para un evento dado.

Delegados

Los manejadores de eventos se conectan a los eventos de un control a través de objetos especiales, conocidos como *delegados*. Un objeto delegado guarda una referencia a un método con una firma que se especifica mediante la declaración del tipo del delegado. Los controles de la GUI tienen delegados predefinidos que corresponden a cada uno de los eventos que pueden generar. Por ejemplo, el delegado para el evento Click de un control Button es del tipo EventHandler (espacio de nombres System). Si busca este tipo en la documentación de la ayuda en línea, verá que se declara de la siguiente manera:

```
public delegate void EventHandler( object sender, EventArgs e );
```

Aquí se utiliza la palabra clave **delegate** para declarar un tipo de delegado llamado EventHandler, el cual puede guardar referencias a los métodos que devuelven void y reciben dos parámetros: uno de tipo object (el emisor del evento) y uno de tipo EventArgs. Si compara la declaración del delegado con el encabezado de ClickButton_Click (figura 13.5, línea 16), podrá ver que este manejador de evento sin duda cumple con los requerimientos del delegado EventHandler. Observe que la anterior declaración crea toda una clase completa para usted. El compilador se encarga de los detalles relacionados con la declaración de esta clase especial.

Indicar el método que debe llamar un delegado

Un emisor de eventos llama a un objeto delegado de la misma forma que a un método. Como cada manejador de eventos se declara como un delegado, el emisor de eventos sólo necesita llamar al delegado apropiado cuando ocurre un evento; un control Button llama a su delegado EventHandler en respuesta a un clic. El trabajo del delegado es invocar al método apropiado. Para que se pueda hacer la llamada al método ClickButton_Click, Visual Studio asigna ClickButton_Click al delegado, como se muestra en la línea 39 de la figura 13.7. Visual Studio agrega este código cuando usted hace doble clic en el control Button, en el modo de **Diseño**. La expresión

```
new System.EventHandler(this.ClickButton_Click);
```

crea un objeto delegado EventHandler y lo inicializa con el método ClickButton_Click. La línea 39 utiliza el operador += para agregar el delegado al evento Click del objeto Button. Esto indica que ClickButton_Click responderá cuando un usuario haga clic en el botón. Observe que la clase del delegado sobrecarga el operador += cuando el compilador la crea.

Para especificar que se deben invocar varios métodos distintos en respuesta a un evento, podemos agregar otros delegados al evento Click del control Button con instrucciones similares a la línea 39 de la figura 13.7. Los delegados de eventos son *multidifusión*; representan un conjunto de objetos delegados, en donde todos tienen la misma firma. Los delegados multidifusión permiten que varios métodos se llamen en respuesta a un solo evento. Cuando ocurre un evento, el emisor de eventos llama a todos los métodos referenciados por el delegado multidifusión. A esto se le conoce como *multidifusión de eventos*. Los delegados de eventos se derivan de la clase **MulticastDelegate**, la cual se deriva de la clase **Delegate** (ambas del espacio de nombres System).

13.3.4 Otras formas de crear manejadores de eventos

En todas las aplicaciones con GUI que hemos creado hasta ahora, hacemos doble clic en un control en el formulario para crear un manejador de eventos para ese control. Esta técnica crea un manejador de eventos para el *evento predeterminado* de un control: el evento que se utiliza con más frecuencia con ese control. Por lo general, los controles pueden generar muchos tipos distintos de eventos, y cada tipo puede tener su propio manejador de eventos. Por ejemplo, ya ha creado manejadores de eventos Click para controles Button, haciendo doble clic en

un control `Button` en vista de diseño (`Click` es el evento predeterminado para un control `Button`). No obstante, su aplicación también puede proporcionar un manejador de eventos para un evento `MouseHover` del control `Button`, el cual ocurre cuando el puntero del ratón permanece posicionado sobre el control `Button`. Ahora veremos cómo crear un manejador de eventos para un evento que no es el evento predeterminado de un control.

Uso de la ventana Propiedades para crear manejadores de eventos

Puede crear manejadores de eventos adicionales, a través de la ventana **Propiedades**. Si selecciona un control en el formulario, haga clic en el icono **Eventos** (el icono del rayo en la figura 13.8) de la ventana **Propiedades** para que se listen todos los eventos para ese control. Puede hacer doble clic en el nombre de un evento para mostrar el manejador de eventos en el editor, si el manejador de eventos ya existe, o para crearlo. También puede seleccionar un evento y después usar la lista desplegable a su derecha para seleccionar un método existente que deba usarse como el manejador de eventos para ese evento. Los métodos que aparecen en esta lista desplegable son los métodos de la clase que tienen la firma apropiada para ser manejadores de eventos para el evento seleccionado. Para regresar a ver las propiedades de un control, seleccione el icono **Propiedades** (figura 13.8).

13.3.5 Localización de la información de los eventos

En la documentación de Visual Studio puede aprender acerca de los distintos eventos que produce un control. Para ello, seleccione **Ayuda > Índice**. En la ventana que aparezca, seleccione **.Net Framework** en la lista desplegable **Filtrado por** y escriba el nombre de la clase del control en la ventana **Índice**. Para asegurar que seleccione la clase apropiada, escribe el nombre completamente calificado de la clase, como se muestra en la figura 13.9 para la clase `System.Windows.Forms.Button`. Una vez que selecciona la clase de un control en la documentación, aparece una lista de todos los miembros de esa clase. Esta lista incluye los eventos que puede generar la clase. En la figura 13.9 nos desplazamos hasta los eventos de la clase `Button`. Haga clic en el nombre de un evento para ver su descripción y ejemplos de su uso (figura 13.10). Observe que el evento `Click` se lista como miembro de la clase `Control`, ya que el evento `Click` de la clase `Button` se hereda de la clase `Control`.

13.4 Propiedades y distribución de los controles

En esta sección veremos las generalidades sobre las propiedades comunes para muchos controles. Los controles se derivan de la clase **Control** (espacio de nombres `System.Windows.Forms`). La figura 13.11 lista algunas de las propiedades y métodos de la clase `Control`. Las propiedades que se muestran aquí pueden establecerse para muchos controles. Por ejemplo, la propiedad `Text` especifica el texto que aparece en un control. La posición de

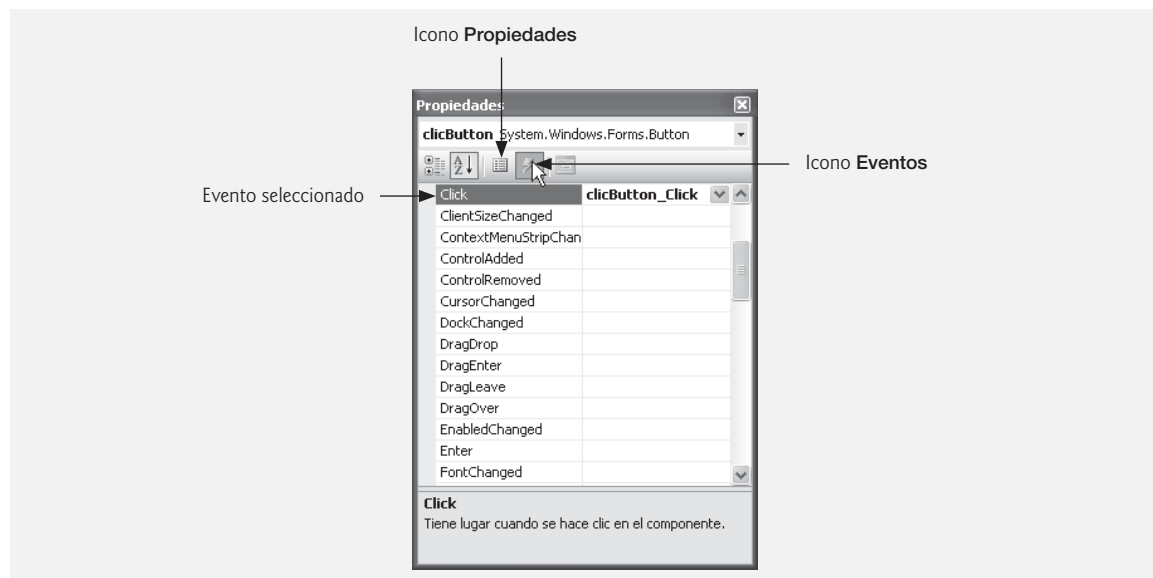


Figura 13.8 | En la ventana **Propiedades** se pueden ver los eventos para un control `Button`.

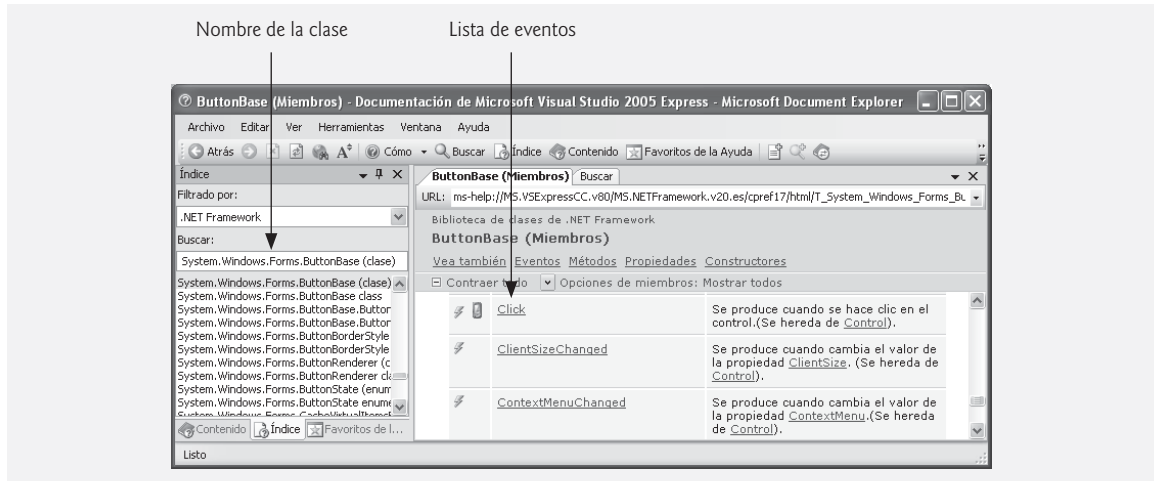


Figura 13.9 | Lista de eventos de Button.

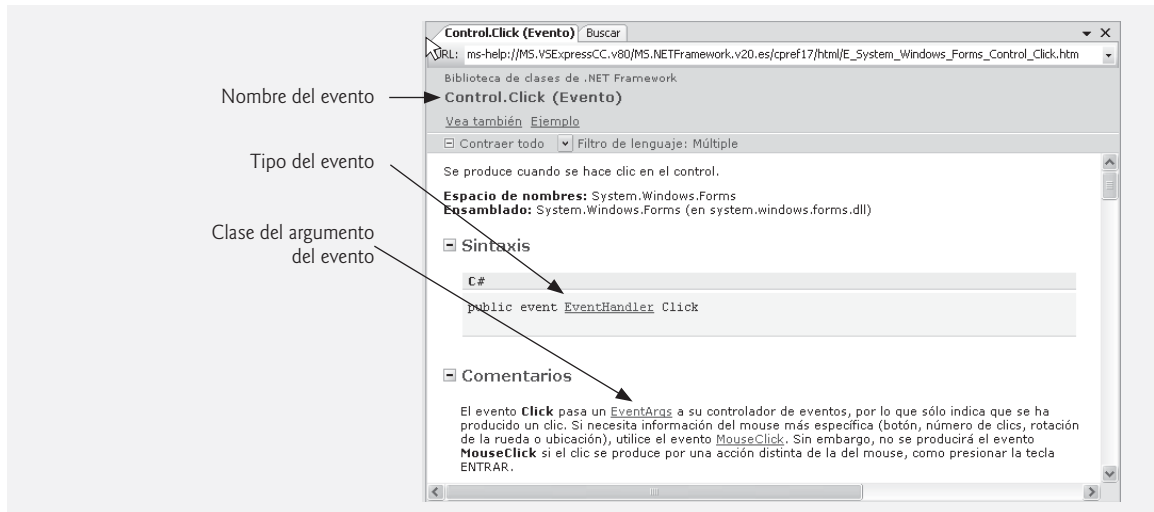


Figura 13.10 | Detalles del evento Click.

Propiedades y métodos de la clase Control	Descripción
<i>Propiedades comunes</i>	
BackColor	El color de fondo del control.
BackgroundImage	La imagen de fondo del control.
Enabled	Especifica si el control está habilitado (es decir, si el usuario puede interactuar con él). Por lo general, ciertas partes de un control deshabilitado aparecen en color gris como indicación visual para el usuario de que el control está deshabilitado.
Focused	Indica si el control tiene el foco.
Font	La fuente utilizada para mostrar el texto del control.

Figura 13.11 | Propiedades y métodos de la clase Control. (Parte 1 de 2).

Propiedades y métodos de la clase <code>Control</code>	Descripción
<code>ForeColor</code>	El color de primer plano del control. Por lo general, esta propiedad determina el color del texto en la propiedad <code>Text</code> .
<code>TabIndex</code>	El orden de tabulación del control. Cuando se oprime <i>Tab</i> , el foco se transfiere entre los controles, con base en el orden de tabulación. Usted puede establecer este orden.
<code>TabStop</code>	Si es <code>true</code> , entonces un usuario puede dar el foco a este control al oprimir <i>Tab</i> .
<code>Text</code>	El texto asociado con el control. La ubicación y apariencia del texto varían, dependiendo del tipo de control.
<code>Visible</code>	Indica si el control es visible.
<i>Métodos comunes</i>	
<code>Focus</code>	Adquiere el foco.
<code>Hide</code>	Oculta el control (establece la propiedad <code>Visible</code> a <code>false</code>).
<code>Show</code>	Muestra el control (establece la propiedad <code>Visible</code> a <code>true</code>).

Figura 13.11 | Propiedades y métodos de la clase `Control`. (Parte 2 de 2).

este texto varía, dependiendo del control. En un formulario de Windows, el texto aparece en la barra de título, pero el texto de un control `Button` aparece en su superficie.

El método **Focus** transfiere el foco a un control y lo convierte en el *control activo*. Cuando usted oprime la ficha *Tab* en una aplicación Windows en ejecución, los controles reciben el orden especificado mediante su propiedad **TabIndex**. Visual Studio establece esta propiedad con base en el orden en el que se agregan los controles a un formulario, pero usted puede cambiar el orden de tabulación. `TabIndex` es útil para los usuarios que introducen información en muchos controles, como en un conjunto de controles `TextBox` que representan el nombre, la dirección y el número telefónico de un usuario. Éste puede introducir información y seleccionar rápidamente el siguiente control, con sólo oprimir *Tab*.

La propiedad **Enabled** indica si el usuario puede interactuar con un control para generar un evento. A menudo, si se deshabilita un control, es debido a que no está disponible una opción para el usuario en ese momento. Por ejemplo, las aplicaciones de edición de texto deshabilitan con frecuencia el comando “pegar” hasta que el usuario copia algo de texto. En la mayoría de los casos, el texto de un control deshabilitado aparece en gris, en vez de negro). También puede ocultar un control al usuario sin deshabilitarlo, estableciendo la propiedad `Visible` a `false` o llamando al método `Hide`. En cada caso, el control aún existe pero no está visible en el formulario.

Puede usar el anclaje y el acoplamiento para especificar la distribución de los controles dentro de un contenedor (tal como `Form`). El **anclaje** hace que los controles permanezcan a una distancia fija desde los extremos del contenedor, incluso aunque éste cambie su tamaño. El anclaje mejora la experiencia del usuario. Por ejemplo, si el usuario espera que un control aparezca en cierta esquina de la aplicación, el anclaje asegura que el control siempre esté en esa esquina; aun si el usuario cambia el tamaño del formulario. El **acoplamiento** une un control a un contenedor, de tal forma que el control se estira a lo largo de todo un extremo del contenedor. Por ejemplo, un botón acoplado en la parte superior de un contenedor se estira a lo largo de toda la parte superior de ese contenedor, sin importar la anchura del contenedor.

Cuando se cambia el tamaño de los contenedores padre, los controles anclados se desplazan (y posiblemente cambian de tamaño) de tal forma que la distancia desde los lados hasta donde están anclados no varíe. De manera predeterminada, la mayoría de los controles se anclan a la esquina superior izquierda del formulario. Para ver los efectos de anclar un control, cree una aplicación Windows simple que contenga dos controles `Button`. Ancle un control a los extremos derecho e inferior, estableciendo la propiedad **Anchor** como se muestra en la figura 13.12. Deje el otro control sin anclar. Ejecute la aplicación y aumente el tamaño del formulario. Observe que el control `Button` anclado a la esquina superior derecha siempre se encuentra a la misma distancia desde la esquina inferior derecha del formulario (figura 13.13), pero el otro control permanece a su distancia original desde la esquina superior izquierda del formulario.

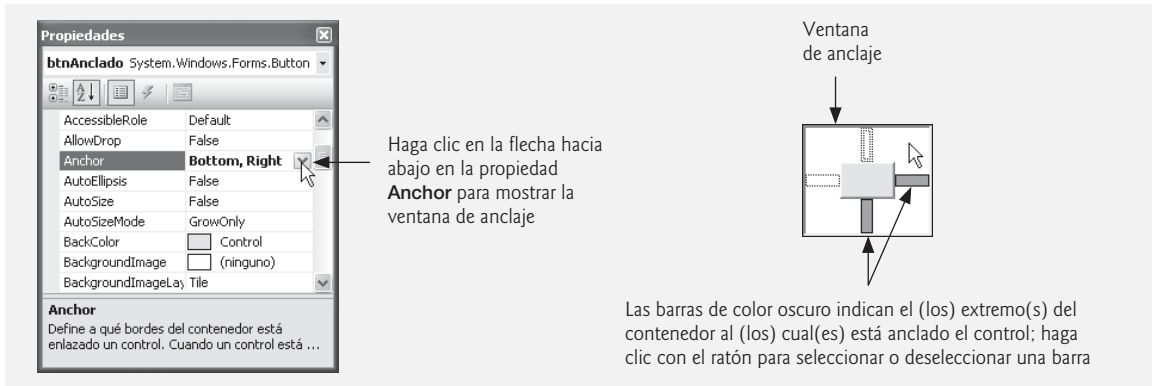


Figura 13.12 | Manipulación de la propiedad **Anchor** de un control.

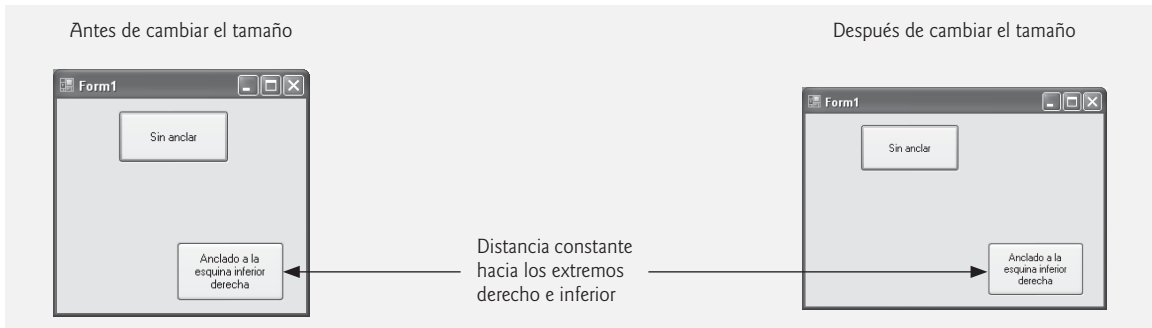


Figura 13.13 | Demostración del anclaje.

Algunas veces es conveniente que un control abarque todo un extremo completo del formulario, aun cuando éste cambie su tamaño. Por ejemplo, un control tal como una barra de estado debe permanecer comúnmente en la parte inferior del formulario. El acoplamiento permite que un control abarque todo un extremo completo (izquierda, derecha, superior o inferior) de su contenedor padre, o que llene todo el contenedor. Cuando se cambia el tamaño del control padre, el control acoplado también cambia su tamaño. En la figura 13.14, un control **Button** está acoplado a la parte superior del formulario (abarcando la parte superior). Cuando se cambia

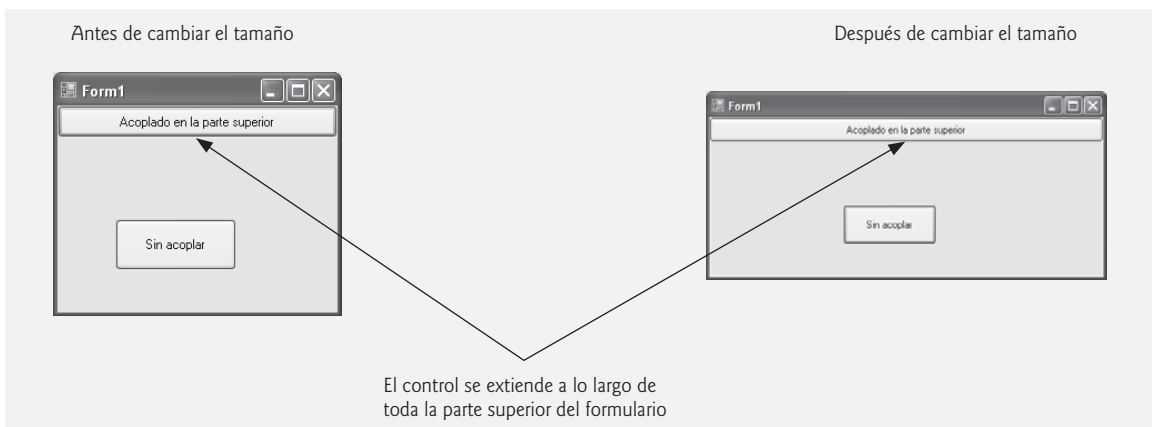


Figura 13.14 | Acoplar un control **Button** a la parte superior de un formulario.

el tamaño del formulario, el control `Button` cambia su tamaño para ajustarse a la nueva anchura del formulario. Los formularios tienen una propiedad **Padding**, la cual especifica la distancia entre los controles acoplados y los bordes del formulario. Esta propiedad especifica cuatro valores (una para cada extremo), y cada valor se establece a 0 de manera predeterminada. En la figura 13.15 se muestra un resumen de algunas propiedades comunes para distribuir controles.

Las propiedades `Anchor` y `Dock` de un objeto `Control` se establecen con respecto al contenedor padre del objeto `Control`, el cual podría ser un formulario o algún otro contenedor padre (como un `Panel`, que veremos en la sección 13.6). Los tamaños mínimo y máximo del formulario (o de otro `Control`) pueden establecerse mediante las propiedades **MinimumSize** y **MaximumSize**, respectivamente. Ambas son de tipo **Size**, que tiene las propiedades **Width** y **Height** para especificar el tamaño del formulario. Las propiedades `MinimumSize` y `MaximumSize` le permiten diseñar la distribución de la GUI para un rango de tamaño dado. El usuario no puede hacer un formulario más pequeño que el tamaño especificado por la propiedad `MinimumSize`, y no puede hacer un formulario más grande que el tamaño especificado por la propiedad `MaximumSize`. Para establecer un formulario a un tamaño fijo (en donde el usuario no pueda cambiar su tamaño), asigne el mismo valor a sus tamaños mínimo y máximo o establezca su propiedad `FormBorderStyle` a `FixedSingle`.



Observación de apariencia visual 13.2

Para los formularios que cambian de tamaño, asegúrese de que la distribución de la GUI aparezca consistente a lo largo de varios tamaños del formulario.

Uso de Visual Studio para editar la distribución de una GUI

Visual Studio cuenta con herramientas que nos ayudan con la distribución de la GUI. Tal vez usted se haya dado cuenta que, cuando arrastra un control sobre un formulario, aparecen líneas azules (conocidas como *líneas de ajuste*) para ayudarlo a posicionar el control con respecto a otros controles (figura 13.16) y los bordes del formulario. Esta nueva característica de Visual Studio 2005 hace que el control que usted arrastra “ajuste su posición” con respecto a los demás controles. Visual Studio también cuenta con el menú **Formato**, el cual contiene varias opciones para modificar la distribución de su GUI. El menú **Formato** no aparece en el IDE, a menos que usted seleccione un control (o conjunto de controles) en la vista de diseño. Cuando selecciona varios controles, puede usar el submenú **Alinear** del menú **Formato** para alinearlos. El menú **Formato** también le permite modificar la cantidad de espacio entre los controles, o centrar un control en el formulario.

Propiedades de la distribución de <code>Control</code>	Descripción
<code>Anchor</code>	Hace que un control permanezca a una distancia fija desde el (los) extremo(s) del contenedor, aun cuando se cambie su tamaño.
<code>Dock</code>	Permite que un control abarque un extremo de su contenedor, o que llene todo el contenedor.
<code>Padding</code>	Establece el espacio entre los bordes de un contenedor y los controles acoplados. El valor predeterminado es 0, lo cual hace que el control parezca estar empotrado a los extremos del contenedor.
<code>Location</code>	Especifica la ubicación (como un conjunto de coordenadas) de la esquina superior izquierda del control, en relación con su contenedor.
<code>Size</code>	Especifica el tamaño del control en píxeles como un objeto <code>Size</code> , que tiene las propiedades <code>Width</code> (anchura) y <code>Height</code> (altura).
<code>MinimumSize</code> , <code>MaximumSize</code>	Indica los tamaños mínimo y máximo de un objeto <code>Control</code> , respectivamente.

Figura 13.15 | Propiedades de distribución de `Control`.

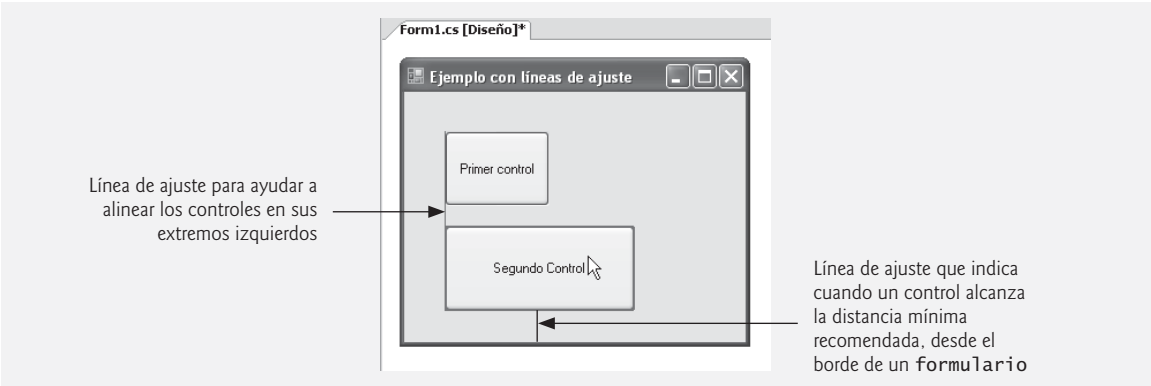


Figura 13.16 | Las líneas de ajuste en Visual Studio 2005.

13.5 Controles Label, TextBox y Button

Los controles `Label` proporcionan información de texto (y también imágenes opcionales) y se definen con la clase `Label` (una clase derivada de `Control`). Un control `Label` muestra texto que el usuario no puede modificar directamente. El texto de un control `Label` puede cambiarse mediante la programación, modificando la propiedad `Text`. La figura 13.17 lista las propiedades comunes de `Label`.

Un cuadro de texto (clase `TextBox`) es un área en la que un programa puede mostrar texto o el usuario puede escribir texto a través del teclado. Un **cuadro de texto de contraseña** es un control `TextBox` que oculta la información que introduce el usuario. A medida que el usuario escribe caracteres, el cuadro de texto de contraseña enmascara la entrada del usuario, mostrando un carácter que usted puede especificar. (Por lo general, *). Si establece la propiedad `PasswordChar`, el control `TextBox` se convierte en un cuadro de texto de contraseña. A menudo, los usuarios se encuentran con ambos tipos de controles `TextBox`, cuando inician sesión en una computadora o sitio Web; el control `TextBox` para el nombre de usuario permite a los usuarios introducir sus nombres de usuario; el control `TextBox` de contraseña permite a los usuarios introducir sus contraseñas. La figura 13.18 lista las propiedades comunes y un evento común de los controles `TextBox`.

Propiedades comunes de <code>Label</code>	Descripción
<code>Font</code>	La fuente del texto en el control <code>Label</code> .
<code>Text</code>	El texto en el control <code>Label</code> .
<code>TextAlign</code>	La alineación del texto en el control <code>Label</code> ; horizontal (izquierda, centro o derecha) y vertical (superior, media o inferior).

Figura 13.17 | Propiedades comunes de `Label`.

Propiedades y evento de <code>TextBox</code>	Descripción
<i>Propiedades comunes</i>	
<code>AcceptsReturn</code>	Si es <code>true</code> en un control <code>TextBox</code> con múltiples líneas, al oprimir <i>Intro</i> en el control <code>TextBox</code> se crea una nueva línea. Si es <code>false</code> , oprimir <i>Intro</i> es lo mismo que oprimir el control <code>Button</code> predeterminado en el formulario. El control <code>Button</code> predeterminado es el que se asigna a la propiedad <code>AcceptButton</code> de un formulario.

Figura 13.18 | Propiedades y evento de `TextBox`. (Parte 1 de 2).

Propiedades y evento de TextBox	Descripción
Multiline	Si es true, el control TextBox puede abarcar varias líneas. El valor predeterminado es false.
PasswordChar	Cuando se asigna un carácter a esta propiedad, el control TextBox se convierte en un cuadro de texto de contraseña, y el carácter especificado enmascara cada carácter que escribe el usuario. Si no se especifica un carácter, el control TextBox muestra el texto que se escribió.
ReadOnly	Si es true, el control TextBox tiene un fondo de color gris y no se puede editar su texto. El valor predeterminado es false.
ScrollBars	Para los cuadros de texto con múltiples líneas, esta propiedad indica cuáles barras de desplazamiento deben aparecer: ninguna (None), Horizontal, Vertical o ambas (Both).
Text	El contenido de texto del control TextBox.
<i>Evento común</i>	
TextChanged	Se genera cuando cambia el texto en un control TextBox (es decir, cuando el usuario agrega o elimina caracteres). Al hacer doble clic en el control TextBox en el modo de Diseño , se genera un manejador de eventos vacío para este evento.

Figura 13.18 | Propiedades y evento de TextBox. (Parte 2 de 2).

Un botón es un control que el usuario oprime para desencadenar una acción específica, o para seleccionar una opción en un programa. Como veremos más adelante, un programa puede usar varios tipos de botones, tales como *casillas de verificación* y *botones de opción*. Todas las clases de botones se derivan de la clase **ButtonBase** (espacio de nombres System.Windows.Forms), la cual define las características comunes de los botones. En esta sección veremos la clase Button, que por lo general permite a un usuario enviar un comando a una aplicación. La figura 13.19 lista las propiedades comunes y un evento común de la clase Button.



Observación de apariencia visual 13.3

Aunque los controles Label, TextBox y otros pueden responder a los clics del ratón, los controles Button son más naturales para este propósito.

Propiedades y evento de Button	Descripción
<i>Propiedades comunes</i>	
Text	Especifica el texto a mostrar en la superficie del control Button.
FlatStyle	Modifica la apariencia de un control Button: los atributos Flat. (Para que el botón se muestre sin una apariencia tridimensional), Popup. (Para que el botón aparezca plano hasta que el usuario mueva el puntero del ratón sobre el botón), Standard (tridimensional) y System, en donde el sistema operativo es el que controla la apariencia del botón. El valor predeterminado es Standard.
<i>Evento común</i>	
Click	Se genera cuando el usuario hace clic en el control Button. Al hacer doble clic en un control Button en vista de diseño, se crea un manejador de eventos vacío para este evento.

Figura 13.19 | Propiedades y evento de Button.

La figura 13.20 utiliza un control TextBox, un Button y un Label. El usuario introduce texto en un cuadro de contraseña y hace clic en el control Button, haciendo que se muestre la entrada de texto en el control Label. Por lo general no se muestra este texto; el propósito de los cuadros de texto de contraseña es ocultar el texto que introduce el usuario. Cuando éste hace clic en el botón **Muéstrame**, esta aplicación extrae el texto que escribió el usuario en el cuadro de texto de contraseña y lo muestra en otro control TextBox.

Primero, cree la GUI arrastrando los controles (un control TextBox, un Button y un Label) en el formulario. Una vez posicionados los controles, cambie sus nombres en la ventana **Propiedades**, de los valores predeterminados textBox1, button1 y label1, a unos más descriptivos: mostrarContraseniaLabel, mostrarContraseniaButton e introducirContraseniaTextBox. La propiedad (Name) en la ventana **Propiedades** nos permite modificar el nombre de la variable para un control. Visual Studio crea el código necesario y lo coloca en el método InitializeComponent de la clase parcial, en el archivo PruebaLabelTextBoxButtonForm.Designer.cs.

Después establecemos la propiedad Text de mostrarContraseniaButton a "Mostrame" y borramos las propiedades Text de mostrarContraseniaLabel e introducirContraseniaTextBox, de manera que estén en blanco cuando el programa empiece a ejecutarse. La propiedad BorderStyle de mostrarContraseniaLabel se estableció a fixed3D, para que el control Label tenga una apariencia tridimensional. La propiedad BorderStyle de todos los controles TextBox se establece a Fixed3D de manera predeterminada. El carácter de contraseña para introducirContraseniaTextBox se estableció mediante la asignación del carácter de asterisco (*) a la propiedad PasswordChar. Esta propiedad sólo acepta un carácter.

Para crear un manejador de eventos para mostrarContraseniaButton, hicimos doble clic en este control, en modo de **Diseño**. Agregamos la línea 22 al cuerpo del manejador de eventos. Cuando el usuario hace clic en el botón **Mostrame** de la aplicación en ejecución, la línea 22 obtiene el texto introducido por el usuario en introducirContraseniaTextBox y muestra el texto en mostrarContraseniaLabel.

```

1  // Fig. 13.20: PruebaLabelTextBoxButtonForm.cs
2  // Uso de los controles TextBox, Label y Button para mostrar
3  // el texto oculto en un cuadro de texto de contraseña.
4  using System;
5  using System.Windows.Forms;
6
7  // formulario que crea un cuadro de texto de contraseña y
8  // un control Label para mostrar el contenido del control TextBox
9  public partial class PruebaLabelTextBoxButtonForm : Form
10 {
11     // constructor predeterminado
12     public PruebaLabelTextBoxButtonForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     // muestra la entrada del usuario en el control Label
18     private void mostrarContraseniaButton_Click(
19         object sender, EventArgs e )
20     {
21         // muestra el texto que escribió el usuario
22         mostrarContraseniaLabel.Text = introducirContraseniaTextBox.Text;
23     } // fin del método mostrarContraseniaButton_Click
24 } // end class PruebaLabelTextBoxButtonForm

```

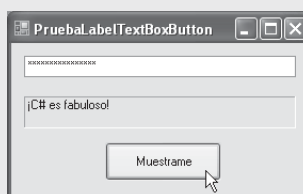


Figura 13.20 | Programa para mostrar el texto oculto en un cuadro de texto de contraseña.

13.6 Controles GroupBox y Panel

Los controles **GroupBox** y **Panel** ordenan los controles en una GUI. Por lo general, estos controles se utilizan para agrupar en una GUI los controles con una funcionalidad similar, o varios controles relacionados. Todos los controles dentro de un control GroupBox o Panel se mueven en conjunto, cuando se mueve el control GroupBox o Panel.

La principal diferencia entre estos dos controles es que el control GroupBox puede mostrar una leyenda (es decir, texto) y no incluye barras de desplazamiento, mientras que el control Panel puede incluir barras de desplazamiento pero no incluye una etiqueta. Los controles GroupBox tienen bordes delgados de manera predefinida; los controles Panel pueden establecerse de manera que también tengan bordes, con sólo cambiar su propiedad **BorderStyle**. Las figuras 13.21 y 13.22 listan las propiedades comunes de los controles GroupBox y Panel, respectivamente.



Observación de apariencia visual 13.4

Los controles Panel y GroupBox pueden contener otros controles Panel y GroupBox para lograr diseños más complejos.



Observación de apariencia visual 13.5

Puede organizar una GUI anclando y acoplando los controles dentro de un control GroupBox o Panel. Después, el control GroupBox o Panel puede anclarse o acoplarse dentro de un formulario. Esto divide los controles en “grupos” funcionales, que pueden ordenarse con facilidad.

Para crear un control GroupBox, arrastre su icono del **Cuadro de herramientas** hacia un formulario. Después, arrastre nuevos controles del **Cuadro de herramientas** hacia el control GroupBox. Estos controles se agregan a la propiedad **Controls** del control GroupBox y se convierten en parte del control GroupBox. La propiedad **Text** del control GroupBox especifica la leyenda.

Para crear un control Panel, arrastre su icono desde el **Cuadro de herramientas**, hasta el formulario. Después puede agregar los controles directamente al control Panel, arrastrándolos desde el **Cuadro de herramientas** hasta el control Panel. Para habilitar las barras de desplazamiento, establezca la propiedad **AutoScroll** del control Panel a **true**. Si el control Panel cambia su tamaño y no puede mostrar todos sus controles, aparecen barras de desplazamiento (figura 13.23). Las barras de desplazamiento pueden usarse para ver todos los controles dentro del control Panel; tanto en tiempo de diseño como en tiempo de ejecución. En la figura 13.23 establecemos la propiedad **BorderStyle** del control Panel a **FixedSingle**, para que pueda ver el control Panel en el formulario.

Propiedades de Groupbox Descripción	
Controls	El conjunto de controles que contiene el control GroupBox.
Text	Especifica el texto de la leyenda que se muestra en la parte superior del control GroupBox.

Figura 13.21 | Propiedades de GroupBox.

Propiedades de Panel Descripción	
AutoScroll	Indica si aparecen barras de desplazamiento cuando el control Panel es demasiado pequeño para mostrar todos sus controles. El valor predeterminado es false .
BorderStyle	Establece el borde del control Panel. El valor predeterminado es None ; las otras opciones son Fixed3D y FixedSingle .
Controls	El conjunto de controles que contiene el control Panel.

Figura 13.22 | Propiedades de Panel.

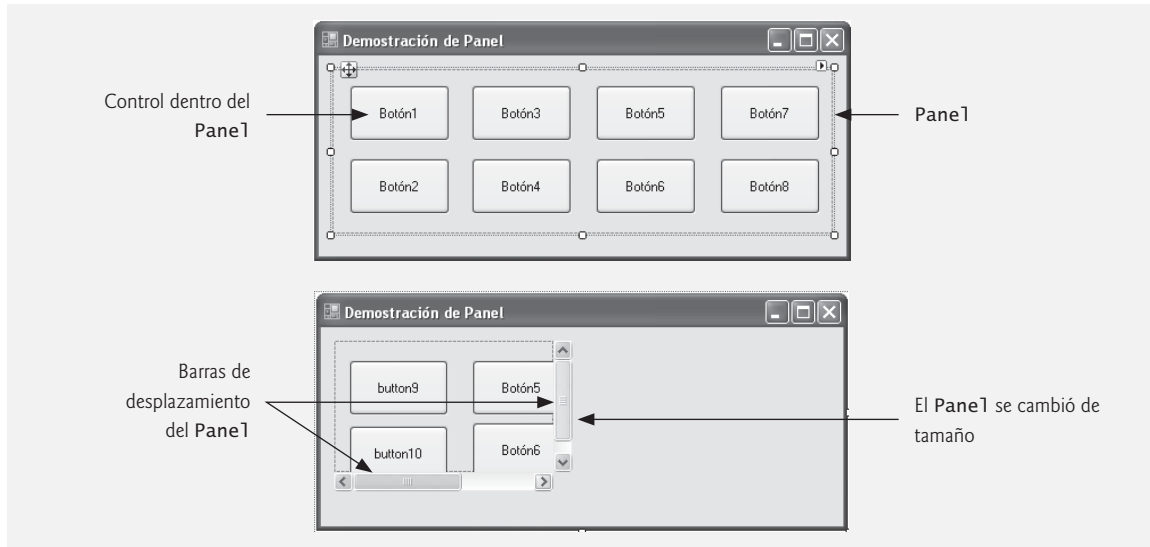


Figura 13.23 | Creación de un control Panel con barras de desplazamiento.



Observación de apariencia visual 13.6

Use controles Panel con barras de desplazamiento para evitar atestar una GUI, y para reducir su tamaño.

El programa de la figura 13.24 utiliza un control GroupBox y un control Panel para ordenar controles Button. Al hacer clic en estos controles Button, sus manejadores de eventos cambian el texto en un control Label.

```

1 // Fig. 13.24: EjemploGroupBoxPanelForm.cs
2 // Uso de los controles GroupBox y Panel para guardar controles Buttons.
3 using System;
4 using System.Windows.Forms;
5
6 // formulario que muestra un control GroupBox y un control Panel
7 public partial class EjemploGroupBoxPanelForm : Form
8 {
9     // constructor predeterminado
10    public EjemploGroupBoxPanelForm()
11    {
12        InitializeComponent();
13    } // fin del constructor
14
15    // manejador de eventos para el boton Hola
16    private void holaButton_Click( object sender, EventArgs e )
17    {
18        mensajeLabel.Text = "Se oprimió Hola"; // cambia el texto en el control Label
19    } // fin del método holaButton_Click
20
21    // manejador de eventos para el botón Adiós
22    private void adiosButton_Click( object sender, EventArgs e )
23    {
24        mensajeLabel.Text = "Se oprimió Adiós"; // cambia el texto en el control Label
25    } // fin del método adiosButton_Click

```

Figura 13.24 | Empleo del control GroupBox y el control Panel para ordenar los controles Button. (Parte 1 de 2).


```

26
27 // manejador de eventos para el botón Izquierda
28 private void izquierdaButton_Click( object sender, EventArgs e )
29 {
30     mensajeLabel.Text = "Se oprimió Izquierda"; // cambia el texto en el control Label
31 } // fin del método izquierdaButton_Click
32
33 // manejador de eventos para el botón Derecha
34 private void derechaButton_Click( object sender, EventArgs e )
35 {
36     mensajeLabel.Text = "Se oprimió Derecha"; // cambia el texto en el control Label
37 } // fin del método derechaButton_Click
38 } // fin de la clase EjemploGroupBoxPanelForm

```

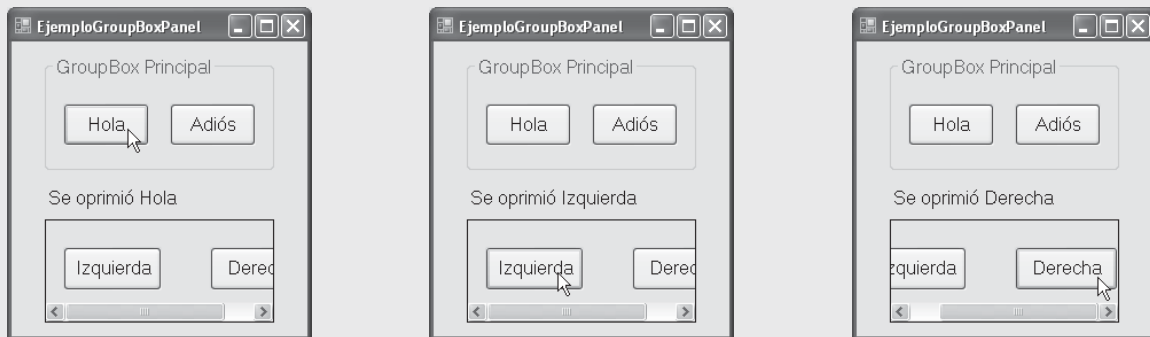


Figura 13.24 | Empleo del control GroupBox y el control Panel para ordenar los controles Button. (Parte 2 de 2).

El control GroupBox (llamado `principalGroupBox`) tiene dos controles Button: `holaButton` (que muestra el texto **Hola**) y `adiosButton` (que muestra el texto **Adiós**). El control Panel (llamado `principalPanel`) también tiene dos controles Button: `izquierdoButton` (que muestra el texto **Izquierda**) y `derechoButton` (que muestra el texto **Derecha**). El control `principalPanel` tiene su propiedad `AutoScroll` establecida en `true`, lo cual permite que aparezcan barras de desplazamiento cuando el contenido del control Panel requiere más espacio que su área visible. Al principio, el control Label (llamado `mensajeLabel`) está en blanco. Para agregar controles a `principalGroupBox` o `principalPanel`, Visual Studio llama al método `Add` de la propiedad `Controls` de cada contenedor. Este código se coloca en la clase parcial ubicada en el archivo `EjemploGroupBoxPanelForm.Designer.cs`.

Los manejadores de eventos para los cuatro controles Button se encuentran en las líneas 16-37. Agregamos una línea en cada manejador de eventos (líneas 18, 24, 30 y 36) para modificar el texto de `mensajeLabel`, de manera que indique cuál control Button oprimió el usuario.

13.7 Controles CheckBox y RadioButton

C# tiene dos tipos de *botones de estado*, que pueden estar en los estados encendido/apagado o verdadero/falso: **CheckBox** y **RadioButton**. Al igual que la clase Button, las clases CheckBox y RadioButton se derivan de la clase `ButtonBase`.

Controles CheckBox

Un control CheckBox es un pequeño cuadro, que está en blanco o contiene una marca de verificación. Cuando el usuario hace clic en un control CheckBox para seleccionarlo, aparece una marca de verificación en el cuadro. Si el usuario hace clic en el control CheckBox otra vez para deseccionarlo, se elimina la marca de verificación. Puede seleccionarse cualquier número de controles CheckBox en un momento dado. En la figura 13.25 aparece una lista de propiedades y eventos comunes de los controles CheckBox.

Propiedades y eventos de CheckBox	Descripción
<i>Propiedades comunes</i>	
Checked	Indica si el control CheckBox está seleccionado (contiene una marca de verificación) o deseleccionado (en blanco). Esta propiedad devuelve un valor Boolean.
CheckState	Indica si el control CheckBox está seleccionado o deseleccionado con un valor de la enumeración CheckState (Checked, Unchecked o Indeterminate). Indeterminate se utiliza cuando no está claro si el estado debe ser Checked (seleccionado) o Unchecked (deseleccionado). Por ejemplo, en Microsoft Word, cuando seleccionamos un párrafo que contiene varios formatos de caracteres y después seleccionamos Formato > Fuente , algunos de los controles CheckBox aparecen en el estado Indeterminate. Cuando CheckState se establece a Indeterminate, por lo general el control CheckBox se muestra en color gris.
Text	Especifica el texto que aparece a la derecha del control CheckBox.
<i>Eventos comunes</i>	
CheckedChanged	Se genera cuando cambia la propiedad Checked. Éste es un evento predeterminado del control CheckBox. Cuando un usuario hace doble clic en el control CheckBox en vista de diseño, se genera un manejador de eventos vacíos para este evento.
CheckStateChanged	Se genera cuando cambia la propiedad CheckState.

Figura 13.25 | Propiedades y eventos de CheckBox.

El programa en la figura 13.26 permite al usuario seleccionar controles CheckBox para modificar el estilo de fuente de un control Label. El manejador de eventos para un control CheckBox aplica negrita y el manejador de eventos para el otro control aplica cursiva. Si ambos controles CheckBox están seleccionados, el estilo de fuente se establece en negrita y cursiva. Al principio, ningún control CheckBox está seleccionado.

El control `negritaCheckBox` tiene en su propiedad `Text` el valor `Negrita`. El control `cursivaCheckBox` tiene en su propiedad `Text` el valor `Cursiva`. La propiedad `Text` de `salidaLabel` contiene el texto `Observe cómo cambia el estilo de la fuente`. Después de crear los controles, definimos sus manejadores de eventos. Al hacer doble clic en los controles CheckBox en tiempo de compilación, se crean los manejadores de eventos `CheckedChanged` vacíos.

Para modificar el estilo de fuente en un control `Label`, debe establecer su propiedad `Font` a un nuevo *objeto Font* (líneas 21-23 y 31-33). El constructor de `Font` que utilizamos aquí recibe el nombre, el tamaño y el estilo de la fuente como argumentos. Los primeros dos argumentos (`salidaLabel.Font.Name` y `salidaLabel.Font.Size`) usan el nombre y tamaño de fuente originales de `salidaLabel`. El estilo se especifica mediante un miembro de la *enumeración FontStyle*, que contiene `Regular`, `Bold`, `Italic`, `Strikeout` y `Underline`. (El estilo `Strikeout` muestra el texto con una línea a través de éste). La propiedad `Style` de un objeto `Font` es de sólo lectura, de manera que puede establecerse sólo a la hora de crear el objeto `Font`.

```
1 // Fig. 13.26: PruebaCheckBoxForm.cs
2 // Uso de controles CheckBox para activar los estilos cursiva y negrita.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6
7 // el formulario contiene controles CheckBox para permitir al usuario modificar
  el texto de ejemplo
8 public partial class PruebaCheckBoxForm : Form
```

Figura 13.26 | Uso de controles CheckBox para modificar los estilos de las fuentes. (Parte 1 de 2).

```

9  {
10 // constructor predeterminado
11 public PruebaCheckBoxForm()
12 {
13     InitializeComponent();
14 } // fin del constructor
15
16 // alterna el estilo de fuente entre negrita
17 // y normal, con base en la configuración actual
18 private void negritaCheckBox_CheckedChanged(
19     object sender, EventArgs e )
20 {
21     salidaLabel.Font =
22         new Font( salidaLabel.Font.Name, salidaLabel.Font.Size,
23             salidaLabel.Font.Style ^ FontStyle.Bold );
24 } // fin del método negritaCheckBox_CheckedChanged
25
26 // alterna el estilo de fuente entre cursiva y
27 // normal, con base en la configuración actual
28 private void cursivaCheckBox_CheckedChanged(
29     object sender, EventArgs e )
30 {
31     salidaLabel.Font =
32         new Font( salidaLabel.Font.Name, salidaLabel.Font.Size,
33             salidaLabel.Font.Style ^ FontStyle.Italic );
34 } // fin del método cursivaCheckBox_CheckedChanged
35 } // fin de la clase PruebaCheckBoxForm

```

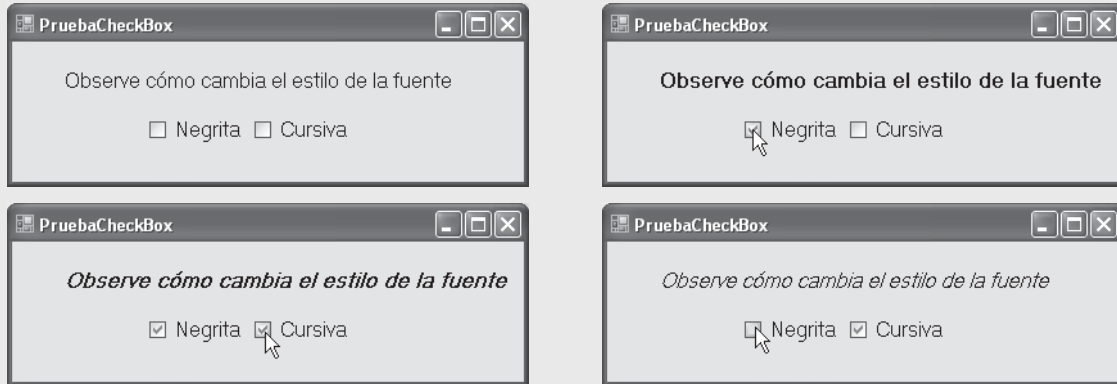


Figura 13.26 | Uso de controles CheckBox para modificar los estilos de las fuentes. (Parte 2 de 2).

Los estilos pueden combinarse mediante *operadores a nivel de bit*: operadores que realizan manipulaciones con bits de información. En el capítulo 1 vimos que todos los datos se representan en la computadora como combinaciones de 0s y 1s. Cada 0 o 1 representa a un bit. `FontStyle` tiene un atributo llamado `System.FlagsAttribute`, lo que significa que los valores de bit de `FontStyle` se seleccionan de una manera que nos permita combinar distintos elementos `FontStyle` para crear estilos compuestos, mediante el uso de operadores a nivel de bit. Estos estilos no son mutuamente exclusivos, por lo que podemos combinar distintos estilos y eliminarlos sin afectar la combinación de los elementos `FontStyle` anteriores. Podemos combinar estos diversos estilos de fuente, usando ya sea el operador OR lógico (`|`) o el operador OR exclusivo (`^`). Cuando se aplica el operador OR lógico a dos bits, si por lo menos uno de los dos bits tiene el valor 1, entonces el resultado es 1. La combinación de estilos mediante el uso del operador OR condicional funciona de la siguiente forma. Suponga que el estilo `FontStyle.Bold` se representa mediante los bits 01 y que `FontStyle.Italic` se representa mediante los bits 10. Si utilizamos el OR condicional (`|`) para combinar los estilos, obtenemos los bits 11.

```

01 = Bold
10 = Italic
--
11 = Bold e Italic

```

El operador OR condicional nos ayuda a crear combinaciones de estilos. No obstante, ¿qué ocurre si queremos deshacer una combinación de estilos, como en la figura 13.26?

El operador lógico OR exclusivo nos permite combinar estilos y deshacer las configuraciones de estilo existentes. Cuando se aplica el OR lógico exclusivo a dos bits, si ambos bits tienen el mismo valor, entonces el resultado es 0. Si ambos bits son distintos, entonces el resultado es 1.

La combinación de estilos mediante el uso de OR lógico exclusivo funciona de la siguiente manera. Suponga de nuevo que `FontStyle.Bold` se representa mediante los bits 01 y que `FontStyle.Italic` se representa mediante los bits 10. Al utilizar el OR exclusivo lógico (^) en ambos estilos, obtenemos los bits 11.

```

01 = Bold
10 = Italic
--
11 = Bold e Italic

```

Ahora suponga que queremos eliminar el estilo `FontStyle.Bold` de la combinación anterior entre `FontStyle.Bold` y `FontStyle.Italic`. La manera más sencilla de hacerlo es volver a aplicar el operador OR exclusivo lógico (^) al estilo compuesto y a `FontStyle.Bold`.

```

11 = Bold e Italic
01 = Bold
--
10 = Italic

```

Éste es un ejemplo simple. Las ventajas de utilizar operadores a nivel de bit para combinar valores `FontStyle` se hacen más evidentes si consideramos que hay cinco valores distintos de `FontStyle` (`Bold`, `Italic`, `Regular`, `Strikeout` y `Underline`), lo que produce 16 combinaciones distintas. Al utilizar los operadores a nivel de bit para combinar los estilos de fuente, se reduce en forma considerable la cantidad de código requerido para comprobar todas las combinaciones de fuente posibles.

En la figura 13.26 necesitamos establecer el estilo de fuente, de manera que el texto aparezca en negrita si no estaba así originalmente, y viceversa. Observe que la línea 23 utiliza el operador OR exclusivo lógico a nivel de bits para hacer esto. Si `salidaLabel.Font.Style` está en negrita, entonces el estilo resultante debe ser sin negrita. Si el texto está originalmente en cursiva, el estilo resultante debe ser negrita y cursiva, en vez de sólo negrita. Lo mismo se aplica para `FontStyle.Italic` en la línea 33.

Si no utilizáramos los operadores a nivel de bits para componer los elementos `FontStyle`, tendríamos que evaluar el estilo actual y cambiarlo de manera acorde. Por ejemplo, en el manejador de eventos `negritaCheckBox_CheckedChanged` podríamos evaluar si tiene el estilo regular y convertirlo en negrita y cursiva; evaluar si tiene el estilo negrita y hacerlo regular; evaluar si tiene el estilo cursiva y hacerlo negrita y cursiva; y evaluar si tiene el estilo negrita y cursiva y hacerlo cursiva. Esto es incómodo, ya que para cada nuevo estilo que agregamos, duplicamos el número de combinaciones. Si agregáramos un control `CheckBox` para el estilo subrayado se requeriría evaluar ocho estilos adicionales. Si agregáramos un control `CheckBox` para el estilo tachado se requeriría evaluar 16 estilos adicionales.

Controles RadioButton

Los botones de opción (que se definen con la clase `RadioButton`) son similares a los controles `CheckBox` en cuanto a que también tienen dos estados: *seleccionado* y *no seleccionado* (también conocido como *deseleccionado*). Sin embargo, por lo general los controles `RadioButton` aparecen como un *grupo*, en el cual sólo puede seleccionarse un control `RadioButton` a la vez. Al seleccionar un control `RadioButton` en el grupo, se obliga a que todos los demás queden *deseleccionados*. Por lo tanto, los controles `RadioButton` se utilizan para representar un conjunto de opciones *mutuamente exclusivas* (es decir, un conjunto en el cual no pueden seleccionarse varias opciones al mismo tiempo).



Observación de apariencia visual 13.7

Use controles `RadioButton` cuando el usuario sólo debe seleccionar una opción en un grupo.



Observación de apariencia visual 13.8

Use controles `CheckBox` cuando el usuario deba poder seleccionar varias opciones en un grupo.

Todos los controles `RadioButton` que se agregan a un contenedor forman parte del mismo grupo. Para separar controles `RadioButton` en varios grupos, los controles deben agregarse a controles `GroupBox` o `Panel`. En la figura 13.27 se listan las propiedades comunes y un evento común de la clase `RadioButton`.



Observación de ingeniería de software 13.2

Los controles `Form`, `GroupBox` y `Panel` pueden actuar como grupos lógicos para los controles `RadioButton`. Los controles `RadioButton` dentro de cada grupo son mutuamente exclusivos entre sí, pero no con los controles `RadioButton` en distintos grupos lógicos.

El programa en la figura 13.28 utiliza controles `RadioButton` para permitir a los usuarios seleccionar opciones para un cuadro de diálogo `MessageBox`. Después de seleccionar los atributos deseados, el usuario oprime el botón **Mostrar** para mostrar el cuadro de diálogo `MessageBox`. Un control `Label` en la esquina inferior izquierda muestra el resultado del cuadro de diálogo `MessageBox` (es decir, cuál control `Button` oprimió el usuario: **Sí**, **No**, **Cancelar**, etcétera).

Para almacenar las opciones del usuario, creamos e inicializamos los objetos `tipoIcono` y `tipoBoton` (líneas 11-12). El objeto `tipoIcono` es de tipo `MessageBoxIcon`, y puede tener los valores `Asterisk` (asterisco), `Error`, `Exclamation` (exclamación), `Hand` (mano), `Information` (información), `None` (ninguno), `Question` (pregunta), `Stop` (alto) y `Warning` (advertencia). Los resultados de ejemplo sólo muestran los iconos `Error`, `Exclamation`, `Information` y `Question`.

El objeto `tipoBoton` es de tipo `MessageBoxButtons` y puede tener los valores `AbortRetryIgnore` (abortar/reintentar/ignorar), `OK` (aceptar), `OKCancel` (aceptar/cancelar), `RetryCancel` (reintentar/cancelar), `YesNo` (sí/no) y `YesNoCancel` (sí/no/cancelar). El nombre indica las opciones presentes para el usuario en el cuadro de diálogo `MessageBox`. Las ventanas de resultados de ejemplo muestran cuadros de diálogo `MessageBox` para todos los valores de la enumeración `MessageBoxButtons`.

Creamos dos controles `GroupBox`, uno para cada conjunto de valores de enumeración. Las leyendas de los controles `GroupBox` son **Tipo de botón** e **Icono**. Los controles `GroupBox` contienen controles `RadioButton` para las opciones de enumeración correspondientes, y las propiedades `Text` de estos controles se establecen en forma apropiada. Como los controles `RadioButton` están agrupados, sólo puede seleccionarse un control `RadioButton` de cada control `GroupBox`. También hay un control `Button` (`mostrarButton`) con la etiqueta **Mostrar**. Cuando

Propiedades y evento de <code>RadioButton</code>	Descripción
<i>Propiedades comunes</i>	
<code>Checked</code>	Indica si el control <code>RadioButton</code> está seleccionado.
<code>Text</code>	Especifica el texto del control <code>RadioButton</code> .
<i>Evento común</i>	
<code>CheckedChanged</code>	Se genera cada vez que se selecciona o deselecciona el control <code>RadioButton</code> . Al hacer doble clic en un control <code>RadioButton</code> en vista de diseño, se genera un manejador de eventos vacío para este evento.

Figura 13.27 | Propiedades y evento de `RadioButton`.

un usuario hace clic en este control `Button`, se muestra un cuadro de diálogo `MessageBox` personalizado. Un control `Label` (`mostrarLabel`) muestra cuál botón oprimió el usuario dentro del cuadro de diálogo `MessageBox`.

El manejador de eventos para los controles `RadioButton` maneja el evento `CheckedChanged` de cada control `RadioButton`. Cuando se selecciona un control `RadioButton` contenido dentro del control `GroupBox` llamado **Tipo de botón**, el manejador de eventos correspondiente al control `RadioButton` seleccionado establece `tipoBoton` al valor apropiado. Las líneas 21-45 contienen el manejo de eventos para estos controles `RadioButton`. De manera similar, cuando el usuario selecciona los controles `RadioButton` que pertenecen al control `GroupBox` llamado **Icono**, los manejadores de eventos asociados con estos eventos (líneas 48-80) establecen `tipoIcono` a su valor correspondiente.

```

1  // Fig. 13.28: PruebaRadioButtonsForm.cs
2  // Uso de controles RadioButton para establecer las opciones de una ventana de mensajes.
3  using System;
4  using System.Windows.Forms;
5
6  // el formulario contiene varios controles RadioButton--el usuario selecciona
7  // uno de cada grupo para crear un cuadro de diálogo MessageBox personalizado
8  public partial class PruebaRadioButtonsForm : Form
9  {
10     // crea variables que almacenan la selección de opciones del usuario
11     private MessageBoxIcon tipoIcono;
12     private MessageBoxButtons tipoBoton;
13
14     // constructor predeterminado
15     public PruebaRadioButtonsForm()
16     {
17         InitializeComponent();
18     } // fin del constructor
19
20     // cambia los controles Button con base en la opción elegida por el emisor
21     private void tipoBoton_CheckedChanged( object sender, EventArgs e )
22     {
23         if ( sender == aceptarButton ) // muestra botón Aceptar
24             tipoBoton = MessageBoxButtons.OK;
25
26         // muestra botones Aceptar y Cancelar
27         else if ( sender == aceptarCancelarButton )
28             tipoBoton = MessageBoxButtons.OKCancel;
29
30         // muestra botones Abortar, Reintentar e Ignorar
31         else if ( sender == abortarReintentarIgnorarButton )
32             tipoBoton = MessageBoxButtons.AbortRetryIgnore;
33
34         // muestra botones Sí, No y Cancelar
35         else if ( sender == siNoCancelarButton )
36             tipoBoton = MessageBoxButtons.YesNoCancel;
37
38         // muestra botones Sí y No
39         else if ( sender == siNoButton )
40             tipoBoton = MessageBoxButtons.YesNo;
41
42         // sólo queda una opción: mostrar botones Reintentar y Cancelar
43         else
44             tipoBoton = MessageBoxButtons.RetryCancel;
45     } // fin del método tipoBoton_Changed
46

```

Figura 13.28 | Uso de controles `RadioButton` para establecer las opciones de la ventana de mensajes. (Parte 1 de 3).

```

47 // cambia el icono con base en la opción elegida por el emisor
48 private void tipoIcono_CheckedChanged( object sender, EventArgs e )
49 {
50     if ( sender == asteriscoButton ) // muestra icono de asterisco
51         tipoIcono = MessageBoxIcon.Asterisk;
52
53     // muestra icono de error
54     else if ( sender == errorButton )
55         tipoIcono = MessageBoxIcon.Error;
56
57     // muestra icono de signo de exclamación
58     else if ( sender == exclamacionButton )
59         tipoIcono = MessageBoxIcon.Exclamation;
60
61     // muestra icono de mano
62     else if ( sender == manoButton )
63         tipoIcono = MessageBoxIcon.Hand;
64
65     // muestra icono de información
66     else if ( sender == informacionButton )
67         tipoIcono = MessageBoxIcon.Information;
68
69     // muestra icono de signo de interrogación
70     else if ( sender == preguntaButton )
71         tipoIcono = MessageBoxIcon.Question;
72
73     // muestra icono de alto
74     else if ( sender == altoButton )
75         tipoIcono = MessageBoxIcon.Stop;
76
77     // sólo queda una opción: muestra icono de advertencia
78     else
79         tipoIcono = MessageBoxIcon.Warning;
80 } // fin del método tipoIcono_CheckChanged
81
82 // muestra MessageBox y Button que oprimió el usuario
83 private void mostrarButton_Click( object sender, EventArgs e )
84 {
85     // muestra MessageBox y almacena el
86     // valor del control Button que se oprimió
87     DialogResult resultado = MessageBox.Show(
88         "Éste es su MessageBox personalizado.", "MessageBox personalizado",
89         tipoBoton, tipoIcono, 0, 0 );
90
91     // comprueba qué control Button se oprimió en el MessageBox
92     // cambia el texto mostrado de manera acorde
93     switch (resultado)
94     {
95         case DialogResult.OK:
96             mostrarLabel.Text = "Se oprimió Aceptar.";
97             break;
98         case DialogResult.Cancel:
99             mostrarLabel.Text = "Se oprimió Cancelar.";
100            break;
101         case DialogResult.Abort:
102             mostrarLabel.Text = "Se oprimió Abortar.";
103             break;
104         case DialogResult.Retry:
105             mostrarLabel.Text = "Se oprimió Reintentar.";

```

Figura 13.28 | Uso de controles RadioButton para establecer las opciones de la ventana de mensajes. (Parte 2 de 3).


```

106         break;
107     case DialogResult.Ignore:
108         mostrarLabel.Text = "Se oprimió Ignorar.";
109         break;
110     case DialogResult.Yes:
111         mostrarLabel.Text = "Se oprimió Sí.";
112         break;
113     case DialogResult.No:
114         mostrarLabel.Text = "Se oprimió No.";
115         break;
116     } // fin de switch
117 } // fin del método mostrarButton_Click
118 } // fin de la clase PruebaRadioButtonsForm

```

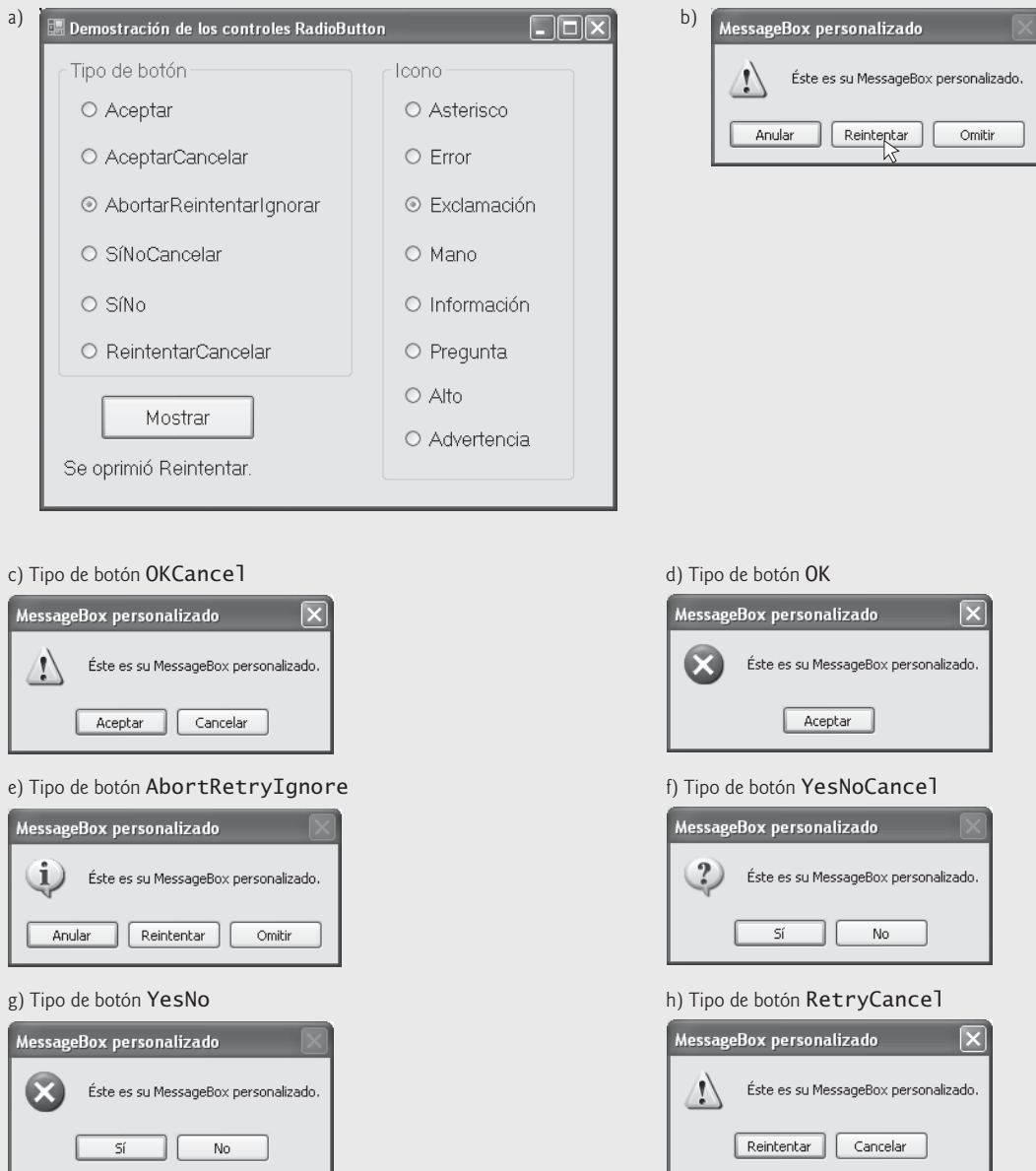


Figura 13.28 | Uso de controles RadioButton para establecer las opciones de la ventana de mensajes. (Parte 3 de 3).

El manejador de eventos `Click` para `mostrarButton` (líneas 83-117) crea un cuadro de diálogo `MessageBox` (líneas 87-89). Las opciones del `MessageBox` se especifican con los valores almacenados en `tipoIcono` y `tipoBoton`. Cuando el usuario hace clic en uno de los botones del cuadro de diálogo `MessageBox`, el resultado del cuadro de mensajes se devuelve a la aplicación. Este resultado es un valor de la *enumeración* `DialogResult` que contiene `Abort` (abortar), `Cancel` (cancelar), `Ignore` (ignorar), `No`, `None` (ninguno), `OK` (aceptar), `Retry` (reintentar) o `Yes` (sí). La instrucción `switch` en las líneas 93-116 evalúa el resultado y establece el valor de `mostrarLabel.Text` de manera apropiada.

13.8 Controles PictureBox

Un control `PictureBox` muestra una imagen. Esta imagen puede tener uno de varios formatos, como mapa de bits, GIF (Formato de intercambio de gráficos) y JPEG. (En el capítulo 17, Gráficos y multimedia, hablaremos sobre las imágenes). La propiedad `Image` de un control `PictureBox` especifica la imagen que se mostrará, y la propiedad `SizeMode` indica cómo se mostrará: `Normal`, `StretchImage` (estirar imagen), `AutoSize` (tamaño automático) o `CenterImage` (centrar imagen). La figura 13.29 describe las propiedades comunes y un evento común del control `PictureBox`.

La figura 13.30 utiliza un control `PictureBox` llamado `imagenPictureBox` para mostrar una de tres imágenes de mapa de bits: `imagen0`, `imagen1` o `imagen2`. Estas imágenes están ubicadas en el directorio `imagenes`, dentro de los directorios `bin/Debug` y `bin/Release` del proyecto. Cada vez que el usuario hace clic en el botón

Propiedades y evento de PictureBox	Descripción
<i>Propiedades comunes</i>	
Image	Establece la imagen a mostrar en el control PictureBox.
SizeMode	Enumeración que controla el tamaño y el posicionamiento de la imagen. Los valores son <code>Normal</code> (predeterminado), <code>StretchImage</code> , <code>AutoSize</code> y <code>CenterImage</code> . <code>Normal</code> coloca la imagen en la esquina superior izquierda del control PictureBox, y <code>CenterImage</code> coloca la imagen en el centro. Estas dos opciones truncan la imagen si es demasiado grande. <code>StretchImage</code> cambia el tamaño de la imagen para que se ajuste al control PictureBox. <code>AutoSize</code> cambia el tamaño del control PictureBox para que muestre toda la imagen.
<i>Evento común</i>	
Click	Ocurre cuando el usuario hace clic en el control. Al hacer doble clic en este control en la vista de diseño, se crea un manejador de eventos para este evento.

Figura 13.29 | Propiedades y evento de PictureBox.

```
1 // Fig. 13.30: PruebaPictureBoxForm.cs
2 // Uso de un control PictureBox para mostrar imágenes.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6 using System.IO;
7
8 // formulario para mostrar distintas imágenes en PictureBox al hacer clic en el botón
9 public partial class PruebaPictureBoxForm : Form
10 {
11     private int numImagen = -1; // determina cuál imagen se va a mostrar
12
13     // constructor predeterminado
14     public PruebaPictureBoxForm()
```

Figura 13.30 | Uso de un control PictureBox para mostrar imágenes. (Parte I de 2).

```

15     {
16         InitializeComponent();
17     } // fin del constructor
18
19     // cambia la imagen cada vez que se hace clic en el botón Siguiente imagen
20     private void siguienteButton_Click( object sender, EventArgs e )
21     {
22         numImagen = ( numImagen + 1 ) % 3; // numImagen itera de 0 a 2
23
24         // crea objeto Image del archivo, muestra en control PicutreBox
25         imagenPictureBox.Image = Image.FromFile(
26             Directory.GetCurrentDirectory() + @"\" + "imagenes\" + "imagen" +
27             numImagen + ".bmp" );
28     } // fin del método siguienteButton_Click
29 } // fin de la clase PruebaPictureBoxForm

```

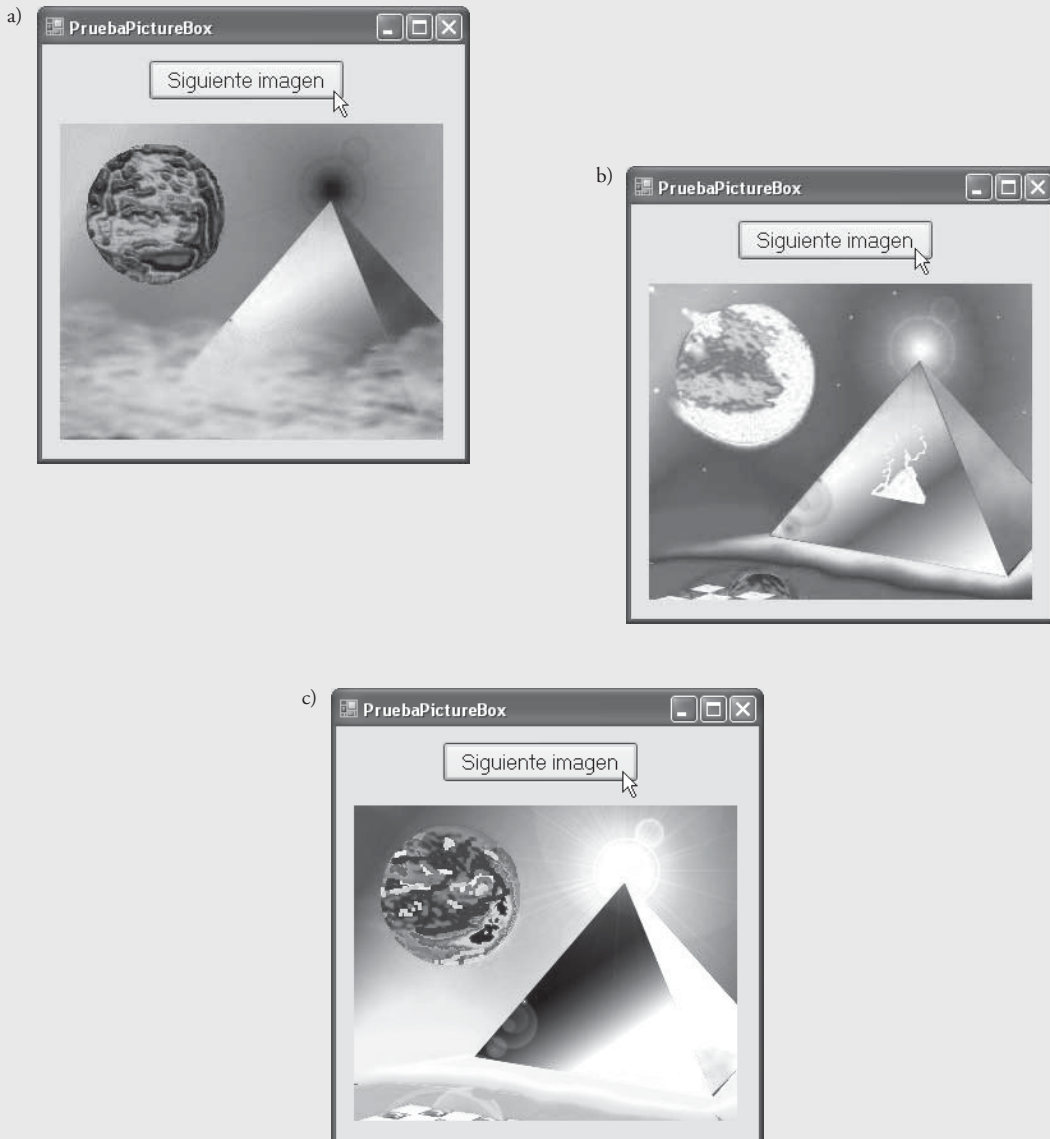


Figura 13.30 | Uso de un control PictureBox para mostrar imágenes. (Parte 2 de 2).

Siguiente imagen, ésta cambia a la siguiente imagen en la secuencia. Cuando se muestra la última imagen y el usuario hace clic en el botón **Siguiente imagen**, se muestra de nuevo la primera imagen. Dentro del manejador de eventos `siguienteButton_Click` (líneas 20-28) utilizamos un `int` (`numImagen`) para almacenar el número de la imagen que deseamos mostrar. Después establecemos la propiedad `Image` de `imagenPictureBox` para asignarle un objeto `Image` (líneas 25-27).

13.9 Controles ToolTip

En el capítulo 2 demostramos el uso de los cuadros de información sobre herramientas (tool tips): el útil texto que aparece cuando colocamos el puntero del ratón sobre un elemento en una GUI. Recuerde que los cuadros de información sobre herramientas que se muestran en Visual Studio le ayudan a familiarizarse con las características del IDE, y le sirven como recordatorios útiles para la funcionalidad de los iconos de cada una de las barras de herramientas. Muchos programas utilizan los cuadros de información sobre herramientas para recordar a los usuarios el propósito de cada control. Por ejemplo, Microsoft Word cuenta con cuadros de información sobre herramientas que ayudan a los usuarios a determinar el propósito de cada uno de los iconos de la aplicación. En esta sección le demostraremos cómo usar el **componente ToolTip** para agregar, a sus aplicaciones, cuadros de información sobre herramientas. La figura 13.31 describe las propiedades comunes y un evento común de la clase `ToolTip`.

Cuando agregamos un componente `ToolTip` desde el **Cuadro de herramientas**, aparece en la *bandeja de componentes*: la región gris debajo del formulario en el modo de **Diseño**. Una vez que se agrega un componente `ToolTip` a un formulario, aparece una nueva propiedad en la ventana **Propiedades** para los demás controles del formulario. Esta propiedad aparece en la ventana **Propiedades** como **ToolTip en**, seguida del nombre del componente `ToolTip`. Por ejemplo, si el componente `ToolTip` de nuestro formulario se llamara `ayudaToolTip`, estableceríamos el valor de la propiedad **ToolTip en ayudaToolTip** de un control para especificar el texto del cuadro de información sobre herramientas de ese control. La figura 13.32 demuestra el uso del componente `ToolTip`.

Propiedades y evento de ToolTip	Descripción
<i>Propiedades comunes</i>	
AutoPopDelay	La cantidad de tiempo (en milisegundos) que aparece el cuadro de información sobre herramientas, mientras el ratón se coloca sobre un control.
InitialDelay	La cantidad de tiempo (en milisegundos) que debe colocarse un ratón sobre un control para que aparezca un cuadro de información sobre herramientas.
ReshowDelay	La cantidad de tiempo (en milisegundos) entre la cual aparecen dos cuadros de información sobre herramientas distintos (cuando el ratón se desplaza de un control a otro).
<i>Evento común</i>	
Draw	Se genera cuando se muestra el cuadro de información sobre herramientas. Este evento permite a los programadores modificar la apariencia del cuadro de información sobre herramientas.

Figura 13.31 | Propiedades y evento de `ToolTip`.

```
1 // Fig. 13.32: EjemploToolTipForm.cs
2 // Demostración del componente ToolTip.
3 using System;
4 using System.Windows.Forms;
5
6 public partial class EjemploToolTipForm : Form
7 {
```

Figura 13.32 | Demostración del componente `ToolTip`. (Parte I de 2).

```

8      // constructor predeterminado
9      public EjemploToolTipForm()
10     {
11         InitializeComponent();
12     } // fin del constructor
13
14     // no se necesitan manejadores de eventos para este ejemplo
15
16 } // fin de la clase ToolTipExampleForm

```



Figura 13.32 | Demostración del componente ToolTip. (Parte 2 de 2).

Para este ejemplo, crearemos una GUI que contenga dos controles `Label`, para poder demostrar el uso de distinta información de texto en el cuadro de información sobre herramientas para cada control `Label`. Para mejorar la legibilidad de los resultados de ejemplo, establecimos la propiedad `BorderStyle` de cada control `Label` a `FixedSingle`, la cual muestra un borde sólido. Como no hay código para manejar eventos en este ejemplo, la clase de la figura 13.32 sólo contiene un constructor.

En este ejemplo, al componente `ToolTip` le llamamos `etiquetasToolTip`. La figura 13.33 muestra el componente `ToolTip` en la bandeja de componentes. Establecimos el texto del cuadro de información sobre herramientas para el primer control `Label` a "Primera etiqueta" y el cuadro de información sobre herramientas para el segundo control `Label` a "Segunda etiqueta". La figura 13.34 demuestra cómo establecer el texto del cuadro de información sobre herramientas para el primer control `Label`.

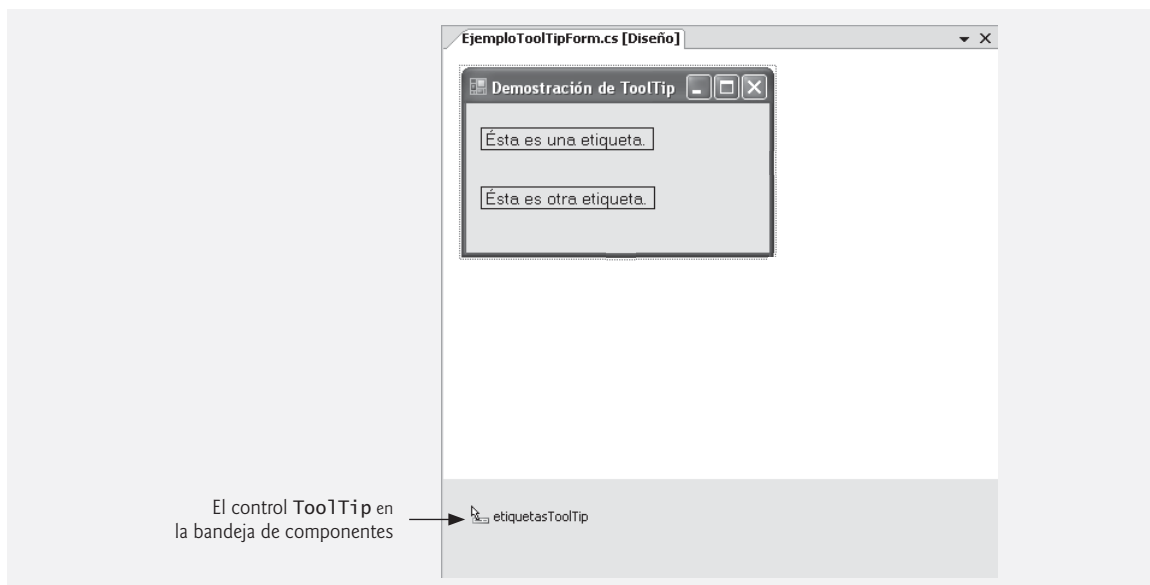


Figura 13.33 | Demostración de la bandeja de componentes.

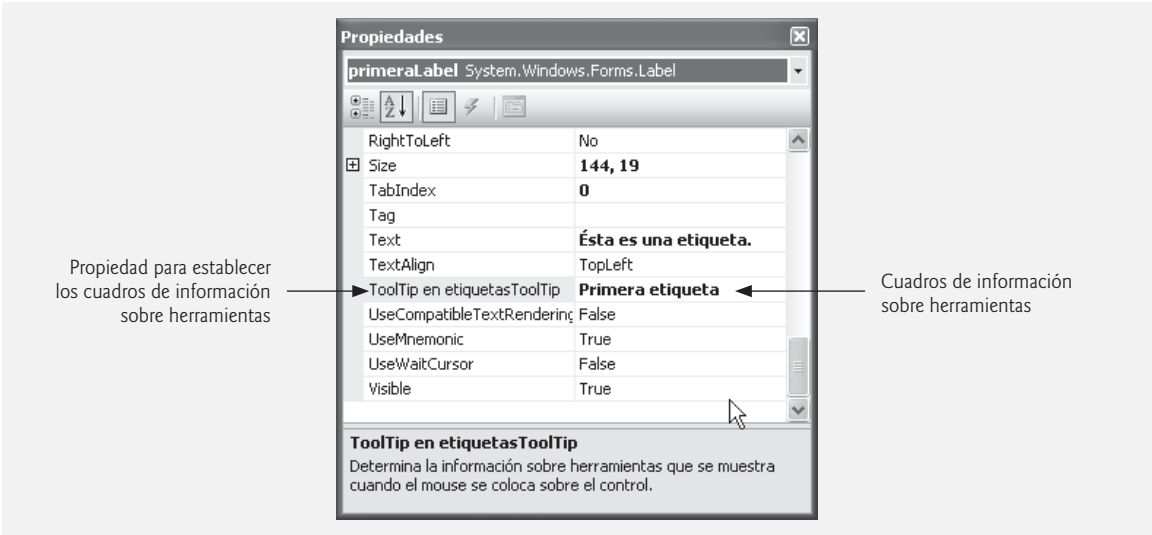


Figura 13.34 | Establecer el texto del cuadro de información sobre herramientas de un control.

13.10 Control NumericUpDown

A veces es conveniente restringir las opciones de entrada de un usuario a un rango específico de valores numéricos. Éste es el propósito del *control NumericUpDown*. Este control aparece como un control *TextBox*, con dos pequeños controles *Button* del lado derecho: uno con una flecha hacia arriba y el otro con una flecha hacia abajo. De manera predeterminada, un usuario puede escribir valores numéricos en este control como si fuera un control *TextBox*, o puede hacer clic en las flechas hacia arriba o hacia abajo para incrementar o decrementar el valor en el control, respectivamente. Los valores más grande y más pequeño del rango se especifican mediante las propiedades *Maximum* y *Minimum*, respectivamente (ambas de tipo *decimal*). La propiedad *Increment* (también de tipo *decimal*) especifica por cuánto cambia el número actual en el control cuando el usuario hace clic en las flechas hacia arriba o hacia abajo del control. La figura 13.35 describe las propiedades comunes y un evento común de la clase *NumericUpDown*.

Propiedades y evento de <i>NumericUpDown</i>	Descripción
<i>Propiedades comunes</i>	
<i>Increment</i>	Especifica por cuánto cambia el número actual en el control, cuando el usuario hace clic en las flechas hacia arriba y hacia abajo.
<i>Maximum</i>	El valor más grande en el rango del control.
<i>Minimum</i>	El valor más pequeño en el rango del control.
<i>UpDownAlign</i>	Modifica la alineación de los controles <i>Button</i> arriba y abajo en el control. <i>NumericUpDown</i> . Esta propiedad se puede utilizar para mostrar estos controles <i>Button</i> , ya sea a la izquierda o a la derecha del control.
<i>Value</i>	El valor numérico actual que se muestra en el control.
<i>Evento común</i>	
<i>ValueChanged</i>	Este evento se genera cuando cambia el valor en el control. Es el evento predeterminado para el control <i>NumericUpDown</i> .

Figura 13.35 | Propiedades y evento de *NumericUpDown*.

La figura 13.36 demuestra el uso de un control NumericUpDown para una GUI que calcula la tasa de interés. Los cálculos que se realizan en esta aplicación son similares a los que se realizan en la figura 6.6. Se utilizan controles TextBox para introducir los montos del capital y la tasa de interés, y un control NumericUpDown para introducir el número de años para los que deseamos calcular el interés.

Para el control NumericUpDown llamado anioUpDown, establecemos la propiedad Minimum en 1 y la propiedad Maximum en 10. Dejamos la propiedad Increment establecida en 1, su valor predeterminado. Estas configuraciones especifican que los usuarios pueden introducir un número de años en el rango de 1 a 10, en incrementos de 1. Si establecemos la propiedad Increment en 0.5, también podemos introducir valores como 1.5 o 2.5. Establecemos la *propiedad ReadOnly* del control NumericUpDown en true para indicar que el usuario no puede introducir un número en el control para realizar una selección. Por ende, el usuario debe hacer clic en las flechas hacia arriba o hacia abajo para modificar el valor en el control. De manera predeterminada, la propiedad ReadOnly se establece en false. Los resultados de esta aplicación se muestran en un control TextBox

```

1  // Fig. 13.36: calculadoraInteresesForm.cs
2  // Demostración del control NumericUpDown.
3  using System;
4  using System.Windows.Forms;
5
6  public partial class calculadoraInteresesForm : Form
7  {
8      // constructor predeterminado
9      public calculadoraInteresesForm()
10     {
11         InitializeComponent();
12     } // fin del constructor
13
14     private void calcularButton_Click(
15         object sender, EventArgs e )
16     {
17         // declara las variables para guardar entrada del usuario
18         decimal principal; // guarda principal
19         double tasa; // guarda tasa de interés
20         int anio; // guarda número de años
21         decimal monto; // guarda monto
22         string salida; // guarda salida
23
24         // extrae la entrada del usuario
25         principal = Convert.ToDecimal( principalTextBox.Text );
26         tasa = Convert.ToDouble( interesTextBox.Text );
27         anio = Convert.ToInt32( anioUpDown.Value );
28
29         // establece encabezado de salida
30         salida = "Año\tMonto en depósito\r\n";
31
32         // calcula el monto para cada año y lo adjunta a la salida
33         for ( int contadorAnios = 1; contadorAnios <= anio; contadorAnios++ )
34         {
35             monto = principal *
36                 ( ( decimal ) Math.Pow( ( 1 + tasa / 100 ), contadorAnios ) );
37             salida += ( contadorAnios + "\t" +
38                 String.Format( "{0:C}", monto ) + "\r\n" );
39         } // fin de for
40
41         mostrarTextBox.Text = salida; // muestra el resultado
42     } // fin del método calcularButton_Click
43 } // fin de la clase calculadoraInteresesForm

```

Figura 13.36 | Demostración del control NumericUpDown. (Parte I de 2).

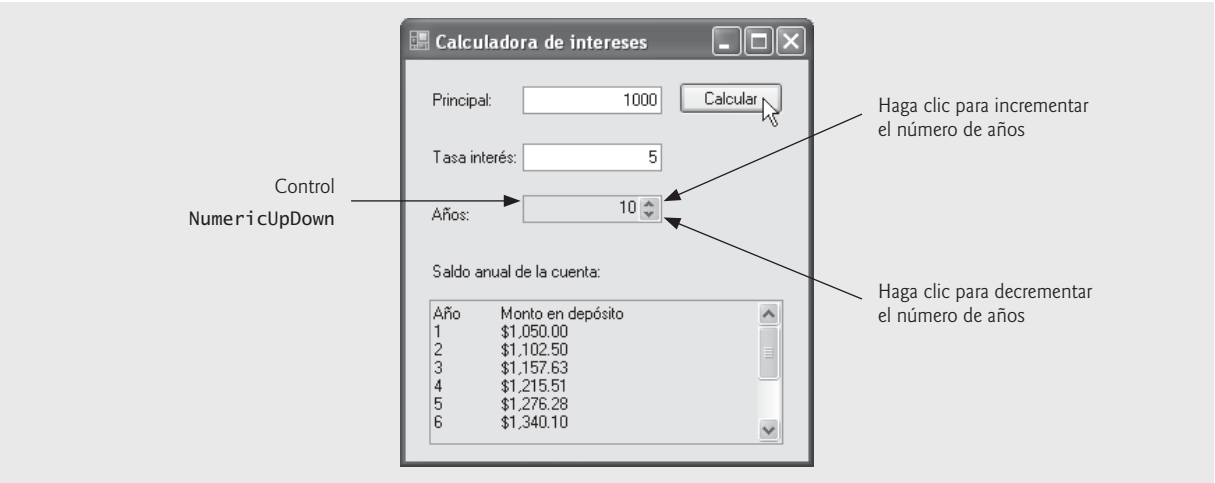


Figura 13.36 | Demostración del control `NumericUpDown`. (Parte 2 de 2).

multilínea de sólo lectura con una barra de desplazamiento vertical, para que el usuario pueda desplazarse por todos los resultados.

13.11 Manejo de los eventos del ratón

En esta sección explicamos cómo manejar los *eventos del ratón*, como *hacer clic*, *presionar* y *mover*, los cuales se generan cuando el usuario interactúa con un control a través del teclado. Los eventos del ratón se pueden manejar para cualquier control que se derive de la clase `System.Windows.Forms.Control`. Para la mayoría de los eventos del ratón, la información acerca del evento se pasa al método manejador de eventos, a través de un objeto de la clase `MouseEventArgs`, y el delegado que se utiliza para crear los manejadores de eventos del ratón es `EventHandler`. Cada método para manejar estos eventos del ratón requiere un `object` y un objeto `MouseEventArgs` como argumentos.

La clase `MouseEventArgs` contiene información relacionada con el evento del ratón, como las coordenadas *x* y *y* del puntero del ratón, el botón que se oprimió (`Right`, `Left` o `Middle`) y el número de veces que se hizo clic con el ratón. Observe que las coordenadas *x* y *y* del objeto `MouseEventArgs` son relativas para el control que generó el evento; es decir, el punto $(0,0)$ representa la esquina superior izquierda del control en donde ocurrió el evento de ratón. En la figura 13.37 se describen varios eventos comunes del ratón.

Eventos del ratón y argumentos de los eventos	
<i>Eventos del ratón con argumento del evento de tipo <code>EventArgs</code></i>	
<code>MouseEnter</code>	Ocurre cuando el cursor del ratón entra a los límites del control.
<code>MouseLeave</code>	Ocurre cuando el cursor del ratón sale de los límites del control.
<i>Eventos del ratón con argumento del evento de tipo <code>MouseEventArgs</code></i>	
<code>MouseDown</code>	Ocurre cuando se oprime un botón del ratón mientras el cursor se encuentra dentro de los límites de un control.
<code>MouseHover</code>	Ocurre cuando el cursor del ratón se coloca en los límites del control.
<code>MouseMove</code>	Ocurre cuando el cursor del ratón se desplaza dentro de los límites del control.
<code>MouseUp</code>	Ocurre cuando se suelta un botón del ratón, mientras el cursor se encuentra sobre los límites del control.

Figura 13.37 | Eventos del ratón y argumentos de los eventos. (Parte 1 de 2).

Eventos del ratón y argumentos de los eventos

Propiedades de la clase MouseEventArgs

Button	Especifica cuál botón del ratón se oprimió: Left (izquierdo), Right (derecho), Middle (central) o ninguno.
Clicks	El número de veces que se hizo clic con el botón del ratón.
X	La coordenada <i>x</i> dentro del control en el cual ocurrió el evento.
Y	La coordenada <i>y</i> dentro del control en el cual ocurrió el evento.

Figura 13.37 | Eventos del ratón y argumentos de los eventos. (Parte 2 de 2).

La figura 13.38 utiliza eventos del ratón para dibujar en un formulario. Cada vez que el usuario arrastra el ratón (es decir, cuando desplaza el ratón mientras mantiene oprimido un botón del mismo), aparecen pequeños círculos en el formulario, en la posición en la que ocurre cada evento del ratón durante la operación de arrastre.

En la línea 10, el programa declara la variable `debePintar`, la cual determina si se debe o no dibujar en el formulario. Queremos que el programa dibuje sólo mientras se mantiene oprimido el botón del ratón. Por ende,

```

1  // Fig 13.38: PintorForm.cs
2  // Uso del ratón para dibujar en un formulario.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // crea un formulario que actúa como una superficie de dibujo
8  public partial class PintorForm : Form
9  {
10     bool debePintar = false; // determina si va a pintar o no
11
12     // constructor predeterminado
13     public PintorForm()
14     {
15         InitializeComponent();
16     } // fin del constructor
17
18     // debe pintar cuando esté oprimido el botón del ratón
19     private void PintorForm_MouseDown( object sender, MouseEventArgs e )
20     {
21         // indica que el usuario está arrastrando el ratón
22         debePintar = true;
23     } // fin del método PintorForm_MouseDown
24
25     // deja de pintar cuando se suelta el botón del ratón
26     private void PintorForm_MouseUp( object sender, MouseEventArgs e )
27     {
28         // indica que el usuario soltó el botón del ratón
29         debePintar = false;
30     } // fin del método PintorForm_MouseUp
31
32     // dibuja un círculo cada vez que se mueve el ratón con el botón oprimido
33     private void PintorForm_MouseMove( object sender, MouseEventArgs e )
34     {

```

Figura 13.38 | Uso del ratón para dibujar en un formulario. (Parte 1 de 2).


```

35     if ( debePintar ) // comprueba si está oprimido el botón del ratón
36     {
37         // dibuja un círculo en donde se encuentra el puntero del ratón
38         Graphics graphics = CreateGraphics();
39         graphics.FillEllipse(
40             new SolidBrush( Color.BlueViolet ), e.X, e.Y, 4, 4 );
41         graphics.Dispose();
42     } // fin del if
43 } // fin del método PintorForm_MouseMove
44 } // fin de la clase PintorForm

```



Figura 13.38 | Uso del ratón para dibujar en un formulario. (Parte 2 de 2).

cuando el usuario hace clic o mantiene oprimido un botón del ratón, el sistema genera un evento `MouseDown` y el manejador de eventos (líneas 19-23) establece `debePintar` a `true`. Cuando el usuario suelta el botón del ratón, el sistema genera un evento `MouseUp`, `debePintar` se establece a `false` en el manejador de eventos `PintorForm_MouseUp` (líneas 26-30) y el programa deja de dibujar. A diferencia de los eventos `MouseMove`, que ocurren en forma continua a medida que el usuario mueve el ratón, el sistema genera un evento `MouseDown` sólo cuando se oprime un botón del ratón y genera un evento `MouseUp` cuando se suelta un botón.

Cada vez que el ratón se mueve por encima de un control, se produce el evento `MouseMove` para ese control. Dentro del manejador de eventos `PintorForm_MouseMove` (líneas 33-43), el programa dibuja sólo si `debePintar` es `true` (es decir, se oprimió un botón del ratón). La línea 38 llama al método `CreateGraphics` heredado de `Form` para crear un objeto **Graphics** que permita al programa dibujar en el formulario. La clase `Graphics` cuenta con métodos que dibujan varias figuras. Por ejemplo, las líneas 39-40 utilizan el método `FillEllipse` para dibujar un círculo. El primer parámetro para el método `FillEllipse` en este caso es un objeto de la clase `SolidBrush`, que especifica el color sólido que rellenará la figura. El color se proporciona como argumento para el constructor de la clase `SolidBrush`. El tipo `Color` contiene numerosas constantes de color predefinidas; nosotros seleccionamos `Color.BlueViolet`. `FillEllipse` dibuja un óvalo en un rectángulo delimitador que se especifica mediante las coordenadas *x* y *y* de su esquina superior izquierda, su anchura y su altura; los últimos cuatro argumentos para el método. Las coordenadas *x* y *y* representan la ubicación del evento del ratón y pueden tomarse de los argumentos del evento del ratón (*e.X* y *e.Y*). Para dibujar un círculo, establecemos la anchura y la altura del rectángulo delimitador, de manera que sean iguales; en este ejemplo, ambos tienen 4 píxeles. `Graphics`, `SolidBrush` y `Color` forman parte del espacio de nombres `System.Drawing`. En el capítulo 17 hablaremos detalladamente sobre la clase `Graphics` y sus métodos.

13.12 Manejo de los eventos del teclado

Los *eventos de tecla* ocurren cuando se oprimen y sueltan las teclas del teclado. Dichos eventos pueden controlarse para cualquier control que herede de `System.Windows.Forms.Control`. Hay tres eventos: `KeyPress`, `KeyUp` y `KeyDown`. El evento **KeyPress** ocurre cuando el usuario oprime una tecla que representa un carácter ASCII. La tecla específica puede determinarse mediante la propiedad `KeyChar` del argumento `KeyPressEventArgs` del

manejador de eventos. ASCII es un conjunto de 128 caracteres de símbolos alfanuméricos; en el apéndice D podrá encontrar un listado completo de estos símbolos.

El evento `KeyPress` no indica si se oprimieron *teclas modificadoras*. (Por ejemplo: *Mayús*, *Alt* y *Ctrl*) cuando ocurrió un evento. Si esta información es importante, pueden utilizarse los eventos **`KeyUp`** o **`KeyDown`**. El argumento **`KeyEventArgs`** para cada uno de esos eventos contiene información acerca de las teclas modificadoras. A menudo, estas teclas se utilizan en conjunto con el ratón, para seleccionar o resaltar información. La figura 13.39 lista información importante sobre los eventos de tecla. Varias propiedades devuelven valores de la *enumeración* **`Keys`**, la cual proporciona constantes que especifican las diversas teclas en un teclado. Al igual que la enumeración `FontStyle` (sección 13.7), la enumeración `Keys` es un objeto `System.FlagsAttribute`, por lo que las constantes de enumeración pueden combinarse para indicar que se oprimieron varias teclas al mismo tiempo.

La figura 13.40 demuestra el uso de los manejadores de eventos de tecla para mostrar la tecla que oprimió un usuario. El programa es un formulario con dos controles `Label`, que muestra la tecla oprimida en uno de los controles y la información sobre las teclas modificadoras en el otro.

Al principio, los dos controles `Label` (`characterLabel` e `infoTeclaLabel`) están vacíos. El control `characterLabel` muestra el valor del carácter de la tecla oprimida, mientras que `teclaInfoLabel` muestra información relacionada con la tecla oprimida. Debido a que los eventos `KeyDown` y `KeyPress` transmiten distinta información, el formulario (`TeclasDemoForm`) maneja ambos.

El manejador de eventos `KeyPress` (líneas 16-19) accede a la propiedad `KeyChar` del objeto `KeyPressEventArgs`. Esto devuelve la tecla oprimida como un carácter, el cual mostramos después en `characterLabel` (línea 18). Si la tecla oprimida no es un carácter ASCII, entonces el evento `KeyPress` no ocurrirá y `characterLabel` no mostrará ningún texto. ASCII es un formato común de codificación para letras, números, signos de puntuación y otros caracteres. No acepta teclas tales como las *teclas de función* (como *F1*) o las teclas modificadoras (*Alt*, *Ctrl* y *Mayús*).

El manejador de eventos `KeyDown` (líneas 22-31) muestra información de su objeto `EventArgs`. El manejador de eventos evalúa las teclas *Alt*, *Mayús* y *Ctrl* mediante el uso de las propiedades `Alt`, `Shift` y `Control`, cada una de las cuales devuelve un valor `bool`: `true` si se oprime la tecla correspondiente y `false` en caso contrario. Después, el manejador de eventos muestra las propiedades `KeyCode`, `KeyData` y `KeyValue`.

La propiedad `KeyCode` devuelve un valor de la enumeración `Keys` (línea 28). La propiedad `KeyCode` devuelve la tecla oprimida, pero no proporciona información acerca de las teclas modificadoras. Por ende, la letra a mayúscula y la letra a minúscula se representan como la tecla *A*.

La propiedad `KeyData` (línea 29) también devuelve un valor de la enumeración `Keys`, pero esta propiedad incluye datos acerca de las teclas modificadoras. Por ende, si se introduce “A”, la propiedad `KeyData` muestra que se oprimió tanto la tecla *A* como la tecla *Mayús*. Por último, `KeyValue` (línea 30) devuelve el código de la tecla oprimida como un `int`. Este `int` es el *código de tecla*, el cual proporciona un valor `int` para un amplio rango de teclas y para los botones del ratón. El código de tecla virtual de Windows es útil cuando se prueban las teclas que no son ASCII (como *F12*).

Eventos del teclado y argumentos de los eventos

Eventos de tecla con argumentos de tipo `KeyEventArgs`

<code>KeyDown</code>	Se genera cuando se oprime por primera vez una tecla.
<code>KeyUp</code>	Se genera cuando se suelta una tecla.

Evento de tecla con argumento de tipo `KeyPressEventArgs`

<code>KeyPress</code>	Se genera cuando se oprime una tecla.

Propiedades de la clase `KeyPressEventArgs`

<code>KeyChar</code>	Devuelve el carácter ASCII para la tecla oprimida.
<code>Handled</code>	Indica si se manejó el evento <code>KeyPress</code> .

Figura 13.39 | Eventos del teclado y argumentos de los eventos. (Parte I de 2).

Eventos del teclado y argumentos de los eventos

Propiedades de la clase EventArgs

Alt	Indica si se oprimió la tecla <i>Alt</i> .
Control	Indica si se oprimió la tecla <i>Ctrl</i> .
Shift	Indica si se oprimió la tela <i>Mayús</i> .
Handled	Indica si se manejó el evento.
KeyCode	Devuelve el código de la tecla que se oprimió, en forma de un valor de la enumeración <i>Keys</i> . Esto no incluye información sobre las teclas modificadoras. Se utiliza para evaluar una tecla específica.
KeyData	Devuelve el código de la tecla que se oprimió, en combinación con la información de las teclas modificadoras en forma de un valor <i>Keys</i> . Esta propiedad contiene toda la información acerca de la tecla oprimida.
KeyValue	Devuelve el código de tecla como un <i>int</i> , en vez de devolverlo como un valor de la enumeración <i>Keys</i> . Esta propiedad se utiliza para obtener una representación numérica de la tecla oprimida. El valor <i>int</i> se conoce como código de tecla virtual de Windows.
Modifiers	Devuelve un valor <i>Keys</i> que indica si se oprimieron teclas modificadoras (<i>Alt</i> , <i>Ctrl</i> y <i>Mayús</i>). Esta propiedad sólo se utiliza para determinar la información sobre las teclas modificadoras.

Figura 13.39 | Eventos del teclado y argumentos de los eventos. (Parte 2 de 2).

```

1  // Fig. 13.40: TeclasDemoForm.cs
2  // Muestra información acerca de la tecla que oprimió el usuario.
3  using System;
4  using System.Windows.Forms;
5
6  // formulario para mostrar información sobre la tecla que se oprimió
7  public partial class TeclasDemoForm : Form
8  {
9      // constructor predeterminado
10     public TeclasDemoForm()
11     {
12         InitializeComponent();
13     } // fin del constructor
14
15     // muestra el caracter que se oprimió, usando KeyChar
16     private void TeclasDemoForm_KeyPress( object sender, KeyPressEventArgs e )
17     {
18         caracterLabel.Text = "Se oprimió la tecla: " + e.KeyChar;
19     } // fin del método TeclaDemo_KeyPress
20
21     // muestra teclas modificadoras, código de tecla, datos de tecla y su valor
22     private void TeclasDemoForm_KeyDown( object sender, EventArgs e )
23     {
24         infoTeclaLabel.Text =
25             "Alt: " + ( e.Alt ? "Sí" : "No" ) + '\n' +
26             "Mayús: " + ( e.Shift ? "Sí" : "No" ) + '\n' +
27             "Ctrl: " + ( e.Control ? "Sí" : "No" ) + '\n' +
28             "Código tecla: " + e.KeyCode + '\n' +
29             "Datos tecla: " + e.KeyData + '\n' +
30             "Valor de tecla: " + e.KeyValue;
31     } // fin del método TeclasDemo_KeyDown

```

Figura 13.40 | Demostración de los eventos de teclado. (Parte 1 de 2).

```

32
33 // borra controles Label cuando se suelta la tecla
34 private void TeclasDemoForm_KeyUp( object sender, EventArgs e )
35 {
36     caracterLabel.Text = "";
37     infoTeclaLabel.Text = "";
38 } // fin del método TeclaDemo_KeyUp
39 } // fin de la clase TeclasDemoForm

```



Figura 13.40 | Demostración de los eventos de teclado. (Parte 2 de 2).

El manejador de eventos `KeyUp` (líneas 34-38) borra ambos controles `Label` cuando se suelta la tecla. Como podemos ver de la salida, las teclas que no son ASCII no se muestran en `caracterLabel`, debido a que no se genera el evento `KeyPress`. No obstante, sí se genera el evento `KeyDown`, por lo que `teclaInfoLabel` muestra información acerca de la tecla que se oprimió. La enumeración `Keys` puede utilizarse para evaluar teclas específicas, comparando la tecla oprimida con un valor de `KeyCode` específico.



Observación de ingeniería de software 13.3

Para hacer que un control reaccione cuando se oprime cierta tecla (como `Intro`), utilice un manejador de eventos para evaluar la tecla oprimida. Para hacer que se haga clic en un control `Button` cuando se oprime `Intro` en un formulario, establezca la propiedad `AcceptButton` del formulario.

13.13 Conclusión

En este capítulo presentamos varios controles comunes de la GUI. Hablamos con detalle sobre el manejo de eventos y le mostramos cómo crear manejadores de eventos. También hablamos sobre la manera en que se utilizan los delegados para conectar manejadores de eventos a los eventos de controles específicos. Aprendió a utilizar las propiedades de un control y Visual Studio para especificar la distribución de su GUI. Después demostramos el uso de varios controles, empezando con los controles `Label`, `Button` y `TextBox`. Aprendió a utilizar controles `Group-`

Box y Panel para organizar otros controles. Después demostramos el uso de los controles CheckBox y RadioButton, los cuales son botones de estado que permiten a los usuarios seleccionar de entre varias opciones. Mostramos imágenes en los controles PictureBox, mostramos texto de ayuda en una GUI mediante los componentes ToolTip y especificamos un rango de valores de entrada para los usuarios con un control NumericUpDown. Después demostramos cómo manejar los eventos del ratón y del teclado. En el siguiente capítulo presentaremos controles adicionales de la GUI. Usted aprenderá a agregar menús a sus GUIs y crear aplicaciones Windows que muestren varios formularios.

14

Conceptos de interfaz gráfica de usuario: parte 2

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Crear menús, ventanas con fichas y programas con interfaz de múltiples documentos (MDI).
- Utilizar los controles `ListView` y `TreeView` para mostrar información.
- Crear hipervínculos mediante el uso del control `LinkLabel`.
- Mostrar listas de información en controles `ListBox` y `ComboBox`.
- Introducir datos de fecha y hora con el control `DateTimePicker`.
- Crear controles personalizados.

Yo no aclamo haber controlado los eventos, sino que confieso plenamente que los eventos me han controlado a mí.

—Abraham Lincoln

¡Captura su realidad en pintura!

—Paul Cézanne

Si un actor entra por la puerta, no tienes nada. Pero si entra por la ventana, tienes un problema.

—Billy Wilder

¿Qué luz es la que asoma por aquella ventana? ¡Es el Oriente! ¡Y Julieta es el Sol!

—William Shakespeare

- 14.1 Introducción
- 14.2 Menús
- 14.3 Control MonthCalendar
- 14.4 Control DateTimePicker
- 14.5 Control LinkLabel
- 14.6 Control ListBox
- 14.7 Control CheckedListBox
- 14.8 Control ComboBox
- 14.9 Control TreeView
- 14.10 Control ListView
- 14.11 Control TabControl
- 14.12 Ventanas de la interfaz de múltiples documentos (MDI)
- 14.13 Herencia visual
- 14.14 Controles definidos por el usuario
- 14.15 Conclusión

14.1 Introducción

Este capítulo continúa con nuestro estudio acerca de las GUIs. Empezaremos nuestra discusión con los menús, que presentan a los usuarios una serie de comandos (u opciones) organizados en forma lógica. A continuación hablaremos sobre cómo introducir y mostrar fechas y horas mediante los controles `MonthCalendar` y `DateTimePicker`. Mostraremos cómo desarrollar menús con las herramientas que proporciona Visual Studio. También presentaremos los controles `LinkLabel`: poderosos componentes de la GUI que permiten al usuario visitar uno de varios destinos, como un archivo en el equipo actual o una página Web, con sólo hacer clic con el ratón.

Mostraremos cómo manipular una lista de valores mediante un control `ListBox`, y cómo combinar varias casillas de verificación (controles `CheckBox`) en un control `CheckedListBox`. También crearemos listas desplegables mediante el uso de controles `ComboBox` y mostraremos datos en forma jerárquica con un control `TreeView`. Aprenderá a usar otros dos componentes importantes de la GUI: controles de tabulación y ventanas de la interfaz de múltiples documentos (MDI). Estos componentes le permiten crear programas reales con GUIs sofisticadas.

Visual Studio cuenta con un extenso conjunto de componentes de la GUI, varios de los cuales veremos en este capítulo (y en el anterior). Visual Studio también nos permite diseñar controles personalizados y agregarlos al **Cuadro de herramientas**, como demostraremos en el último ejemplo de este capítulo. Las técnicas que presentamos aquí forman los cimientos para crear GUIs y controles personalizados más robustos.

14.2 Menús

Los *menús* proporcionan grupos de comandos relacionados para aplicaciones Windows. Aunque éstos dependen del programa, algunos (como **Abrir** y **Guardar**) son comunes para muchas aplicaciones. Los menús son una parte integral de las GUIs, ya que organizan comandos sin “atestar” la GUI.

En la figura 14.1, un menú de Visual Studio expandido muestra varios comandos (conocidos como *elementos de menú*) y también *submenús* (menús dentro de un menú). Observe que los menús de nivel superior aparecen en la parte izquierda de la figura, mientras que los submenús o elementos de menú se muestran a la derecha. Al menú que contiene un elemento de menú se le conoce como el *menú padre* de ese elemento. Un elemento de menú que contiene un submenú se considera como el padre de ese submenú.

Todos los menús pueden tener métodos de acceso rápido a sus elementos con la tecla *Alt* (también conocidos como *métodos de acceso abreviado*, *teclas de acceso rápido* o “*shortcuts*”), a las cuales se accede oprimiendo *Alt* y la letra subrayada (por ejemplo, comúnmente *Alt + A* expande el menú **Archivo**). Los menús que no son de nivel superior pueden tener teclas de acceso rápido también (combinaciones de *Ctrl*, *Mayús*, *Alt*, *F1*, *F2*, teclas de letras,

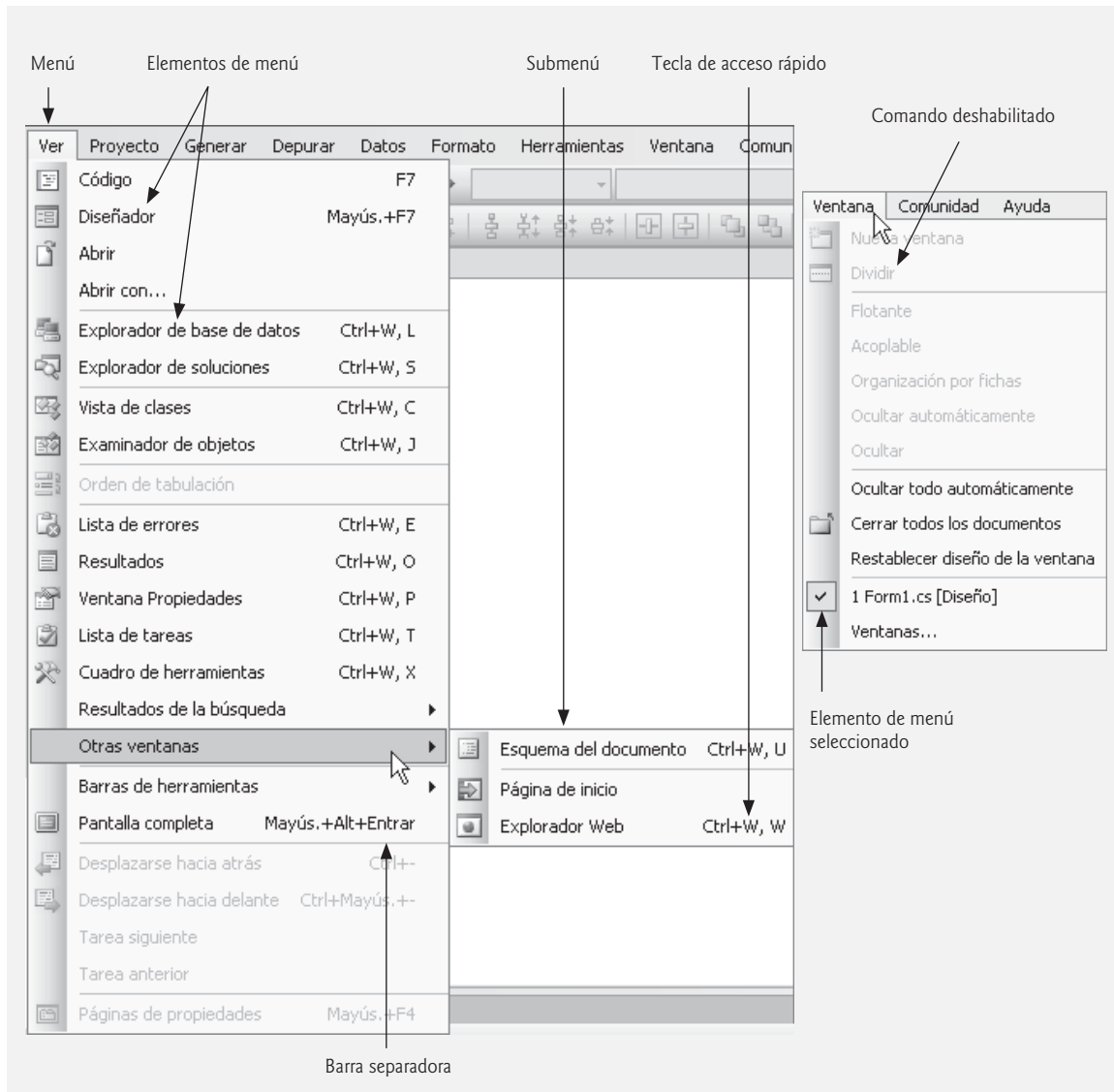


Figura 14.1 | Menús, submenús y elementos de menú.

etc.). Algunos elementos de menú muestran marcas de verificación, que por lo general indican que pueden seleccionarse varias opciones en el menú a la vez.

Para crear un menú, abra el **Cuadro de herramientas** y arrastre un control **MenuStrip** en el formulario. Esto crea una barra de menú a lo largo de la parte superior del formulario (debajo de la barra de títulos) y coloca un icono **MenuStrip** en la bandeja de componentes. Para seleccionar el control **MenuStrip**, haga clic en este icono. Ahora puede usar el modo de **Diseño** para crear y editar menús para su aplicación. Al igual que otros controles, los menús tienen propiedades y eventos, a los cuales se puede acceder a través de la ventana **Propiedades**.

Para agregar elementos al menú, haga clic en el cuadro de texto **Escriba aquí** (figura 14.2) y escriba el nombre del elemento del menú. Esta acción agrega una entrada al menú de tipo **ToolStripMenuItem**. Una vez que usted oprima la tecla **Intro**, se agregará al menú el nombre del elemento de menú. Después aparecerán más cuadros de texto **Escriba aquí**, para permitirle agregar elementos por debajo o al lado del elemento de menú original (figura 14.3).

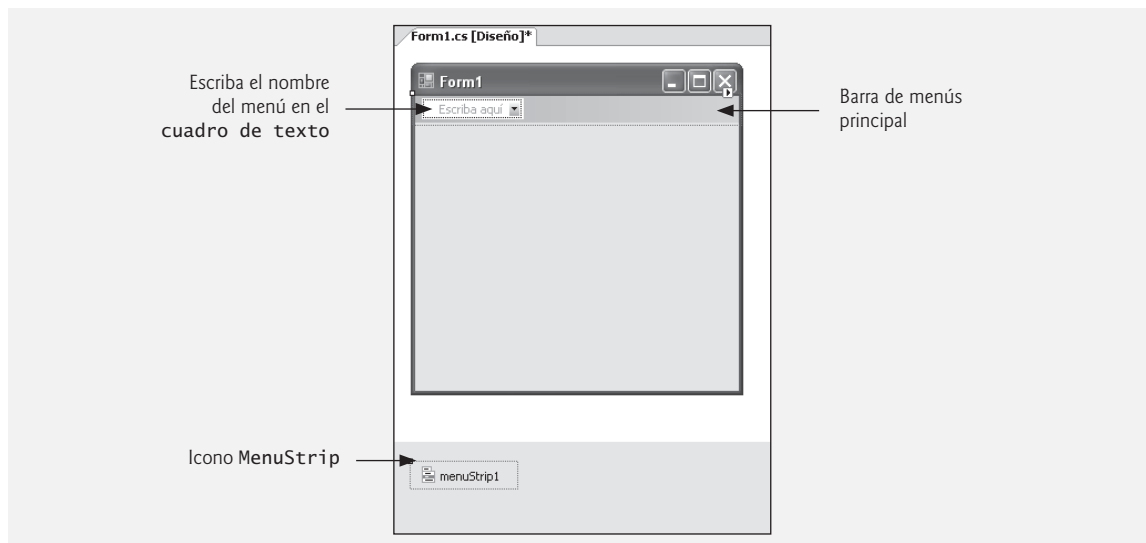


Figura 14.2 | Edición de menús en Visual Studio.

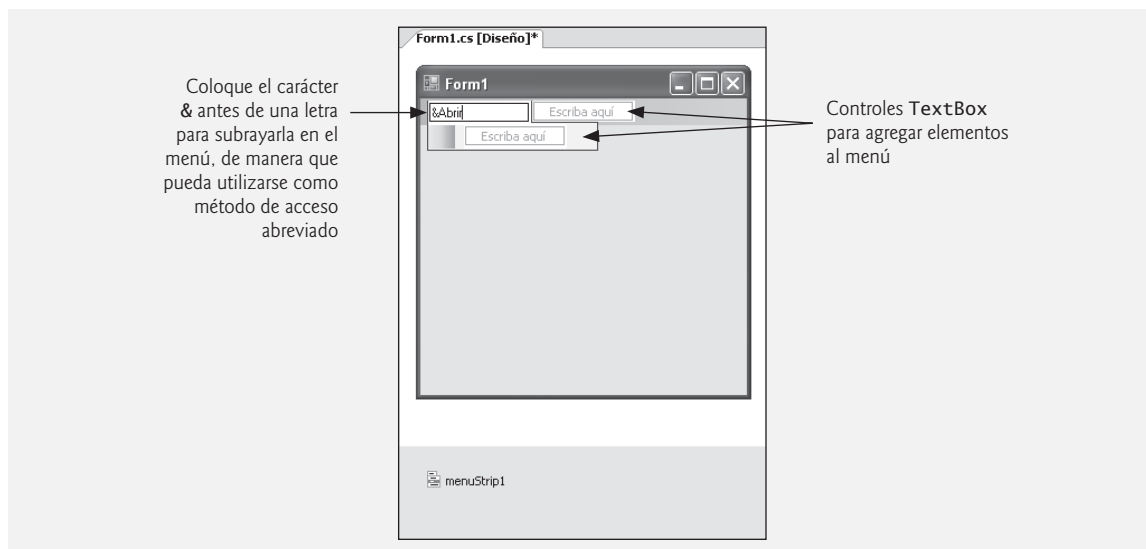


Figura 14.3 | Agregar elementos ToolStripMenuItem a un control MenuStrip.

Para crear un *método de acceso abreviado* (o *método abreviado de teclado*), escriba un signo & antes del carácter a subrayar. Por ejemplo, para crear el elemento de menú **Archivo** con la letra **A** subrayada, escriba &Archivo. Para mostrar un signo &, escriba &&. Para agregar otras teclas de acceso abreviado (por ejemplo, <Ctrl>-F9) para los elementos de menú, establezca la propiedad **ShortcutKeys** del elemento ToolStripMenuItem apropiado. Para hacer esto, seleccione la flecha hacia abajo que se encuentra a la derecha de esta propiedad en la ventana **Propiedades**. En la ventana que aparezca (figura 14.4), utilice los controles CheckBox y la lista desplegable para seleccionar las teclas de método abreviado. Cuando termine, haga clic en cualquier otra parte de la pantalla. Para ocultar las teclas de método abreviado, establezca la propiedad **ShowShortcutKeys** a false y para modificar la manera en que se muestran las teclas de control en el elemento de menú, modifique la propiedad **ShortcutKeyDisplayString**.

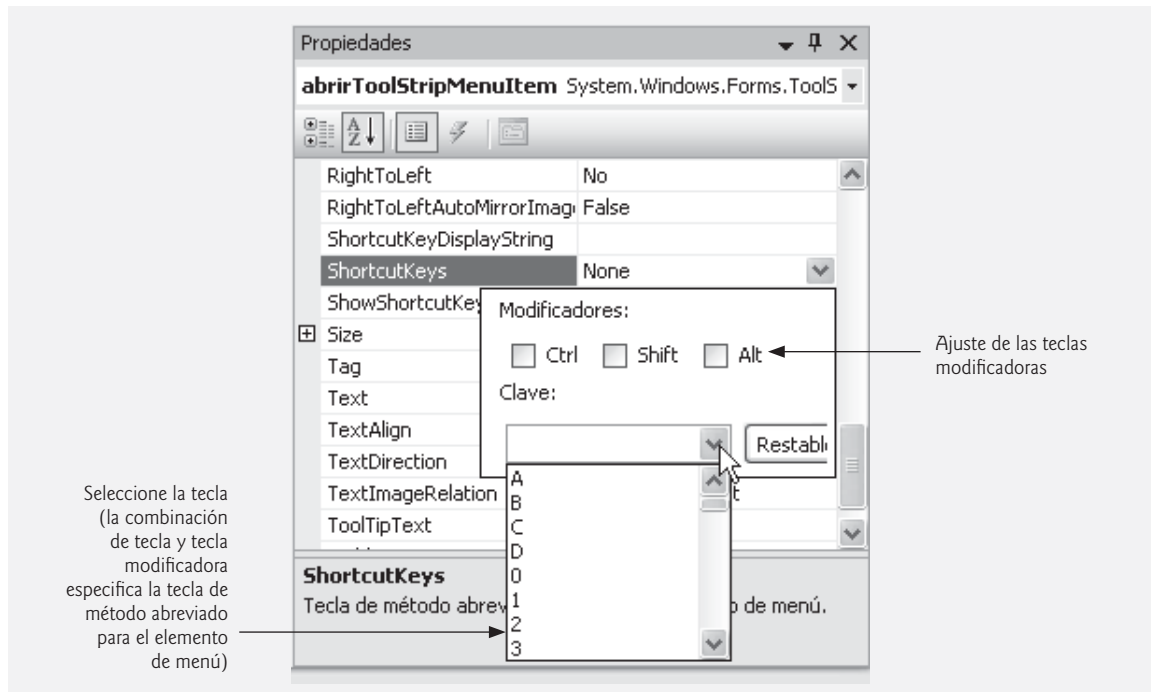


Figura 14.4 | Establecer las teclas de método abreviado de un elemento de menú.



Observación de apariencia visual 14.1

Los controles **Button** pueden tener métodos de acceso abreviados. Coloque el símbolo **&** justo antes del carácter deseado en la etiqueta del control **Button**. Para oprimir el botón usando su tecla de acceso en la aplicación en ejecución, el usuario debe oprimir **Alt** y el carácter subrayado.

Para eliminar un elemento de menú, selecciónelo con el ratón y oprima **Supr.** Los elementos de menú pueden agruparse en forma lógica mediante **barras separadoras**, las cuales se insertan haciendo clic con el botón derecho del ratón en el menú y seleccionando **Insertar Separador** o escribiendo “-” en el texto de un elemento de menú.

Además del texto, Visual Studio le permite agregar con facilidad controles **TextBox** y **ComboBox** (listas desplegables) como elementos de menú. Al agregar un elemento en modo de **Diseño**, tal vez se haya dado cuenta de que antes de escribir el texto para un nuevo elemento, aparece una lista desplegable. Al hacer clic en la flecha hacia abajo (figura 14.5), usted puede seleccionar el tipo de elemento que desea agregar: **Menu Item** (de tipo **ToolStripMenuItem**, el valor predeterminado), **ComboBox** (de tipo **ToolStripComboBox**) y **TextBox** (de tipo **ToolStripTextBox**). Nos enfocaremos en los elementos **ToolStripMenuItem**. [Nota: si ve esta lista desplegable para los elementos de menú que no se encuentran en el nivel superior, aparecerá una cuarta opción, que le permite insertar una barra separadora.]

Los elementos **ToolStripMenuItem** generan un evento **Click** cuando se seleccionan. Para crear un manejador de eventos **Click** vacío, haga doble clic en el elemento de menú en modo de **Diseño**. Las acciones comunes en respuesta a estos eventos incluyen el mostrar cuadros de diálogo y establecer propiedades. En la figura 14.6 se muestra un resumen de las propiedades comunes de los menús y un evento común.



Observación de apariencia visual 14.2

Es una convención colocar una **elipsis (...)** después del nombre de un elemento de menú que, cuando se selecciona, muestra un cuadro de diálogo (por ejemplo, **Guardar como...**). Los elementos de menú que producen una acción inmediata sin pedir al usuario más información (por ejemplo, **Guardar**) no deben tener una **elipsis** después de su nombre.

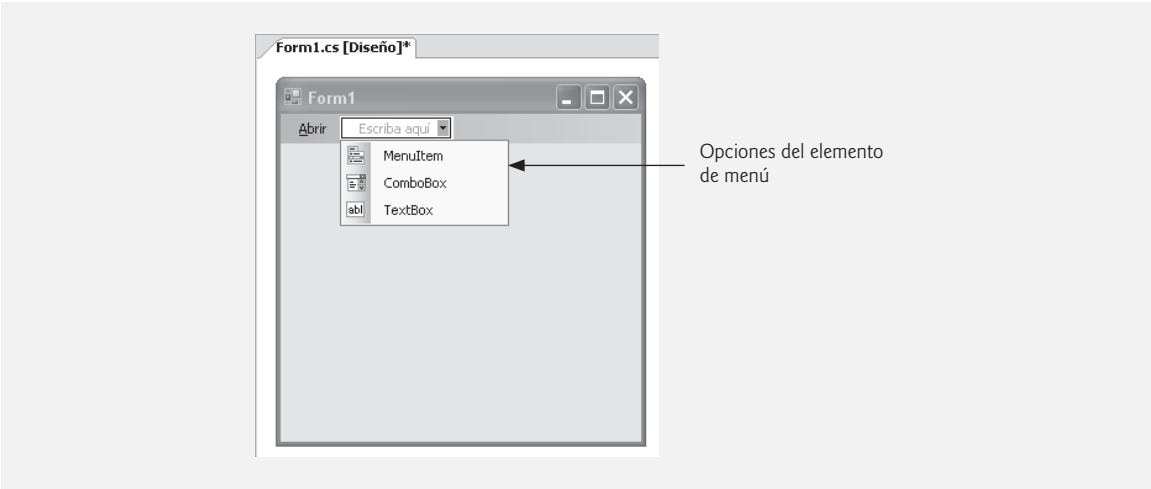


Figura 14.5 | Opciones del elemento de menú.

Propiedades y evento de MenuStrip y ToolStripMenuItem	
	Descripción
<i>Propiedades de MenuStrip</i>	
MenuStrip	Contiene los elementos de menú de nivel superior para este control MenuStrip.
HasChildren	Indica si el control MenuStrip tiene controles hijos (elementos de menú).
RightToLeft	Hace que se muestre el texto de derecha a izquierda. Esto es útil para los lenguajes que se leen de derecha a izquierda.
<i>Propiedades de ToolStripMenuItem</i>	
Checked	Indica si está seleccionado un elemento de menú. El valor predeterminado es false, lo que significa que el elemento de menú no está seleccionado.
CheckOnClick	Indica que un elemento de menú debe aparecer seleccionado o deseleccionado, a medida que se hace clic en el elemento.
Index	Especifica la posición de un elemento en su menú padre. Un valor de 0 coloca el elemento MenuItem al principio del menú.
MenuItem	Lista los elementos del submenú para un elemento de menú específico.
ShortcutKey-DisplayString	Especifica el texto que debe aparecer al lado de un elemento de menú para una tecla de método abreviado. Si se deja en blanco, se muestran los nombres de las teclas. En caso contrario, se muestra el texto en esta propiedad para la tecla de método abreviado.
ShortcutKeys	Especifica la tecla de método abreviado para el elemento de menú (por ejemplo, <Ctrl>-F9 equivale a hacer clic en un elemento específico).
ShowShortcutKeys	Indica si se muestra una tecla de método abreviado al lado del texto del elemento de menú. El valor predeterminado es true, el cual muestra la tecla de método abreviado.
Text	Especifica el texto del elemento de menú. Para crear un método de acceso abreviado con Alt, coloque el signo & antes de un carácter (por ejemplo, &Archivo para especificar un menú llamado Archivo con la letra A subrayada).

Figura 14.6 | Propiedades y evento de MenuStrip y ToolStripMenuItem. (Parte 1 de 2).

Propiedades y evento de ToolStripMenuItem	Descripción
<i>Evento común de ToolStripMenuItem</i>	
Click	Se genera cuando se hace clic en un elemento, o cuando se muestra una tecla de método abreviado. Éste es el evento predeterminado cuando se hace doble clic en el menú, estando en el diseñador.

Figura 14.6 | Propiedades y evento de ToolStripMenuItem. (Parte 2 de 2).

La clase `MenuPruebaForm` (figura 14.7) crea un menú simple en un formulario. Este formulario tiene un menú **Archivo** de nivel superior, con los elementos de menú **Acerca de** (el cual muestra un cuadro de diálogo `MessageBox`) y **Salir** (que termina el programa). El programa también incluye un menú **Formato**, que contiene elementos de menú que modifican el formato del texto en un control `Label`. El menú **Formato** tiene los submenús **Color** y **Fuente**, que modifican el color y la fuente del texto en un control `Label`.

Para crear esta GUI, empiece arrastrando el control `MenuStrip` del **Cuadro de herramientas** hacia el formulario. Después use el modo de **Diseño** para crear la estructura de menús que se muestra en las ventanas de resultados de ejemplo. El menú **Archivo** (`archivoToolStripMenuItem`) tiene los elementos de menú **Acerca de** (`acercaDeToolStripMenuItem`) y **Salir** (`salirToolStripMenuItem`); el menú **Formato** (`formatoToolStripMenuItem`) tiene dos submenús. El primer submenú, **Color** (`colorToolStripMenuItem`), contiene los elementos de menú **Negro** (`negroToolStripMenuItem`), **Azul** (`azulToolStripMenuItem`), **Rojo** (`rojoToolStripMenuItem`) y **Verde** (`verdeToolStripMenuItem`). El segundo submenú, **Fuente** (`fuentesToolStripMenuItem`), contiene los elementos de menú **Times New Roman** (`timesToolStripMenuItem`), **Courier** (`courierToolStripMenuItem`), **Comic Sans** (`comicToolStripMenuItem`), una barra separadora (`barraToolStripMenuItem`), **Negrita** (`negritaToolStripMenuItem`) y **Cursiva** (`cursivaToolStripMenuItem`).

Cuando hacemos clic en el elemento de menú **Acerca de** en el menú **Archivo**, aparece un cuadro de diálogo `MessageBox` (líneas 18-23). El elemento de menú **Salir** cierra la aplicación a través del método `static Exit` de la clase `Application` (línea 28). Los métodos `static` de la clase `Application` controlan la ejecución del programa. El método `Exit` hace que nuestra aplicación termine.

```

1  // Fig. 14.7: MenuPruebaForm.cs
2  // Uso de menús para cambiar los colores y estilos de las fuentes.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // nuestro formulario contiene un menú que cambia el color y el
8  // estilo de la fuente del texto mostrado en el control Label
9  public partial class MenuPruebaForm : Form
10 {
11     // constructor predeterminado
12     public MenuPruebaForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     // muestra MessageBox cuando se selecciona el MenuItem Acerca de
18     private void acercaDeToolStripMenuItem_Click( object sender, EventArgs e )
19     {
20         MessageBox.Show(

```

Figura 14.7 | Menús para modificar la fuente y el color del texto. (Parte 1 de 4).

```

21         "Éste es un ejemplo\nde\ uso de menús.",
22         "Acerca de", MessageBoxButtons.OK, MessageBoxIcon.Information );
23     } // fin del método acercaDeMenuItem_Click
24
25     // sale del programa cuando se selecciona el MenuItem Salir
26     private void salirToolStripMenuItem_Click( object sender, EventArgs e )
27     {
28         Application.Exit();
29     } // fin del método salirToolStripMenuItem_Click
30
31     // restablece marcas de verificación para los MenuItems del color
32     private void BorrarColor()
33     {
34         // borra todas las marcas de verificación
35         negroToolStripMenuItem.Checked = false;
36         azulToolStripMenuItem.Checked = false;
37         rojoToolStripMenuItem.Checked = false;
38         verdeToolStripMenuItem.Checked = false;
39     } // fin del método BorrarColor
40
41     // actualiza estado del menú y color negro para la fuente
42     private void negroToolStripMenuItem_Click( object sender, EventArgs e )
43     {
44         // restablece marcas de verificación para los MenuItems de color
45         BorrarColor();
46
47         // establece Color a Negro
48         mostrarLabel.ForeColor = Color.Black;
49         negroToolStripMenuItem.Checked = true;
50     } // fin del método negroToolStripMenuItem_Click
51
52     // actualiza estado del menú y color azul para la fuente
53     private void azulToolStripMenuItem_Click( object sender, EventArgs e )
54     {
55         // restablece marcas de verificación para los MenuItems de color
56         BorrarColor();
57
58         // establece Color a Azul
59         mostrarLabel.ForeColor = Color.Blue;
60         azulToolStripMenuItem.Checked = true;
61     } // fin del método azulToolStripMenuItem_Click
62
63     // actualiza estado del menú y color rojo para la fuente
64     private void rojoToolStripMenuItem_Click( object sender, EventArgs e )
65     {
66         // restablece marcas de verificación para los MenuItems de color
67         BorrarColor();
68
69         // establece Color a Rojo
70         mostrarLabel.ForeColor = Color.Red;
71         rojoToolStripMenuItem.Checked = true;
72     } // fin del método rojoToolStripMenuItem_Click
73
74     // actualiza estado del menú y color verde para la fuente
75     private void verdeToolStripMenuItem_Click( object sender, EventArgs e )
76     {
77         // restablece marcas de verificación para los MenuItems de color
78         BorrarColor();

```

Figura 14.7 | Menús para modificar la fuente y el color del texto. (Parte 2 de 4).

```

79
80     // establece Color a Verde
81     mostrarLabel.ForeColor = Color.Green;
82     verdeToolStripMenuItem.Checked = true;
83 } // fin del método verdeToolStripMenuItem_Click
84
85 // restablece marcas de verificación para los MenuItems de la fuente
86 private void BorrarFuente()
87 {
88     // borra todas las marcas de verificación
89     timesToolStripMenuItem.Checked = false;
90     courierToolStripMenuItem.Checked = false;
91     comicToolStripMenuItem.Checked = false;
92 } // fin del método BorrarFuente
93
94 // actualiza estado del menú y establece Fuente a Times New Roman
95 private void timesToolStripMenuItem_Click( object sender, EventArgs e )
96 {
97     // restablece marcas de verificación para los MenuItems de la Fuente
98     BorrarFuente();
99
100    // establece fuente Times New Roman
101    timesToolStripMenuItem.Checked = true;
102    mostrarLabel.Font = new Font(
103        "Times New Roman", 14, mostrarLabel.Font.Style );
104 } // fin del método timesToolStripMenuItem_Click
105
106 // actualiza estado del menú y establece la Fuente a Courier
107 private void courierToolStripMenuItem_Click(
108     object sender, EventArgs e )
109 {
110     // restablece marcas de verificación para los MenuItems de la Fuente
111     BorrarFuente();
112
113     // establece fuente Courier
114     courierToolStripMenuItem.Checked = true;
115     mostrarLabel.Font = new Font(
116         "Courier", 14, mostrarLabel.Font.Style );
117 } // fin del método courierToolStripMenuItem_Click
118
119 // actualiza estado del menú y establece Fuente a Comic Sans MS
120 private void comicToolStripMenuItem_Click( object sender, EventArgs e )
121 {
122     // restablece marcas de verificación para los MenuItems de la Fuente
123     BorrarFuente();
124
125     // establece fuente Comic Sans
126     comicToolStripMenuItem.Checked = true;
127     mostrarLabel.Font = new Font(
128         "Comic Sans MS", 14, mostrarLabel.Font.Style );
129 } // fin del método comicToolStripMenuItem_Click
130
131 // alterna marca de verificación y activa estilo negrita
132 private void negritaToolStripMenuItem_Click( object sender, EventArgs e )
133 {
134     // alterna marca de verificación
135     negritaToolStripMenuItem.Checked = !negritaToolStripMenuItem.Checked;
136

```

Figura 14.7 | Menús para modificar la fuente y el color del texto. (Parte 3 de 4).

```

137 // usa or para alternar negrita, mantiene los demás estilos
138 mostrarLabel.Font = new Font(
139     mostrarLabel.Font.FontFamily, 14,
140     mostrarLabel.Font.Style ^ FontStyle.Bold );
141 } // fin del método negritaToolStripMenuItem_Click
142
143 // alterna marca de verificación y alterna estilo cursiva
144 private void cursivaToolStripMenuItem_Click(
145     object sender, EventArgs e )
146 {
147     // alterna marca de verificación
148     cursivaToolStripMenuItem.Checked = !cursivaToolStripMenuItem.Checked;
149
150     // usa or para alternar cursiva, mantiene los demás estilos
151     mostrarLabel.Font = new Font(
152         mostrarLabel.Font.FontFamily, 14,
153         mostrarLabel.Font.Style ^ FontStyle.Italic );
154 } // fin del método cursivaToolStripMenuItem_Click
155 } // fin de la clase MenuPruebaForm

```

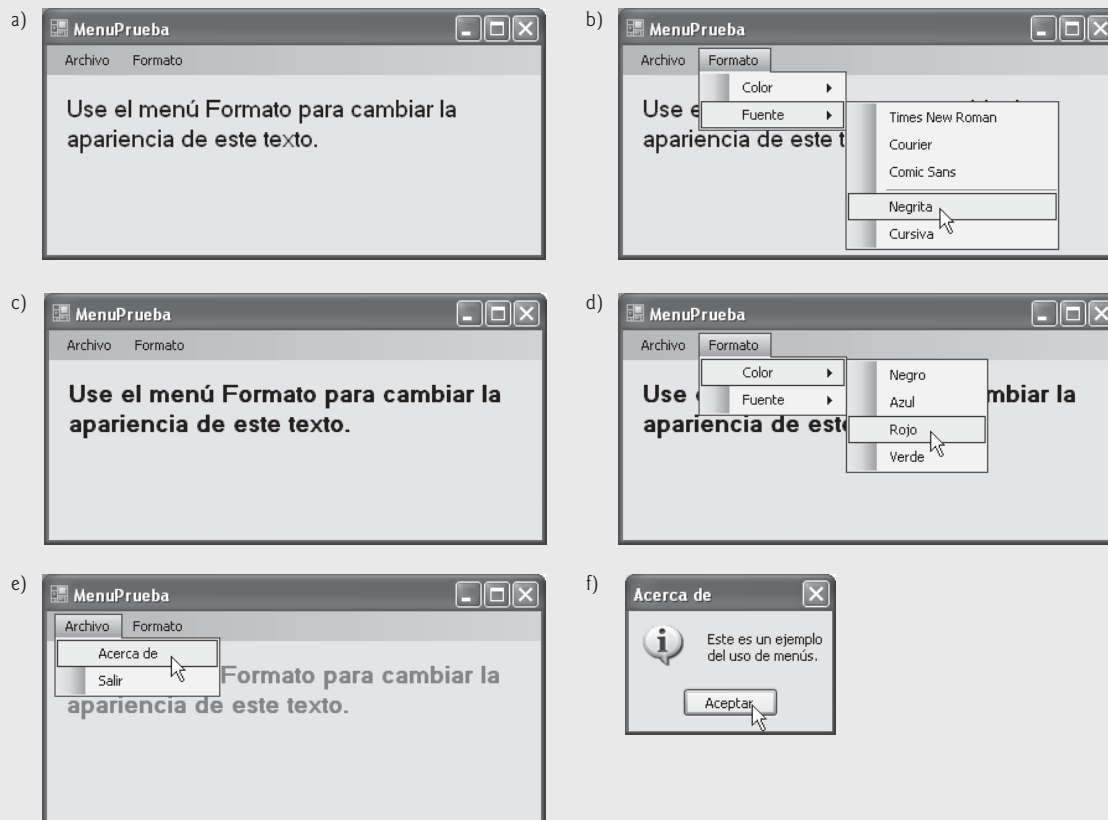


Figura 14.7 | Menús para modificar la fuente y el color del texto. (Parte 4 de 4).

Hicimos los elementos en el submenú **Color** (**Negro**, **Azul**, **Rojo** y **Verde**) mutuamente exclusivos: el usuario sólo puede seleccionar uno a la vez (en breve explicaremos cómo hicimos esto). Para indicar que un elemento de menú está seleccionado, estableceremos la propiedad **Checked** de cada elemento del menú **Color** a **true**. Esto hace que aparezca una marca de verificación a la izquierda de un elemento de menú.

Cada elemento del menú **Color** tiene su propio manejador de eventos `Click`. El manejador de eventos para el color **Negro** es `negroToolStripMenuItem_Click` (líneas 42-50). De manera similar, los manejadores de eventos para los colores **Azul**, **Rojo** y **Verde** son `azulToolStripMenuItem_Click` (líneas 53-61), `rojoToolStripMenuItem_Click` (líneas 64-72) y `verdeToolStripMenuItem_Click` (líneas 75-83), respectivamente. Cada elemento del menú **Color** debe ser mutuamente exclusivo, por lo que cada manejador de eventos llama al método `BorrarColor` (líneas 32-39) antes de establecer su correspondiente propiedad `Checked` a `true`. El método `BorrarColor` establece la propiedad `Checked` de cada `MenuItem` de color a `false`, con lo que se evita en efecto que haya más de un elemento de menú seleccionado al mismo tiempo. En el diseñador, al principio establecemos la propiedad `Checked` del elemento de menú **Negro** a `true`, ya que al principio del programa, el texto en el formulario es negro.



Observación de ingeniería de software 14.1

El control `MenuStrip` no impone la exclusión mutua de los elementos de menú, aun cuando la propiedad `Checked` sea `true`. Usted debe programar este comportamiento.

El menú **Fuente** contiene tres elementos de menú para las fuentes (**Courier**, **Times New Roman** y **Comic Sans**) y dos elementos de menú para los estilos de las fuentes (**Negrita** y **Cursiva**). Agregamos una barra separadora entre los elementos de menú de fuente y de estilo de fuente, para indicar que son opciones separadas. Un objeto `Font` sólo puede especificar una fuente a la vez, pero puede establecer varios estilos al mismo tiempo (por ejemplo, una fuente puede estar tanto en negrita como en cursiva). Establecemos los elementos del menú **Fuente** para mostrar marcas de verificación. Al igual que con el menú **Color**, debemos imponer la exclusión mutua de estos elementos en nuestros manejadores de eventos.

Los manejadores de eventos para los elementos **TimesRoman**, **Courier** y **ComicSans** del menú **Fuente** son `timesToolStripMenuItem_Click` (líneas 95-104), `courierToolStripMenuItem_Click` (líneas 107-117) y `comicToolStripMenuItem_Click` (líneas 120-129), respectivamente. Tanto éstos como manejadores de eventos del menú **Color** se comportan en forma similar. Cada manejador de eventos borra las propiedades `Checked` para todos los elementos del menú **Fuente** mediante una llamada al método `BorrarFuente` (líneas 86-92), y después establece la propiedad `Checked` del elemento de menú que generó el evento para que fuera `true`. Esto hace que se cumpla la exclusión mutua de los elementos del menú **Fuente**. En el diseñador, al principio establecemos la propiedad `Checked` del elemento de menú **Times New Roman** a `true`, ya que ésta es la fuente original para el texto en el formulario. Los manejadores de eventos para los elementos de menú **Negrita** y **Cursiva** (líneas 132-154) utilizan el operador OR exclusivo lógico a nivel de bit (`^`) para combinar los estilos de la fuente, como vimos en el capítulo 13.

14.3 Control MonthCalendar

Muchas aplicaciones deben realizar cálculos de fecha y hora. El .NET Framework cuenta con dos controles que permiten a una aplicación extraer la información de la fecha y la hora: los controles `MonthCalendar` y `DateTimePicker` (sección 14.4).

El control **MonthCalendar** (figura 14.8) muestra un calendario mensual en el formulario. El usuario puede seleccionar una fecha del mes actual que se muestra en pantalla, o puede usar los vínculos incluidos para navegar hasta otro mes. Cuando se selecciona una fecha, ésta se resalta. Para seleccionar varias fechas, haga clic en ellas en el calendario, mientras mantiene oprimida la tecla *Mayús*. El evento predeterminado para este control es **Date-Changed**, el cual se genera cuando se selecciona una nueva fecha. Se proporcionan propiedades para que usted pueda modificar la apariencia del calendario, la cantidad de fechas que pueden seleccionarse a la vez, y las fechas mínima y máxima que pueden establecerse. En la figura 14.9 se muestra un resumen de las propiedades y un evento común de `MonthCalendar`.

14.4 Control DateTimePicker

El control **DateTimePicker** (vea los resultados de la figura 14.11) es similar al control `MonthCalendar`, sólo que muestra el calendario cuando se selecciona una flecha hacia abajo. El control `DateTimePicker` puede usarse para obtener la información de la fecha y la hora del usuario. Además, este control puede personalizarse más que un



Figura 14.8 | El control MonthCalendar.

Propiedades y evento de MonthCalendar	Descripción
<i>Propiedades de MonthCalendar</i>	
FirstDayOfWeek	Establece qué día de la semana se mostrará primero, para cada semana en el calendario.
MaxDate	La última fecha que puede seleccionarse.
MaxSelectionCount	El número máximo de fechas que pueden seleccionarse a la vez.
MinDate	La primera fecha que puede seleccionarse.
MonthlyBoldedDates	Un arreglo de fechas que se mostrarán en negrita en el calendario.
SelectionEnd	La última de las fechas seleccionada por el usuario.
SelectionRange	Las fechas seleccionadas por el usuario.
SelectionStart	La primera de las fechas seleccionadas por el usuario.
<i>Evento común de MonthCalendar</i>	
ValueChanged	Se genera cuando se selecciona una fecha en el calendario.

Figura 14.9 | Propiedades y evento de MonthCalendar.

control MonthCalendar; se proporcionan más propiedades para editar la apariencia visual del calendario desplegable. La propiedad **Format** especifica las opciones de selección del usuario, mediante el uso de la enumeración **DateTimePickerFormat**. Los valores en esta enumeración son **Long** (muestra la fecha en formato largo, como en **Viernes, julio 1, 2005**), **Short** (muestra la fecha en formato corto, como en **1/7/2005**), **Time** (muestra un valor de tiempo, como en **11:48:02 PM**) y **Custom** (indica que se utilizará un formato personalizado). Si se usa el valor **Custom**, la apariencia que tendrá el control DateTimePicker se especifica mediante la propiedad **CustomFormat**. El evento predeterminado para este control es **ValueChanged**, el cual ocurre cuando se cambia el valor seleccionado (ya sea una fecha o una hora). En la figura 14.10 se muestra un resumen de las propiedades y un evento común de DateTimePicker.

La figura 14.11 demuestra el uso del control DateTimePicker para seleccionar la fecha de devolución de un artículo. Muchas compañías utilizan esta funcionalidad. Por ejemplo, varias compañías de renta de DVDs en línea especifican el día en que se envía una película, y la fecha estimada en la que llegará a su hogar. En esta aplicación, el usuario selecciona un día para la llegada, y después se muestra una fecha estimada de entrega. La fecha siempre es dos días después de la fecha de llegada, o tres días si se llega al domingo (no se entrega el correo en domingo).

El control DateTimePicker (llamado `llegadaDateTimePicker`) tiene su propiedad **Format** establecida en **Long**, por lo que el usuario puede seleccionar una fecha, y no una hora, en esta aplicación. Cuando el usuario selecciona una fecha, se produce el evento **ValueChanged**. El manejador de eventos para este evento (líneas 14-29) primero

Propiedades y evento de DateTimePicker	Descripción
<i>Propiedades de DateTimePicker</i>	
CalendarForeColor	Establece el color del texto para el calendario.
CalendarMonthBackground	Establece el color de fondo del calendario.
CustomFormat	Establece la cadena de formato personalizado para las opciones del usuario.
Format	Establece el formato de la fecha y/o la hora utilizadas para las opciones del usuario.
MaxDate	La fecha y hora máximas que pueden seleccionarse.
MinDate	La fecha y hora mínimas que pueden seleccionarse.
ShowCheckBox	Indica si debe mostrarse un control CheckBox a la izquierda de la fecha y hora mostradas.
ShowUpDown	Se utiliza para indicar que el control debe tener controles Button con flechas hacia arriba y hacia abajo. Esto es útil para las instancias en las que se utiliza el control DateTimePicker para seleccionar una hora; los controles Button pueden usarse para incrementar o decrementar los valores de hora, minuto y segundo.
Value	Los datos seleccionados por el usuario.
<i>Evento común de DateTimePicker</i>	
ValueChanged	Se genera cuando cambia la propiedad Value, incluyendo cuando el usuario selecciona una nueva fecha u hora.

Figura 14.10 | Propiedades y evento de DateTimePicker.

obtiene la fecha seleccionada de la propiedad **Value** del control DateTimePicker (línea 17). Las líneas 20-22 usan la propiedad **DayOfWeek** de la estructura DateTime para determinar el día de la semana en el cual se encuentra la fecha seleccionada. Los valores de los días se representan mediante la enumeración **DayOfWeek**. Las líneas 25 y 28 usan el método **AddDays** de DateTime para incrementar la fecha por dos o tres días, respectivamente. Después, la fecha resultante se muestra en formato Long, mediante el método **ToString**.

```

1  // Fig. 14.11: DateTimePickerForm.cs
2  // Uso de un control DateTimePicker para seleccionar una fecha de llegada.
3  using System;
4  using System.Windows.Forms;
5
6  public partial class DateTimePickerForm : Form
7  {
8      // constructor predeterminado
9      public DateTimePickerForm()
10     {
11         InitializeComponent();
12     } // fin del constructor
13
14     private void llegadaDateTimePicker_ValueChanged(
15         object sender, EventArgs e )
16     {
17         DateTime fechaLlegada = llegadaDateTimePicker.Value;
18
19         // agrega tiempo adicional cuando los artículos llegan alrededor del domingo

```

Figura 14.11 | Demostración de DateTimePicker. (Parte 1 de 2).

```

20     if ( fechaLlegada.DayOfWeek == DayOfWeek.Friday ||
21         fechaLlegada.DayOfWeek == DayOfWeek.Saturday ||
22         fechaLlegada.DayOfWeek == DayOfWeek.Sunday )
23
24         // estima tres días para la entrega
25         salidaLabel.Text = fechaLlegada.AddDays( 3 ).ToLongDateString();
26     else
27         // en caso contrario, estima sólo dos días para la entrega
28         salidaLabel.Text = fechaLlegada.AddDays( 2 ).ToLongDateString();
29 } // fin del método llegadaDateTimePicker_ValueChanged
30
31 private void DateTimePickerForm_Load( object sender, EventArgs e )
32 {
33     // el usuario no puede seleccionar días antes de hoy
34     llegadaDateTimePicker.MinDate = DateTime.Today;
35
36     // el usuario sólo puede seleccionar días de este año
37     llegadaDateTimePicker.MaxDate = DateTime.Today.AddYears( 1 );
38 } // fin del método DateTimePickerForm_Load
39 } // fin de la clase DateTimePickerForm

```

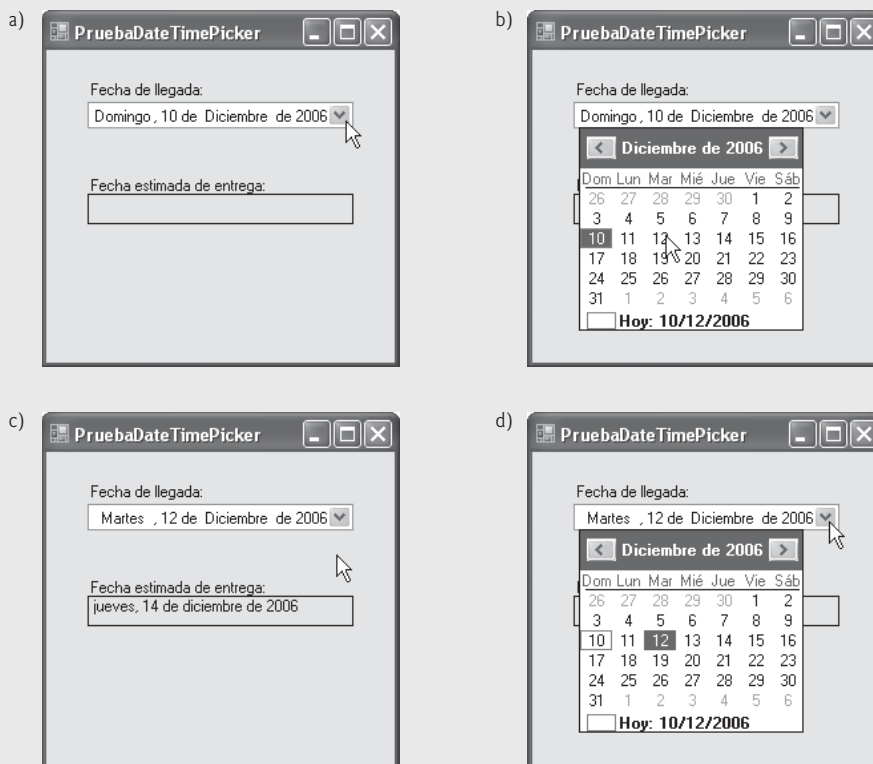


Figura 14.11 | Demostración de DateTimePicker. (Parte 2 de 2).

En esta aplicación, no queremos que el usuario pueda seleccionar un día de llegada antes del día actual, o uno que esté a más de un año de distancia en el futuro. Para asegurarnos de que esto se cumpla, establecemos las propiedades *MinDate* y *MaxDate* del control *DateTimePicker* al cargar el formulario (líneas 34 y 37). La propiedad *Today* devuelve el día actual, y el método *AddYears* (con un argumento de 1) se utiliza para especificar una fecha adelantada un año.

Analicemos los resultados con más detalle. Esta aplicación empieza por mostrar la fecha actual (figura 14.11(a)). En la figura 14.11(b), seleccionamos el 12 de diciembre. En la figura 14.11(c), la fecha estimada de llegada se muestra como el 14. La figura 14.11(d) muestra que el día 12, después de seleccionarlo, se resalta en el calendario.

14.5 Control LinkLabel

El control **LinkLabel** muestra vínculos a otros recursos, como archivos o páginas Web (figura 14.12). Un control **LinkLabel** aparece como texto subrayado (en color azul, de manera predeterminada). Cuando se desplaza el ratón sobre el vínculo, el puntero cambia a una mano; esto es similar al comportamiento de un hipervínculo en una página Web. El vínculo puede cambiar de color para indicar si es nuevo, si se visitó antes o si está activo. Al hacer clic en el control **LinkLabel**, éste genera un evento **LinkClicked** (vea la figura 14.13). La clase **LinkLabel** se deriva de la clase **Label** y, por lo tanto, hereda toda su funcionalidad.



Observación de apariencia visual 14.3

*Un control **LinkLabel** es el preferido para indicar que el usuario puede hacer clic en un vínculo, para ir a un recurso como una página Web, aunque otros controles pueden realizar tareas similares.*

La clase **PruebaLinkLabelForm** (figura 14.14) usa tres controles **LinkLabel** para crear vínculos a la unidad C:, al sitio Web de Deitel (www.deitel.com) y a la aplicación Bloc de notas, respectivamente. Las propiedades **Text** de los objetos **unidadLinkLabel**, **deitelLinkLabel** y **blocDeNotasLinkLabel** de **LinkLabel** describen el propósito de cada vínculo.

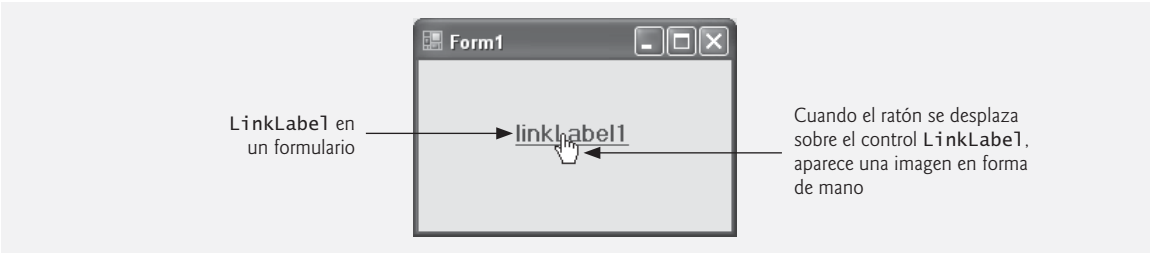


Figura 14.12 | El control **LinkLabel** en un programa en ejecución.

Propiedades y evento de LinkLabel	Descripción
<i>Propiedades comunes</i>	
ActiveLinkColor	Especifica el color del vínculo activo, cuando se hace clic en él.
LinkArea	Especifica qué parte del texto en el control LinkLabel es parte del vínculo.
LinkBehavior	Especifica el comportamiento del vínculo, como la manera en que éste aparece cuando se coloca el ratón encima de él.
LinkColor	Especifica el color original de todos los vínculos, antes de que se visiten. El color predeterminado lo establece el sistema, pero por lo general es azul.
LinkVisited	Si es true , el vínculo aparece como si se hubiera visitado (su color se cambia al color especificado por la propiedad VisitedLinkColor). El valor predeterminado es false .
Text	Especifica el texto del control.

Figura 14.13 | Propiedades y evento de **LinkLabel**. (Parte 1 de 2).

Propiedades y evento de LinkLabel	Descripción
UseMnemonic	Si es true, el carácter & en la propiedad Text actúa como acceso abreviado (de manera similar al acceso abreviado con <i>Alt</i> en los menús).
VisitedLinkColor	Especifica el color de los vínculos visitados. El color predeterminado lo establece el sistema, pero por lo general es púrpura.
<i>Evento común</i>	<i>(Argumentos de evento LinkLabelLinkClickedEventArgs)</i>
LinkClicked	Se genera cuando se hace clic en el vínculo. Éste es el evento predeterminado cuando se hace doble clic en el control, en modo de Diseño .

Figura 14.13 | Propiedades y evento de LinkLabel. (Parte 2 de 2).

```

1  // Fig. 14.14: PruebaLinkLabelForm.cs
2  // Uso de controles LinkLabel para crear hipervínculos.
3  using System;
4  using System.Windows.Forms;
5
6  // el formulario usa controles LinkLabel para explorar la unidad C:\,
7  // cargar una página Web y ejecutar el Bloc de notas
8  public partial class PruebaLinkLabelForm : Form
9  {
10     // constructor predeterminado
11     public PruebaLinkLabelForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // explora la unidad C:\
17     private void unidadLinkLabel_LinkClicked( object sender,
18         LinkLabelLinkClickedEventArgs e )
19     {
20         // cambia la propiedad LinkColor después de hacer clic en el vínculo
21         unidadLinkLabel.LinkVisited = true;
22
23         System.Diagnostics.Process.Start( @"C:\" );
24     } // fin del método unidadLinkLabel_LinkClicked
25
26     // carga www.deitel.com en el explorador Web
27     private void deitelLinkLabel_LinkClicked( object sender,
28         LinkLabelLinkClickedEventArgs e )
29     {
30         // cambia la propiedad LinkColor después de hacer clic en el vínculo
31         deitelLinkLabel.LinkVisited = true;
32
33         System.Diagnostics.Process.Start(
34             "IExplore", "http://www.deitel.com" );
35     } // fin del método deitelLinkLabel_LinkClicked
36
37     // ejecuta la aplicación Bloc de notas
38     private void blocDeNotasLinkLabel_LinkClicked( object sender,
39         LinkLabelLinkClickedEventArgs e )
40     {

```

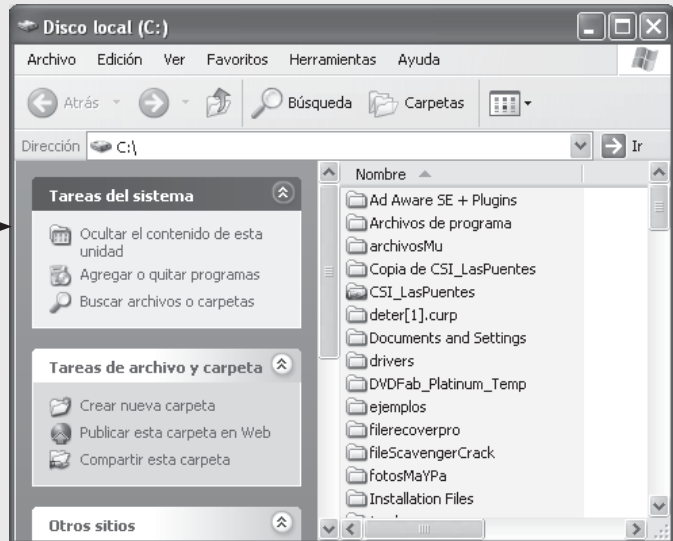
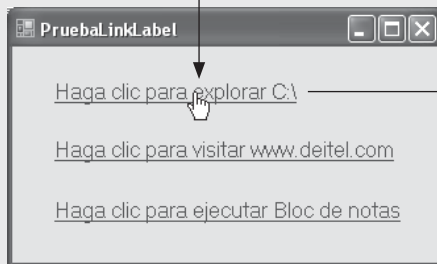
Figura 14.14 | Uso de controles LinkLabel para vincular a una unidad de disco, una página Web y una aplicación. (Parte 1 de 3).

```

41 // cambia la propiedad LinkColor después de hacer clic en el vínculo
42 blocDeNotasLinkLabel.LinkVisited = true;
43
44 // se llama al programa como si estuviera en el menú
45 // ejecutar, por lo que no se requiere la ruta completa
46 System.Diagnostics.Process.Start( "notepad" );
47 } // fin del método unidadLinkLabel_LinkClicked
48 } // fin de la clase PruebaLinkLabelForm

```

Haga clic en el primer control LinkLabel para ver el contenido de la unidad C:



Haga clic en el segundo control LinkLabel para ir al sitio Web de Deitel

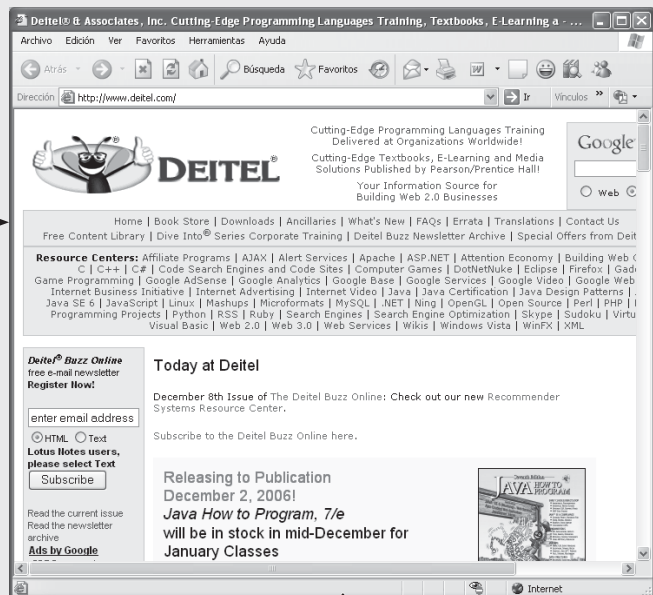
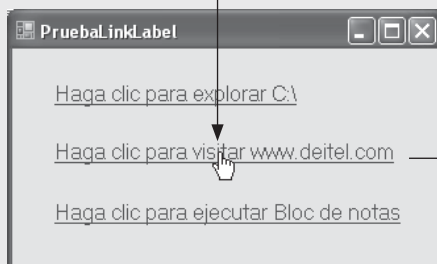


Figura 14.14 | Uso de controles LinkLabel para vincular a una unidad de disco, una página Web y una aplicación. (Parte 2 de 3).



Figura 14.14 | Uso de controles LinkLabel para vincular a una unidad de disco, una página Web y una aplicación. (Parte 3 de 3).

Los manejadores de eventos para los controles LinkLabel llaman al método **Start** de la clase **Process** (espacio de nombres **System.Diagnostics**), el cual le permite ejecutar otros programas desde una aplicación. El método Start puede recibir un argumento, el archivo a abrir (un objeto string), o dos argumentos, la aplicación a ejecutar y sus argumentos de línea de comandos (dos objetos string). Los argumentos del método Start pueden tener la misma forma que como si se utilizaran desde el comando **Ejecutar** de Windows (**Inicio > Ejecutar...**). Para las aplicaciones que conoce Windows no se requieren los nombres de ruta completos y, por lo general puede omitirse la extensión **.exe**. Para abrir un archivo cuyo tipo reconozca Windows, sólo necesita usar el nombre de ruta completo del archivo. El sistema operativo Windows debe poder utilizar la aplicación asociada con la extensión del archivo dado para abrirlo.

El manejador de eventos para el evento LinkClicked de unidadLinkLabel explora la unidad C: (líneas 17-24). La línea 21 establece la propiedad LinkVisited a true, la cual modifica el color del vínculo, de azul a púrpura (los colores de LinkVisited se pueden configurar a través de la ventana **Propiedades** en Visual Studio). Después, el manejador de eventos pasa @"C:\" al método Start (línea 23), el cual abre una ventana del **Explorador de Windows**. El símbolo @ que colocamos antes de "C:\" indica que todos los caracteres en el objeto string deben interpretarse en forma literal. Por ende, la barra diagonal inversa dentro del objeto string no se considera como el primer carácter de una secuencia de escape. Esto simplifica los objetos string que representan rutas de directorios, ya que no necesitamos usar \\ por cada carácter \ en la ruta.

El manejador de eventos para el evento LinkClicked de deitelLinkLabel (líneas 27-35) abre la página Web **www.deitel.com** en Internet Explorer. Para lograr esto, pasamos la dirección de la página Web como un objeto string (líneas 33-34), con lo cual se abre Internet Explorer. La línea 31 establece la propiedad LinkVisited a true.

El manejador de eventos para el evento LinkClicked de blocDeNotasLinkLabel (líneas 38-47) abre la aplicación **Bloc de notas**. La línea 42 establece la propiedad LinkVisited a true, de manera que el vínculo aparezca como visitado. La línea 46 pasa el argumento "notepad" al método Start, que ejecuta notepad.exe. Observe que en la línea 46 no se requiere la extensión **.exe**; Windows puede determinar si reconoce el argumento que se proporciona al método Start como un archivo ejecutable.

14.6 Control ListBox

El control **ListBox** permite al usuario ver y seleccionar varios elementos en una lista. Estos controles son entidades estáticas de la GUI, lo cual significa que deben agregarse elementos a la lista mediante la programación. Se pueden proporcionar al usuario controles **TextBox** y **Button** con los cuales pueda especificar los elementos que desea agregar a la lista, pero el verdadero proceso de agregarlos debe realizarse en el código. El control **CheckedListBox** (sección 14.7) extiende a un control **ListBox** mediante la inclusión de controles **CheckBox** a un lado de cada elemento de la lista. Esto permite a los usuarios colocar marcas de verificación en varios elementos a la vez, como se hace con los controles **CheckBox**. (Los usuarios también pueden seleccionar varios elementos de un control **ListBox** mediante la propiedad **SelectionMode**, que veremos en breve). La figura 14.15 muestra un control **ListBox** y un control **CheckedListBox**. En ambos controles aparecen barras de desplazamiento si el número de elementos es mayor que el área visible del control **ListBox**.

La figura 14.16 lista las propiedades y métodos comunes de **ListBox**, y un evento común. La propiedad **SelectionMode** determina el número de elementos que pueden seleccionarse. Esta propiedad tiene los posibles valores **None**, **One**, **MultiSimple** y **MultiExtended** (de la enumeración **SelectionMode**); en la figura 14.16 explicamos las diferencias entre estas opciones. El evento **SelectedIndexChanged** ocurre cuando el usuario selecciona un nuevo elemento.

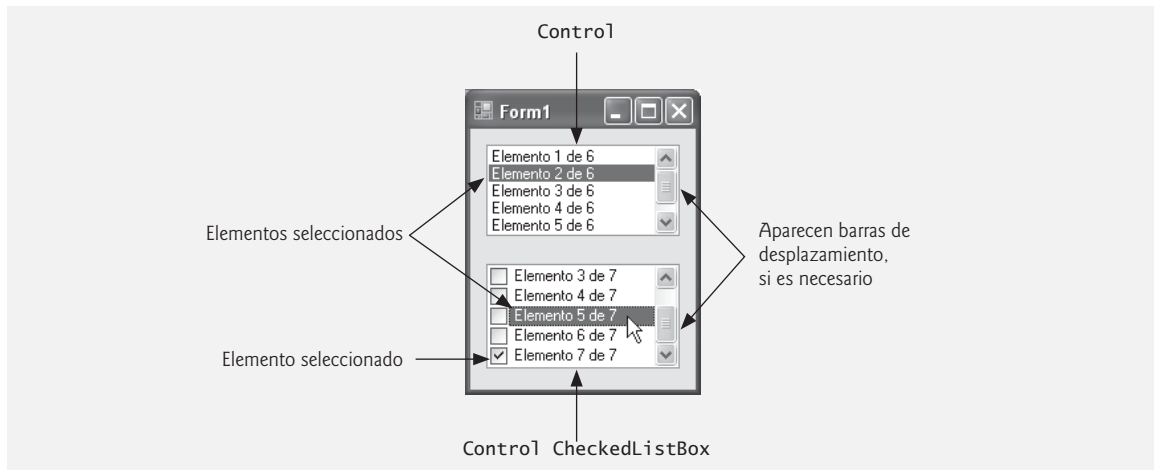


Figura 14.15 | Controles **ListBox** y **CheckedListBox** en un formulario.

Propiedades, métodos y evento de ListBox	Descripción
<i>Propiedades comunes</i>	
Items	La colección de elementos en el control ListBox .
MultiColumn	Indica si el control ListBox puede dividir una lista en varias columnas, con lo cual se eliminan las barras de desplazamiento verticales de la pantalla.
SelectedIndex	Devuelve el índice del elemento seleccionado. Si no se han seleccionado elementos, la propiedad devuelve -1. Si el usuario selecciona varios elementos, esta propiedad devuelve sólo uno de los índices seleccionados. Por esta razón, si desea seleccionar varios elementos, debe usar la propiedad SelectedIndices .
SelectedIndices	Devuelve una colección que contiene los índices para todos los elementos seleccionados.

Figura 14.16 | Propiedades, métodos y evento de **ListBox**. (Parte 1 de 2).

Propiedades, métodos y evento de <code>ListBox</code>	Descripción
<code>SelectedItem</code>	Devuelve una referencia al elemento seleccionado. Si se seleccionan varios elementos, devuelve el elemento con el número de índice más bajo.
<code>SelectedItems</code>	Devuelve una colección del (los) elemento(s) seleccionado(s).
<code>SelectionMode</code>	Determina el número de elementos que pueden seleccionarse, y el medio a través del cual pueden seleccionarse varios elementos. Los valores son <code>None</code> , <code>One</code> , <code>MultiSimple</code> (se permite la selección múltiple) o <code>MultiExtended</code> (se permite la selección múltiple mediante el uso de una combinación de teclas de flechas, o los clics del ratón y las teclas <i>Mayús</i> y <i>Ctrl</i>).
<code>Sorted</code>	Indica si los elementos están ordenados en forma alfabética. Al establecer el valor de esta propiedad a <code>true</code> se ordenan los elementos. El valor predeterminado es <code>false</code> .
<i>Evento común</i>	
<code>ClearSelected</code>	Deselecciona todos los elementos.
<code>GetSelected</code>	Recibe un índice como argumento y devuelve <code>true</code> si se selecciona el elemento correspondiente.
<i>Evento común</i>	
<code>SelectedIndexChanged</code>	Se genera cuando cambia el índice seleccionado. Éste es el evento predeterminado cuando se hace doble clic en el control, estando en el diseñador.

Figura 14.16 | Propiedades, métodos y evento de `ListBox`. (Parte 2 de 2).

Tanto el control `ListBox` como `CheckedListBox` tienen las propiedades `Items`, `SelectedItem` y `SelectedIndex`. La propiedad **Items** devuelve todos los elementos de la lista, en forma de una colección. Las colecciones son una manera común de administrar listas de objetos en el .NET Framework. Muchos componentes .NET de la GUI (por ejemplo, controles `ListBox`) usan colecciones para exponer listas de objetos internos (por ejemplo, los elementos contenidos dentro de un control `ListBox`). En el capítulo 26 hablaremos más acerca de las colecciones. La colección que devuelve la propiedad `Items` se representa como un objeto de tipo **ObjectCollection**. La propiedad **SelectedItem** devuelve el elemento actual seleccionado de `ListBox`. Si el usuario puede seleccionar varios elementos, use la colección **SelectedItems** para devolver todos los elementos seleccionados como una colección. La propiedad **SelectedIndex** devuelve el índice del elemento seleccionado; si pudiera haber más de uno, use la propiedad **SelectedIndices**. Si no hay elementos seleccionados, la propiedad `SelectedIndex` devuelve `-1`. El método **GetSelected** recibe un índice y devuelve `true` si está seleccionado el elemento correspondiente.

Para agregar elementos a un control `ListBox` o `CheckedListBox`, debemos agregar objetos a su colección `Items`. Para lograr esto podemos llamar al método `Add`, para agregar un objeto `string` a la colección `Items` de `ListBox` o `CheckedListBox`. Por ejemplo, podríamos escribir

```
miListBox.Items.Add( miListItem )
```

para agregar el objeto `string` *miListItem* al control `ListBox` llamado *miListBox*. Para agregar varios objetos, puede llamar al método `Add` varias veces, o al método `AddRange` para agregar un arreglo de objetos. Las clases `ListBox` y `CheckedListBox` pueden llamar al método `ToString` del objeto enviado para determinar el control `Label` para la entrada del objeto correspondiente en la lista. Esto le permite agregar distintos objetos a un control `ListBox` o `CheckedListBox`, los cuales pueden devolverse después a través de las propiedades `SelectedItem` y `SelectedItems`.

De manera alternativa, puede agregar elementos a controles `ListBox` y `CheckedListBox` en forma visual, examinando la propiedad `Items` en la ventana **Propiedades**. Al hacer clic en el botón de elipsis se abre el **Editor de la colección Cadena**, el cual contiene un área de texto para agregar elementos; cada elemento aparece en una línea separada (figura 14.17). Después, Visual Studio escribe el código para agregar estos objetos `string` a la colección `Items` dentro del método `InitializeComponent`.

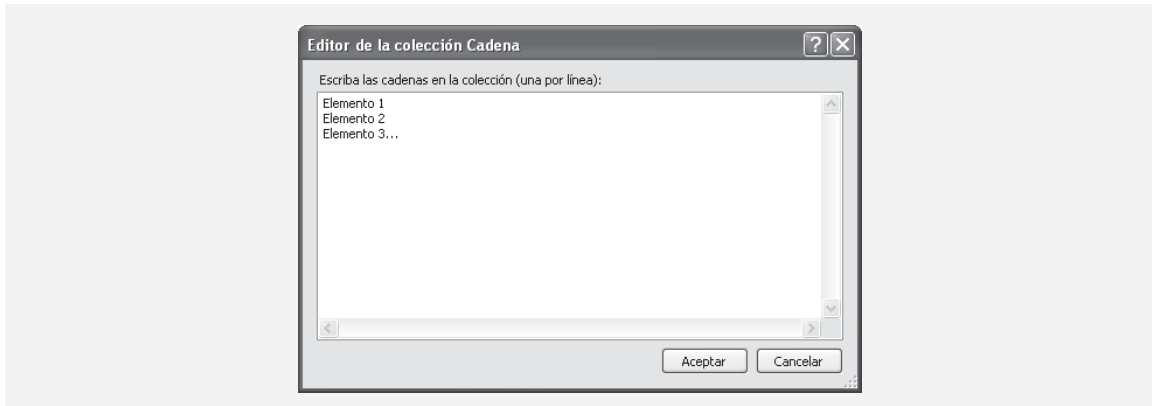


Figura 14.17 | Editor de la colección Cadena.

La figura 14.18 utiliza la clase `PruebaListBoxForm` para agregar, quitar y borrar elementos del control `ListBox` llamado `mostrarListBox`. La clase `PruebaListBoxForm` utiliza el control `TextBox` `entradaTextBox` para permitir al usuario introducir un nuevo elemento. Cuando el usuario hace clic en el botón **Add**, el nuevo elemento aparece en `mostrarListBox`. De manera similar, si el usuario selecciona un elemento y hace clic en **Quitar**, el elemento se elimina. Al hacer clic en **Borrar**, se eliminan todas las entradas en `mostrarListBox`. Para terminar la aplicación, el usuario hace clic en **Salir**.

El manejador de eventos `agregarButton_Click` (líneas 18-22) llama al método **Add** de la colección `Items` en el control `ListBox`. Este método recibe un objeto `string` como el elemento que se agregará a `mostrarListBox`. En este caso, el objeto `string` utilizado es el texto introducido por el usuario, o `entradaTextBox.Text` (línea 20). Después de agregar el elemento, se borra el contenido de `entradaTextBox.Text` (línea 21).

El manejador de eventos `quitarButton_Click` (líneas 25-30) utiliza el método **RemoveAt** para quitar un elemento del control `ListBox`. El manejador de eventos `quitarButton_Click` primero utiliza la propiedad `SelectedIndex` para determinar cuál índice está seleccionado. Si `SelectedIndex` no es -1 (es decir, que haya un elemento seleccionado), la línea 29 quita el elemento que corresponde al índice seleccionado.

El manejador de eventos `borrarButton_Click` (líneas 33-36) llama al método **Clear** de la colección `Items` (línea 35). Esto elimina todas las entradas en `mostrarListBox`. Por último, el manejador de eventos `salirButton_Click` (líneas 39-42) termina la aplicación llamando al método `Application.Exit` (línea 41).

```

1  // Fig. 14.18: PruebaListBoxForm.cs
2  // Programa para agregar, quitar y borrar elementos de un control ListBox
3  using System;
4  using System.Windows.Forms;
5
6  // el formulario utiliza un control TextBox y controles Button para
7  // agregar, quitar y borrar elementos de un control ListBox
8  public partial class PruebaListBoxForm : Form
9  {
10     // constructor predeterminado
11     public PruebaListBoxForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // agrega nuevo elemento al control ListBox (texto de entradaTextBox)
17     // y borra el control TextBox de entrada

```

Figura 14.18 | Programa que agrega, quita y borra elementos de un control `ListBox`. (Parte I de 2).

```

18 private void agregarButton_Click( object sender, EventArgs e )
19 {
20     mostrarListBox.Items.Add( entradaTextBox.Text );
21     entradaTextBox.Clear();
22 } // fin del método agregarButton_Click
23
24 // quita un elemento, si está seleccionado
25 private void quitarButton_Click( object sender, EventArgs e )
26 {
27     // comprueba si el elemento está seleccionado, si es así lo quita
28     if ( mostrarListBox.SelectedIndex != -1 )
29         mostrarListBox.Items.RemoveAt( mostrarListBox.SelectedIndex );
30 } // fin del método quitarButton_Click
31
32 // borra todos los elementos del control ListBox
33 private void borrarButton_Click( object sender, EventArgs e )
34 {
35     mostrarListBox.Items.Clear();
36 } // fin del método borrarButton_Click
37
38 // sale de la aplicación
39 private void salirButton_Click( object sender, EventArgs e )
40 {
41     Application.Exit();
42 } // fin del método salirButton_Click
43 } // fin de la clase PruebaListBoxForm

```

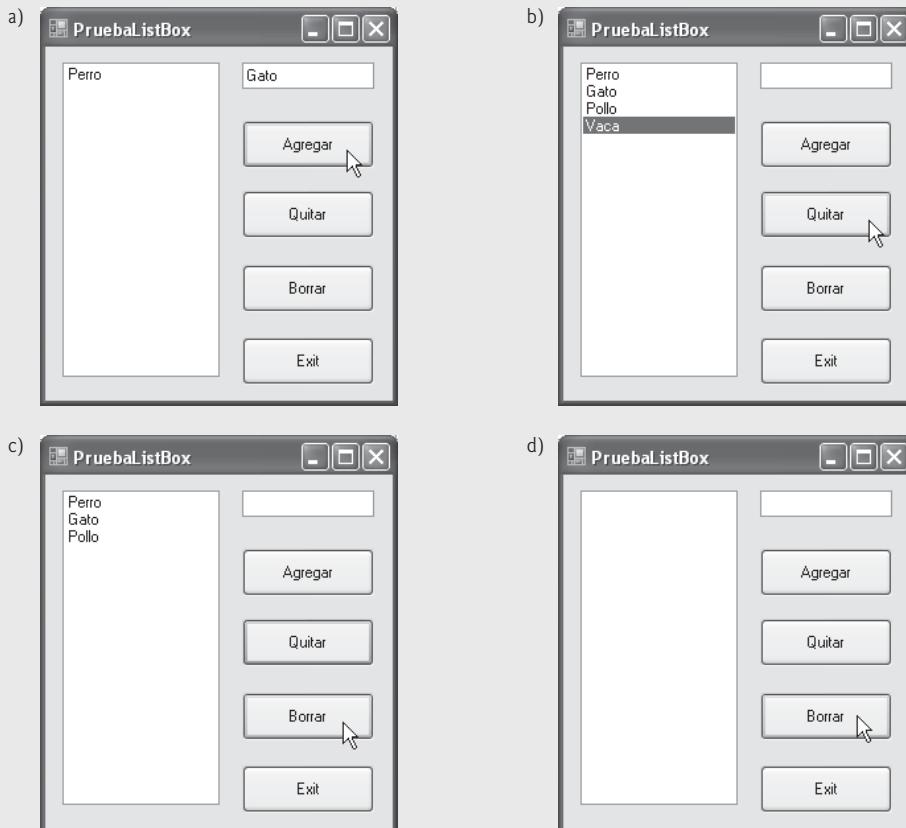


Figura 14.18 | Programa que agrega, quita y borra elementos de un control `ListBox`. (Parte 2 de 2).

14.7 Control `CheckedListBox`

El control `CheckedListBox` se deriva de la clase `ListBox`, e incluye un control `CheckBox` enseguida de cada elemento. Al igual que en los controles `ListBox`, pueden agregarse elementos a través de los métodos `Add` y `AddRange`, o a través del **Editor de la colección Cadena**. En los controles `CheckedListBox` se pueden seleccionar varios elementos, y los únicos valores posibles para la propiedad `SelectionMode` son `None` y `One`. `One` permite la selección múltiple, ya que en los controles `CheckBox` no hay restricciones lógicas sobre los elementos; el usuario puede seleccionar todos los que requiera. Por ende, la única opción es si se va a permitir que el usuario realice selecciones múltiples o no. Esto mantiene el comportamiento del control `CheckedListBox` consistente con el de los controles `CheckBox`. En la figura 14.19 aparecen las propiedades y eventos comunes, y un método común de los controles `CheckBox`.



Error común de programación 14.1

El IDE muestra un mensaje de error si usted trata de establecer la propiedad `SelectionMode` a `MultiSimple` o `MultiExtended` en la ventana **Propiedades** de un control `CheckedListBox`. Si este valor se establece mediante la programación, se produce un error en tiempo de ejecución.

El evento **`ItemCheck`** ocurre cada vez que un usuario selecciona o deselecciona un elemento `CheckedListBox`. Las propiedades `CurrentValue` y `NewValue` del argumento del evento devuelven valores `CheckState` para el estado actual y nuevo del elemento, respectivamente. Una comparación de estos valores le permite determinar si el elemento `CheckedListBox` estaba seleccionado o deseleccionado. El control `CheckedListBox` retiene las propiedades `SelectedItems` y `SelectedIndices` (las hereda de la clase `ListBox`). No obstante, también incluye las propiedades `CheckedItems` y `CheckedIndices`, las cuales devuelven información acerca de los elementos e índices seleccionados.

Propiedades, métodos y evento de <code>CheckedListBox</code>	Descripción
<i>Propiedades comunes</i>	<i>(Todas las propiedades, métodos y eventos de <code>ListBox</code> las hereda el control <code>CheckedListBox</code>.)</i>
<code>CheckedItems</code>	Contiene la colección de elementos que están seleccionados. Esto es distinto del elemento seleccionado, que está resaltado (pero no necesariamente activado). [Nota: puede haber cuando mucho un elemento seleccionado en un momento dado.]
<code>CheckedIndices</code>	Devuelve los índices de todos los elementos seleccionados.
<code>SelectionMode</code>	Determina cuántos elementos pueden seleccionarse. Los únicos valores posibles son <code>One</code> (permite seleccionar varios elementos) o <code>None</code> (no permite seleccionar ningún elemento).
<i>Método común</i>	
<code>GetItemChecked</code>	Recibe un índice y devuelve <code>true</code> si el elemento correspondiente está seleccionado.
<i>Evento común (argumentos de evento <code>ItemCheckEventArgs</code>)</i>	
<code>ItemCheck</code>	Se genera cuando se selecciona o deselecciona un elemento.
<i>Propiedades de <code>ItemCheckEventArgs</code></i>	
<code>CurrentValue</code>	Indica si el elemento actual está seleccionado o deseleccionado. Los posibles valores son <code>Checked</code> , <code>Unchecked</code> e <code>Indeterminate</code> .
<code>ItemCheck</code>	Devuelve el índice con base cero del elemento que cambió.
<code>NewValue</code>	Especifica el nuevo estado del elemento.

Figura 14.19 | Propiedades, método y evento de `CheckedListBox`.

En la figura 14.20, la clase `PruebaCheckedListBoxForm` usa un control `CheckedListBox` y un control `ListBox` para mostrar la selección de libros de un usuario. El control `CheckedListBox` permite al usuario seleccionar varios títulos. En el **Editor de la colección Cadena** se agregaron elementos para algunos libros de Deitel: C++, Java™, Visual Basic, Internet y WWW, Perl, Python, Internet inalámbrica y Java avanzado (el acrónimo HTP significa “Cómo programar”). El control `ListBox` (llamado `mostrarListBox`) muestra la selección del usuario. En las capturas de pantalla que acompañan este ejemplo, el control `CheckedListBox` aparece a la izquierda y el control `ListBox` a la derecha.

Cuando el usuario selecciona o deselecciona un elemento en `entradaCheckedListBox`, se produce un evento `ItemCheck` y se ejecuta el manejador de eventos `entradaCheckedListBox_ItemCheck` (líneas 17-29). Una instrucción `if...else` (líneas 25-28) determina si el usuario seleccionó o deseleccionó un elemento en el control `CheckedListBox`. La línea 25 utiliza la propiedad `NewValue` para determinar si el elemento se está seleccionando (`CheckState.Checked`). Si el usuario selecciona un elemento, la línea 26 agrega una entrada seleccionada al control `ListBox` `mostrarListBox`. Si el usuario deselecciona un elemento, la línea 28 quita el elemento correspondiente de `mostrarListBox`. Este manejador de eventos se creó seleccionando el control `CheckedListBox` en modo de **Diseño**, viendo los eventos del control en la ventana **Propiedades** y haciendo doble clic en el evento `ItemCheck`.

14.8 Control ComboBox

El control **ComboBox** combina las características del control `TextBox` con una *lista desplegable*: un componente de la GUI que contiene una lista, de la cual se puede seleccionar un valor. Por lo general, un control `ComboBox`

```

1  // Fig. 14.20: PruebaCheckedListBoxForm.cs
2  // Uso del control CheckedListBox para agregar elementos al control mostrarListBox
3  using System;
4  using System.Windows.Forms;
5
6  // el formulario usa un control CheckedListBox para agregar elementos al control
   mostrarListBox
7  public partial class PruebaCheckedListBoxForm : Form
8  {
9      // constructor predeterminado
10     public PruebaCheckedListBoxForm()
11     {
12         InitializeComponent();
13     } // fin del constructor
14
15     // elemento a punto de cambiar
16     // lo agrega o lo quita de mostrarListBox
17     private void entradaCheckedListBox_ItemCheck(
18         object sender, ItemCheckEventArgs e )
19     {
20         // obtiene referencia del elemento seleccionado
21         string elemento = entradaCheckedListBox.SelectedItem.ToString();
22
23         // si el elemento está seleccionado, se agrega al control ListBox
24         // en caso contrario, se quita del control ListBox
25         if ( e.NewValue == CheckState.Checked )
26             mostrarListBox.Items.Add( elemento );
27         else
28             mostrarListBox.Items.Remove( elemento );
29     } // fin del método entradaCheckedListBox_ItemCheck
30 } // fin de la clase CheckedListBoxTestForm

```

Figura 14.20 | Uso de los controles `CheckedListBox` y `ListBox` en un programa para mostrar la selección de un usuario. (Parte 1 de 2).

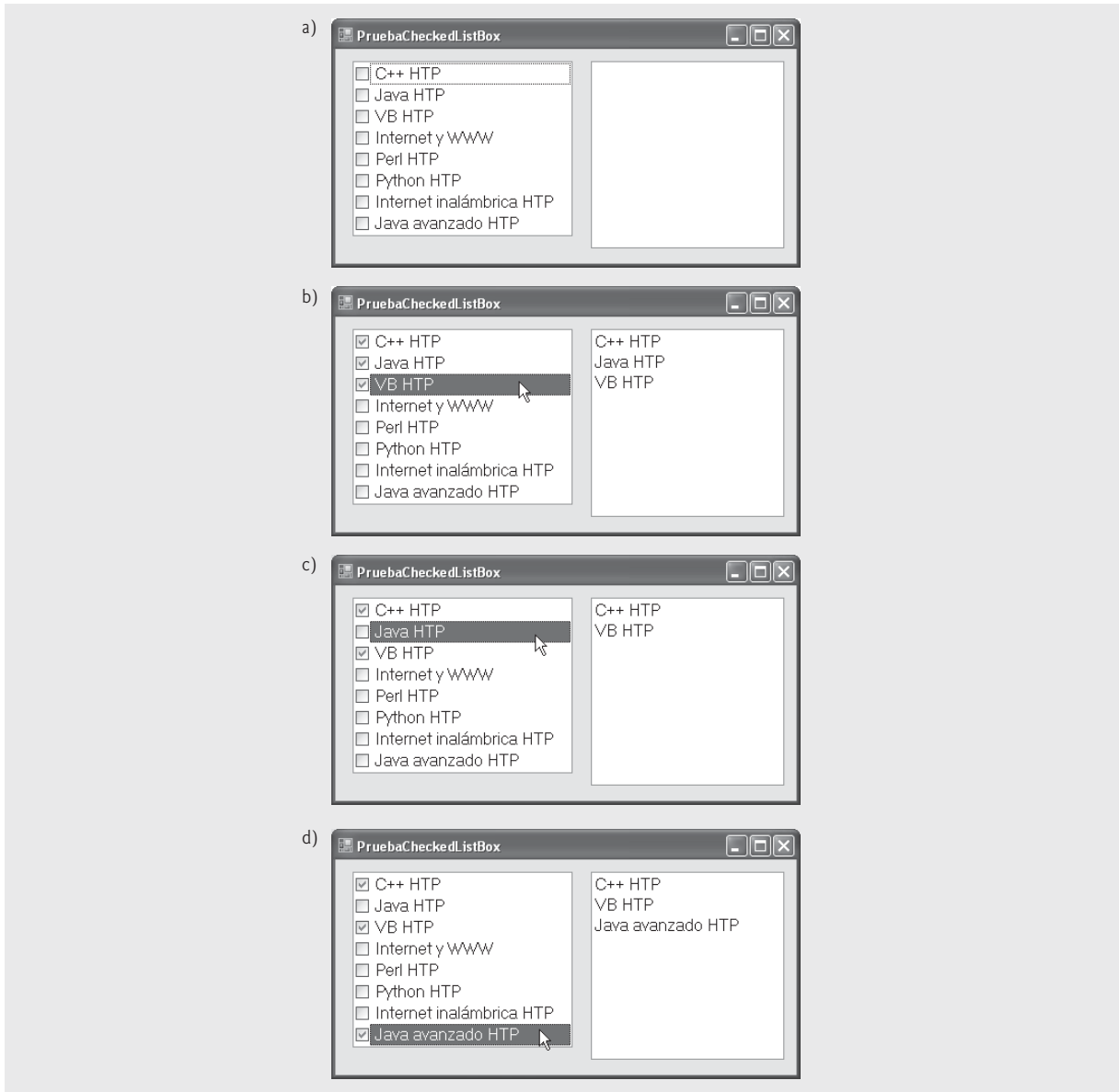


Figura 14.20 | Uso de los controles `CheckedListBox` y `ListBox` en un programa para mostrar la selección de un usuario. (Parte 2 de 2).

aparece como un control `TextBox` con una flecha hacia abajo de lado derecho. De manera predeterminada, el usuario puede introducir texto en el control `TextBox` o puede hacer clic en la flecha hacia abajo para mostrar una lista de elementos predefinidos. Si un usuario selecciona un elemento de la lista, ese elemento se muestra en el control `TextBox`. Si la lista contiene más elementos de los que se pueden mostrar en la lista desplegable, aparece una barra de desplazamiento. El número máximo de elementos que puede mostrar una lista desplegable a la vez se establece mediante la propiedad **`MaxDropDownItems`**. La figura 14.21 muestra un control `ComboBox` de ejemplo en tres estados distintos.

Al igual que con el control `ListBox`, puede agregar objetos a la colección `Items` mediante programación, usando los métodos `Add` y `AddRange`, o en forma visual con el **Editor de la colección Cadena**. La figura 14.22 lista propiedades comunes y un evento común de la clase `ComboBox`.

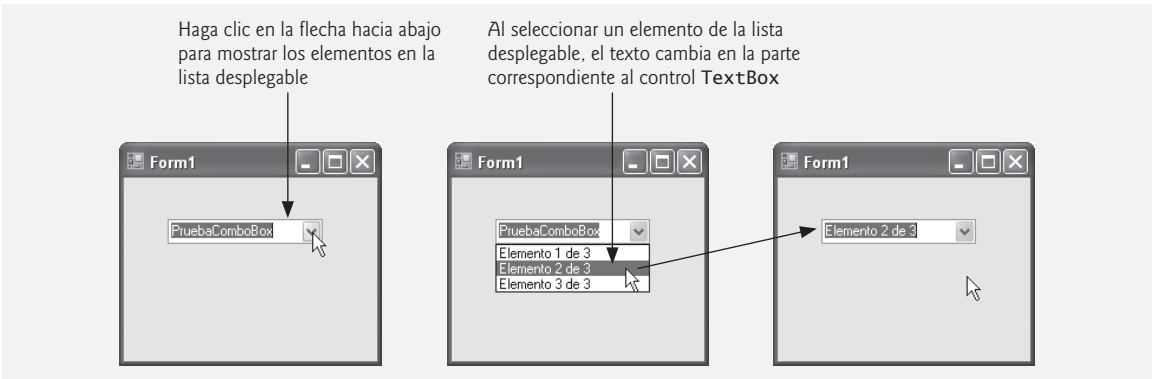


Figura 14.21 | Demostración del control ComboBox.



Observación de apariencia visual 14.4

Use un control `ComboBox` para ahorrar espacio en una GUI. Una desventaja de ello es que, a diferencia de un control `ListBox`, el usuario no puede ver los elementos disponibles sin expandir la lista desplegable.

La propiedad **`DropDownStyle`** determina el tipo de control `ComboBox`, y se representa como un valor de la enumeración **`ComboBoxStyle`**, que contiene los valores `Simple`, `DropDown` y `DropDownList`. La opción **`Simple`** no muestra una flecha hacia abajo, sino que aparece una barra de desplazamiento a un lado del control, lo que permite al usuario seleccionar una opción de la lista. El usuario también puede introducir una selección desde el teclado. El estilo **`DropDown`** (predeterminado) muestra una lista desplegable cuando se hace clic en la flecha hacia

Propiedades y evento de ComboBox	Descripción
<i>Propiedades comunes</i>	
<code>DropDownStyle</code>	Determina el tipo de control <code>ComboBox</code> . El valor <code>Simple</code> significa que la porción del texto puede editarse, y la porción de la lista siempre está visible. El valor <code>DropDown</code> (predeterminado) significa que la porción del texto es editable, pero el usuario debe hacer clic en el botón de flecha para ver la porción de lista. El valor <code>DropDownList</code> significa que la porción de texto no puede editarse, y el usuario debe hacer clic en el botón de flecha hacia abajo para ver la porción de la lista.
<code>Items</code>	La colección de elementos en el control <code>ComboBox</code> .
<code>MaxDropDownItems</code>	Especifica el máximo número de elementos (entre 1 y 100) que puede mostrar la lista desplegable. Si el número de elementos excede al número máximo de elementos a mostrar, aparece una barra de desplazamiento.
<code>SelectedIndex</code>	Devuelve el índice del elemento seleccionado. Si no hay un elemento seleccionado, se devuelve -1.
<code>SelectedItem</code>	Devuelve una referencia al elemento seleccionado.
<code>Sorted</code>	Indica si los elementos están ordenados alfabéticamente. Al establecer el valor de esta propiedad a <code>true</code> se ordenan los elementos. El valor predeterminado es <code>false</code> .
<i>Evento común</i>	
<code>SelectedIndexChanged</code>	Se genera cuando cambia el índice seleccionado (como cuando se selecciona un elemento distinto). Es el evento predeterminado cuando se hace doble clic en el control, estando en el diseñador.

Figura 14.22 | Propiedades y evento de ComboBox.

abajo (o cuando se oprime la tecla fecha hacia abajo). El usuario puede escribir un nuevo elemento en el control ComboBox. El último estilo es **DropDownList**, el cual muestra una lista desplegable pero no permite al usuario escribir en el control TextBox.

El control ComboBox cuenta con las propiedades **Items** (una colección), **SelectedItem** y **SelectedIndex**, las cuales son similares a las propiedades correspondientes en ListBox. Puede haber cuando menos un elemento seleccionado en un ComboBox. Si no hay elementos seleccionados, entonces **SelectedIndex** es -1. Cuando cambia el elemento seleccionado, se produce un evento **SelectedIndexChanged**.

La clase PruebaComboBoxForm (figura 14.23) permite a los usuarios seleccionar una figura para dibujar: círculo, elipse, cuadrado o pastel (con relleno o sin relleno), mediante el uso de un control ComboBox. En este ejemplo, el control ComboBox no puede editarse, por lo que el usuario no puede escribir en el control TextBox.

```

1  // Fig. 14.23: PruebaComboBoxForm.cs
2  // Uso de un control ComboBox para seleccionar una figura a dibujar.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // el formulario usa un control ComboBox para seleccionar distintas figuras a dibujar
8  public partial class PruebaComboBoxForm : Form
9  {
10     // constructor predeterminado
11     public PruebaComboBoxForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // obtiene el índice de la figura seleccionada, dibuja la figura
17     private void imagenComboBox_SelectedIndexChanged(
18         object sender, EventArgs e )
19     {
20         // crea objetos Graphics, Pen y SolidBrush
21         Graphics miGraphics = base.CreateGraphics();
22
23         // crea objeto Pen usando el color DarkRed (rojo oscuro)
24         Pen miPen = new Pen( Color.DarkRed );
25
26         // crea objeto SolidBrush, usando el color rojo oscuro
27         SolidBrush miSolidBrush = new SolidBrush( Color.DarkRed );
28
29         // borra el área de dibujo y la establece de color blanco
30         miGraphics.Clear( Color.White );
31
32         // busca índice, dibuja la figura apropiada
33         switch ( imagenComboBox.SelectedIndex )
34         {
35             case 0: // se selecciona el caso del Círculo
36                 miGraphics.DrawEllipse( miPen, 50, 50, 150, 150 );
37                 break;
38             case 1: // se selecciona el caso del Rectángulo
39                 miGraphics.DrawRectangle( miPen, 50, 50, 150, 150 );
40                 break;
41             case 2: // se selecciona el caso de la Elipse
42                 miGraphics.DrawEllipse( miPen, 50, 85, 150, 115 );
43                 break;
44             case 3: // se selecciona el caso del Pastel

```

Figura 14.23 | Uso de un control ComboBox para dibujar una figura seleccionada. (Parte I de 2).


```

45         miGraphics.DrawPie( miPen, 50, 50, 150, 150, 0, 45 );
46         break;
47     case 4: // se selecciona el caso del Círculo relleno
48         miGraphics.FillEllipse( miSolidBrush, 50, 50, 150, 150 );
49         break;
50     case 5: // se selecciona el caso del Rectángulo relleno
51         miGraphics.FillRectangle( miSolidBrush, 50, 50, 150, 150 );
52         break;
53     case 6: // se selecciona el caso del Elipse relleno
54         miGraphics.FillEllipse( miSolidBrush, 50, 85, 150, 115 );
55         break;
56     case 7: // se selecciona el caso del Pastel relleno
57         miGraphics.FillPie( miSolidBrush, 50, 50, 150, 150, 0, 45 );
58         break;
59     } // fin del switch
60
61     miGraphics.Dispose(); // libera el objeto Graphics
62 } // fin del método imagenComboBox_SelectedIndexChanged
63 } // fin de la clase PruebaComboBoxForm

```

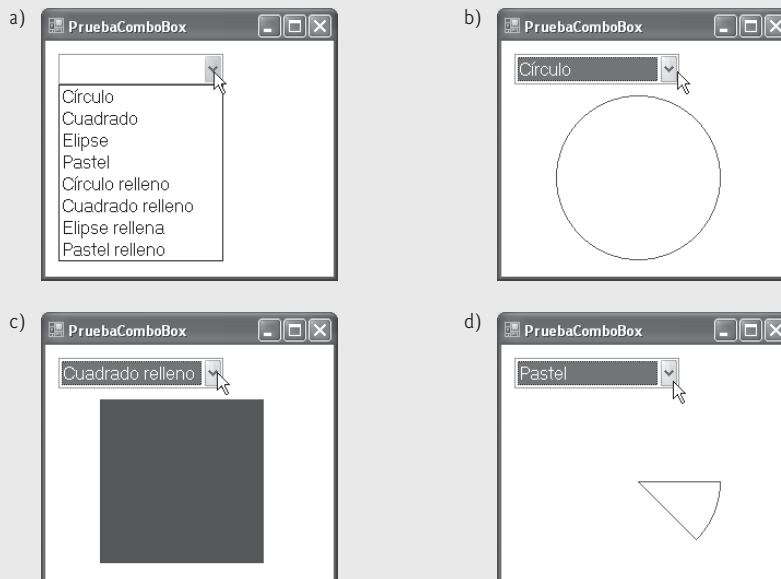


Figura 14.23 | Uso del control ComboBox para dibujar una figura seleccionada. (Parte 2 de 2).



Observación de apariencia visual 14.5

Haga las listas (como las de los controles ComboBox) editables sólo si el programa está diseñado para aceptar elementos que introduzca el usuario. De no ser así, el usuario podría tratar de introducir un elemento personalizado que sea inadecuado para los fines de su aplicación.

Después de crear el control ComboBox `imagenComboBox`, haga que no sea posible editarlo estableciendo su propiedad `DropDownStyle` a `DropDownList` en la ventana **Propiedades**. Después agregue los elementos `Círculo`, `Cuadrado`, `Elipse`, `Pastel`, `Círculo relleno`, `Cuadrado relleno`, `Elipse rellena` y `Pastel relleno` a la colección `Items`, mediante el **Editor de la colección Cadena**. Cada vez que el usuario selecciona un elemento de `imagenComboBox`, se produce un evento `SelectedIndexChanged` y se ejecuta el manejador de eventos `imagenComboBox_SelectedIndexChanged` (líneas 17-60). Las líneas 21-27 crean un objeto `Graphics`, un objeto `Pen` y

un objeto `SolidBrush`, los cuales se utilizan para dibujar en el formulario. El objeto `Graphics` (línea 21) permite que una pluma o brocha dibuje en un componente, usando uno de varios métodos de `Graphics`. Los métodos `DrawEllipse`, `DrawRectangle` y `DrawPie` (líneas 36, 39, 42 y 45) utilizan el objeto `Pen` (línea 24) para dibujar los contornos de sus correspondientes figuras. Los métodos `FillEllipse`, `FillRectangle` y `FillPie` (líneas 48, 51, 54 y 57) utilizan el objeto `SolidBrush` (línea 27) para rellenar sus correspondientes figuras sólidas. La línea 30 colorea todo el formulario de blanco (`White`), mediante el uso del método `Clear` de `Graphics`. En el capítulo 17, Gráficos y multimedia, hablaremos sobre estos métodos con mayor detalle.

La aplicación dibuja una figura con base en el índice del elemento seleccionado. La instrucción `switch` (líneas 33-59) utiliza `imagenComboBox.SelectedIndex` para determinar cuál elemento seleccionó el usuario. El método `DrawEllipse` (línea 36) de `Graphics` recibe un objeto `Pen`, las coordenadas x y y del centro, y la anchura y altura de la elipse para dibujarla. El origen del sistema de coordenadas se encuentra en la esquina superior izquierda del formulario; la coordenada x se incrementa a la derecha, y la coordenada y se incrementa hacia abajo. Un círculo es un caso especial de elipse (la altura y la anchura son iguales). La línea 36 dibuja un círculo. La línea 42 dibuja una elipse con valores distintos para la anchura y la altura.

El método `DrawRectangle` de la clase `Graphics` (línea 39) recibe un objeto `Pen`, las coordenadas x y y de la esquina superior izquierda, y la anchura y altura del rectángulo para dibujarlo. El método `DrawPie` (línea 45) dibuja un pastel como una porción de una elipse. La elipse está delimitada por un rectángulo. El método `DrawPie` recibe un objeto `Pen` las coordenadas x y y de la esquina superior izquierda del rectángulo, su anchura y altura, el ángulo inicial (en grados) y el ángulo de barrido (en grados) del pastel. Los ángulos se incrementan en sentido a favor de las manecillas del reloj. Los métodos `FillEllipse` (líneas 48 y 54), `FillRectangle` (línea 51) y `FillPie` (línea 57) son similares a sus contrapartes sin relleno, sólo que reciben un objeto `SolidBrush` en vez de un objeto `Pen`. En las capturas de pantalla de la figura 14.23 se muestran algunas de las figuras que se dibujan.

14.9 Control TreeView

El control `TreeView` muestra *nodos* de manera jerárquica en un *árbol*. Tradicionalmente, los nodos son objetos que contienen valores y pueden hacer referencia a otros nodos. Un *nodo padre* contiene *nodos hijos*, y los nodos hijos pueden ser padres de otros nodos. Dos nodos hijos que tienen el mismo nodo padre se consideran *nodos hermanos*. Un árbol es una colección de nodos que por lo general se organizan en forma jerárquica. El primer nodo padre de un árbol es el nodo *raíz* (un control `TreeView` puede tener varios nodos raíz). Por ejemplo, el sistema de archivos de una computadora puede representarse como un árbol. El directorio de nivel superior (tal vez `C:`) sería la raíz, cada subcarpeta de `C:` sería un nodo hijo, y cada carpeta hija tendría sus propios hijos. Los controles `TreeView` son útiles para mostrar información jerárquica, como la estructura de archivos que acabamos de mencionar. En el capítulo 24, Estructuras de datos, veremos los nodos y los árboles con más detalle. La figura 14.24 muestra un ejemplo de un control `TreeView` en un formulario.

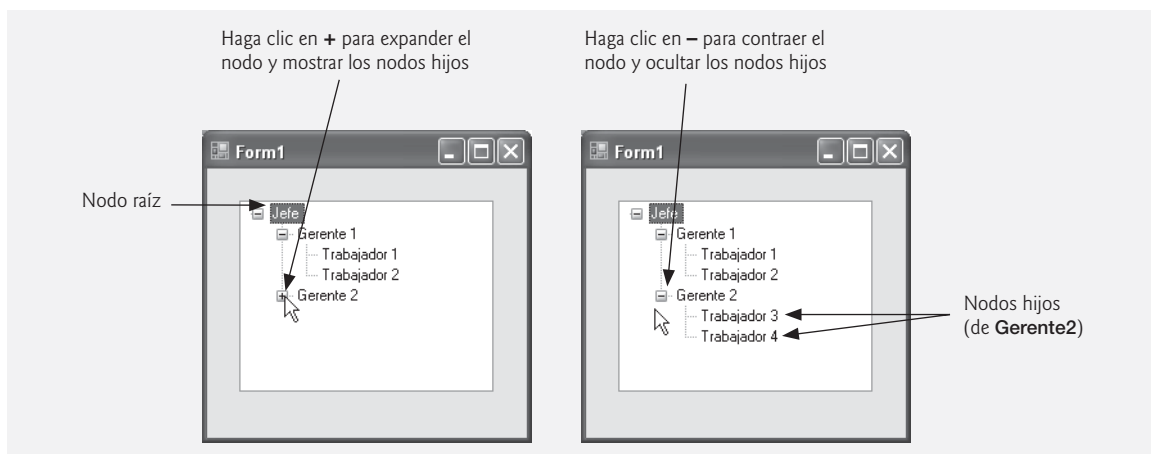


Figura 14.24 | El control `TreeView` que muestra un árbol de ejemplo.

Para expandir o contraer un nodo padre, hay que hacer clic en el cuadro con el signo más o en el cuadro con el signo menos que se encuentra a su derecha. Los nodos sin hijos no tienen estos cuadros.

Los nodos en un control `TreeView` son instancias de la clase **`TreeNode`**. Cada objeto `TreeNode` tiene una **colección `Nodes`** (tipo **`TreeNodeCollection`**), la cual contiene una lista de otros objetos `TreeNode`, conocidos como sus hijos. La propiedad `Parent` devuelve una referencia al nodo padre (o `null` si es un nodo raíz). Las figuras 14.25 y 14.26 listan las propiedades comunes de los controles `TreeView` y `TreeNode`, los métodos comunes de `TreeNode` y un evento común de `TreeView`.

Propiedades y evento de <code>TreeView</code>	Descripción
<i>Propiedades comunes</i>	
<code>CheckBoxes</code>	Indica si aparecen controles <code>CheckBox</code> a un lado de los nodos. Un valor de <code>true</code> muestra los controles <code>CheckBox</code> . El valor predeterminado es <code>false</code> .
<code>ImageList</code>	Especifica un objeto <code>ImageList</code> que contiene los iconos de los nodos. Un objeto <code>ImageList</code> es una colección que contiene objetos <code>Image</code> .
<code>Nodes</code>	Lista la colección de objetos <code>TreeNode</code> en el control. Contiene los métodos <code>Add</code> (agrega un objeto <code>TreeNode</code>), <code>Clear</code> (elimina toda la colección) y <code>Remove</code> (elimina un nodo específico). Al eliminar un nodo padre se eliminan todos sus hijos.
<code>SelectedNodes</code>	El nodo seleccionado.
<i>Evento común (argumentos <code>TreeViewEventArgs</code> del evento)</i>	
<code>AfterSelect</code>	Se genera cuando cambia el nodo seleccionado. Éste es el evento predeterminado cuando se hace doble clic en el control, estando en el diseñador.

Figura 14.25 | Propiedades y evento de `TreeView`.

Propiedades y métodos de <code>TreeNode</code>	Descripción
<i>Propiedades comunes</i>	
<code>Checked</code>	Indica si el objeto <code>TreeNode</code> está seleccionado (la propiedad <code>CheckBoxes</code> debe ser <code>true</code> en el objeto <code>TreeView</code> padre).
<code>FirstNode</code>	Especifica el primer nodo en la colección <code>Nodes</code> (es decir, el primer hijo en el árbol).
<code>FullPath</code>	Indica la ruta del nodo, empezando en la raíz del árbol.
<code>ImageIndex</code>	Especifica el índice de la imagen que se muestra cuando se deselecciona el nodo.
<code>LastName</code>	Especifica el último nodo en la colección <code>Nodes</code> (es decir, el último hijo en el árbol).
<code>NextNode</code>	El siguiente nodo hermano.
<code>Nodes</code>	Colección de objetos <code>TreeNode</code> contenida en el nodo actual (es decir, todos los hijos del nodo actual). Contiene los métodos <code>Add</code> (agrega un objeto <code>TreeNode</code>), <code>Clear</code> (borra toda la colección) y <code>Remove</code> (elimina un nodo específico). Al eliminar a un nodo padre se eliminan todos sus hijos.
<code>PrevNodes</code>	Nodo hermano anterior.
<code>SelectedImageIndex</code>	Especifica el índice de la imagen a usar cuando se seleccione el nodo.
<code>Text</code>	Especifica el texto del control <code>TreeNode</code> .

Figura 14.26 | Propiedades y métodos de `TreeNode`. (Parte 1 de 2).

Propiedades y métodos de TreeNode	Descripción
<i>Métodos comunes</i>	
Collapse	Contrae un nodo.
Expand	Expande un nodo.
ExpandAll	Expande todos los hijos de un nodo.
GetNodeCount	Devuelve el número de nodos hijos.

Figura 14.26 | Propiedades y métodos de TreeNode. (Parte 2 de 2).

Para agregar nodos al control TreeView en forma visual, haga clic en la elipsis que está a un lado de la propiedad Nodes, en la ventana **Propiedades**. A continuación se abrirá el **Editor TreeNode** (figura 14.27), el cual muestra un árbol vacío que representa el control TreeView. Hay botones para crear un nodo raíz, y para agregar o eliminar un nodo. A la derecha están las propiedades del nodo actual. Aquí puede cambiar el nombre del nodo.

Para agregar nodos por medio de la programación, primero cree un nodo raíz. Cree un nuevo objeto TreeNode y pase a este objeto un string para mostrarlo en pantalla. Después llame al método Add para agregar este nuevo objeto TreeNode a la colección Nodes de TreeView. Por ende, para agregar un nodo raíz al control TreeView llamado *miTreeView*, escriba

```
miTreeView.Nodes.Add( new TreeNode( raizLabel ) );
```

en donde *miTreeView* es el objeto TreeView al cual estamos agregando nodos, y *raizLabel* es el texto que se va a mostrar en *miTreeView*. Para agregar hijos a un nodo raíz, agregue nuevos objetos TreeNode a su colección Nodes. Para seleccionar el nodo raíz apropiado del control TreeView, escribimos

```
miTreeView.Nodes[ miIndice ]
```

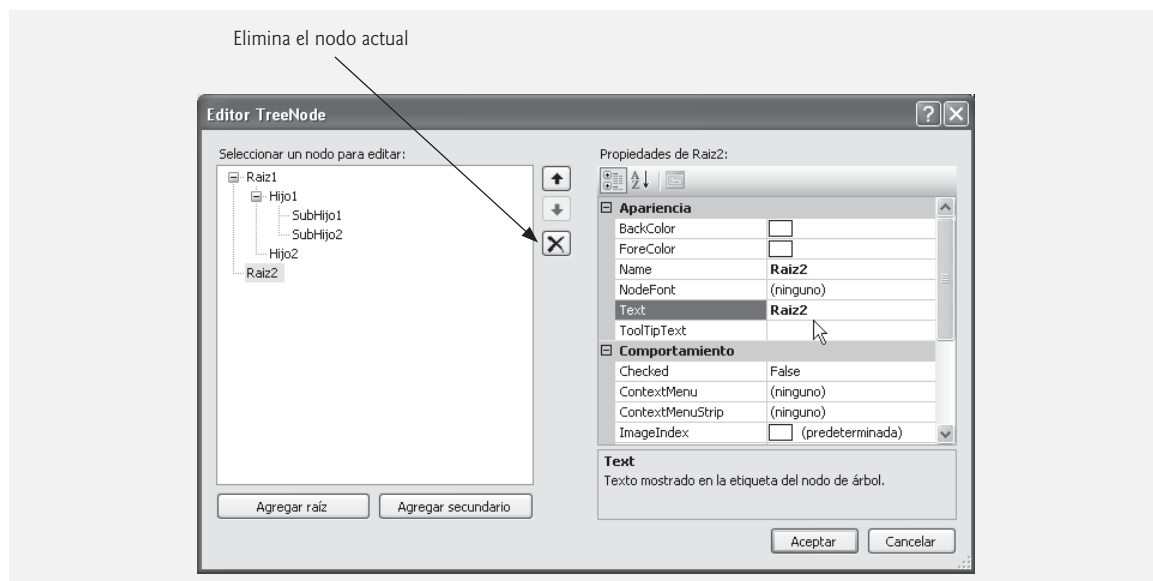


Figura 14.27 | El Editor TreeNode.

en donde *miIndice* es el índice del nodo raíz en la colección *Nodes* de *miTreeView*. Para agregar nodos a los hijos nodos, utilizamos el mismo proceso mediante el cual agregamos nodos raíz a *miTreeView*. Para agregar un hijo al nodo raíz en el índice *miIndice*, escribimos

```
miTreeView.Nodes[ miIndice ].Nodes.Add( new TreeNode( hijoLabel ) );
```

La clase *EstructuraDirectorioTreeViewForm* (figura 14.28) utiliza un control *TreeView* para mostrar el contenido de un directorio seleccionado por el usuario. Para especificar el directorio, se utilizan un control *TextBox* y un control *Button*. Primero es necesario escribir la ruta completa del directorio que deseamos que se muestre en pantalla. Después hay que hacer clic en el control *Button* para establecer el directorio especificado

```

1  // Fig. 14.28: EstructuraDirectorioTreeViewForm.cs
2  // Uso del control TreeView para mostrar una estructura de directorio.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6
7  // el formulario usa TreeView para mostrar la estructura del directorio
8  public partial class EstructuraDirectorioTreeViewForm : Form
9  {
10     string subcadenaDirectorio; // almacena la última parte del nombre de ruta completo
11
12     // constructor predeterminado
13     public EstructuraDirectorioTreeViewForm()
14     {
15         InitializeComponent();
16     } // fin del constructor
17
18     // llena el nodo actual con subdirectorios
19     public void LlenarTreeView(
20         string valorDirectorio, TreeNode nodoPadre )
21     {
22         // arreglo que almacena todos los subdirectorios en el directorio
23         string[] arregloDirectorios =
24             Directory.GetDirectories( valorDirectorio );
25
26         // llena el nodo actual con subdirectorios
27         try
28         {
29             // comprueba si hay subdirectorios
30             if ( arregloDirectorios.Length != 0 )
31             {
32                 // para cada subdirectorio, crea nuevo objeto TreeNode,
33                 // lo agrega como hijo del nodo actual y llena en forma
34                 // recursiva los nodos hijos con subdirectorios
35                 foreach ( string directorio in arregloDirectorios )
36                 {
37                     // obtiene la última parte del nombre de ruta completo
38                     // buscando la última ocurrencia de "\" y devolviendo la
39                     // parte del nombre de ruta que va después de esta ocurrencia
40                     subcadenaDirectorio = directorio.Substring(
41                         directorio.LastIndexOf( '\\' ) + 1,
42                         directorio.Length - directorio.LastIndexOf( '\\' ) - 1 );
43
44                     // crea objeto TreeNode para el directorio actual

```

Figura 14.28 | Uso del control *TreeView* para mostrar una estructura de directorio. (Parte 1 de 2).

```

45         TreeNode miNodo = new TreeNode( subcadenaDirectorio );
46
47         // agrega el directorio actual al nodo padre
48         nodoPadre.Nodes.Add( miNodo );
49
50         // llena en forma recursiva cada subdirectorio
51         LlenarTreeView( directorio, miNodo );
52     } // fin de foreach
53 } // fin de if
54 } // fin de try
55
56 // atrapa excepción
57 catch ( UnauthorizedAccessException )
58 {
59     nodoPadre.Nodes.Add( "Acceso denegado" );
60 } // fin de catch
61 } // fin del método LlenarTreeView
62
63 // maneja evento click de entrarButton
64 private void entrarButton_Click( object sender, EventArgs e )
65 {
66     // borra todos los nodos
67     directorioTreeView.Nodes.Clear();
68
69     // comprueba si existe el directorio introducido por el usuario
70     // si existe, llena el objeto TreeView,
71     // si no existe, muestra cuadro MessageBox de error
72     if ( Directory.Exists( entradaTextBox.Text ) )
73     {
74         // agrega el nombre de ruta completo a directorioTreeView
75         directorioTreeView.Nodes.Add( entradaTextBox.Text );
76
77         // inserta subcarpetas
78         LlenarTreeView(
79             entradaTextBox.Text, directorioTreeView.Nodes[ 0 ] );
80     } // fin de if
81     // muestra MessageBox de error si no encontró el directorio
82     else
83         MessageBox.Show( entradaTextBox.Text + " no se pudo encontrar.",
84             "No se encontró el directorio", MessageBoxButtons.OK,
85             MessageBoxIcon.Error );
86 } // fin del método entrarButton_Click
87 } // fin de la clase EstructuraDirectorioTreeViewForm

```

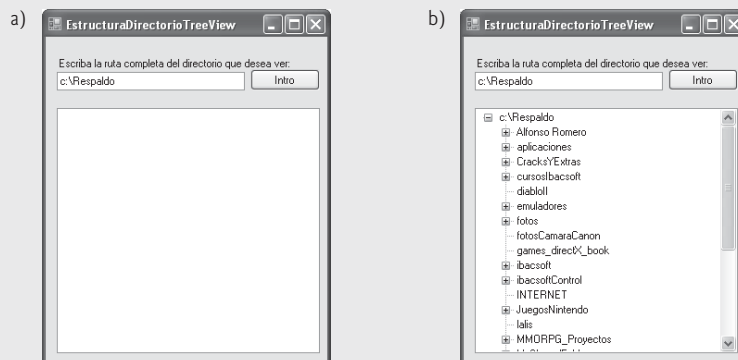


Figura 14.28 | Uso del control TreeView para mostrar una estructura de directorio. (Parte 2 de 2).

como el nodo raíz en el control `TreeView`. Cada subdirectorio de este directorio se convierte en un nodo hijo. Este esquema es similar al que se utiliza en el **Explorador de Windows**. Las carpetas pueden expandirse o contraerse haciendo clic en los cuadros con los signos más o menos que aparecen a su izquierda.

Cuando el usuario hace clic en el botón `entrarButton`, se borran todos los nodos en `directorioTreeView` (línea 67). Después se utiliza la ruta introducida en `entradaTextBox` para crear el nodo raíz. La línea 75 agrega el directorio a `directorioTreeView` como el nodo raíz, y las líneas 78-79 llaman al método `LlenarTreeView` (líneas 19-61), el cual recibe un directorio (un `string`) y un nodo padre. Luego, el método `LlenarTreeView` crea nodos hijos que corresponden a los subdirectorios del directorio que recibe como argumento.

El método `LlenarTreeView` (líneas 19-61) obtiene una lista de subdirectorios mediante el uso del método `GetDirectories` de la clase `Directory` (espacio de nombres `System.IO`) en las líneas 23-24. El método `GetDirectories` recibe un objeto `string` (el directorio actual) y devuelve un arreglo de objetos `string` (los subdirectorios). Si no se puede acceder a un directorio por razones de seguridad, se lanza una excepción `UnauthorizedAccessException`. Las líneas 57-60 atrapan esta excepción y agregan un nodo que contiene “Acceso denegado”, en vez de mostrar los subdirectorios.

Si hay directorios accesibles, las líneas 40-42 utilizan el método `Substring` para mejorar la legibilidad, acortando el nombre de ruta completo a sólo el nombre del directorio. A continuación, cada objeto `string` en `arregloDirectorios` se utiliza para crear un nuevo nodo hijo (línea 45). Utilizamos el método `Add` (línea 48) para agregar cada nodo hijo al padre. Después se hacen llamadas recursivas al método `LlenarTreeView` con cada subdirectorio (línea 51), lo que en cierto momento llena el control `TreeView` con toda la estructura de directorios. Observe que nuestro algoritmo recursivo puede provocar un retraso cuando el programa carga directorios extensos. No obstante, una vez que se agregan los nombres de las carpetas a la colección `Nodes` apropiada, pueden expandirse y contraerse sin retraso. En la siguiente sección, presentaremos un algoritmo alternativo para resolver este problema.

14.10 Control ListView

El control `ListView` es similar a un control `ListBox`, en cuanto a que ambos muestran listas de las que el usuario puede seleccionar uno o más elementos (en la figura 14.31 encontrará un ejemplo de un control `ListView`). La diferencia importante entre las dos clases es que un control `ListView` puede mostrar iconos enseguida de los elementos de la lista (los cuales se controlan mediante su propiedad `ImageList`). La propiedad `MultiSelect` (de tipo `Boolean`) determina si pueden seleccionarse varios elementos. Pueden incluirse controles `CheckBox`, para lo cual se establece la propiedad `CheckBoxes` (de tipo `Boolean`) a `true`, para dar al control `ListView` una apariencia similar a la de un control `CheckedListBox`. La propiedad `View` especifica el diseño del control `ListBox`. La propiedad `Activation` determina el método mediante el cual el usuario selecciona un elemento de la lista. En la figura 14.29 se explican los detalles de estas propiedades, junto con el evento `ItemActivate`.

Propiedades y evento de <code>ListView</code>	Descripción
<i>Propiedades comunes</i>	
<code>Activation</code>	Determina la forma en que el usuario activa un elemento. Esta propiedad recibe un valor en la enumeración <code>ItemActivation</code> . Los posibles valores son <code>OneClick</code> (activación por un solo clic), <code>TwoClick</code> (activación por dos clics, el elemento cambia de color cuando se selecciona) y <code>Standard</code> (activación con doble clic, el elemento no cambia de color).
<code>CheckBoxes</code>	Indica si los elementos aparecen con controles <code>CheckBox</code> . Si es <code>true</code> , se muestran los controles <code>CheckBox</code> . El valor predeterminado es <code>false</code> .
<code>LargeImageList</code>	Especifica el objeto <code>ImageList</code> que contiene iconos grandes para mostrar.

Figura 14.29 | Propiedades y evento de `ListView`. (Parte 1 de 2).

Propiedades y evento de ListView	Descripción
Items	Devuelve la colección de objetos ListViewItem en el control.
MultiSelect	Determina si se permite la selección múltiple. El valor predeterminado es true, que permite la selección múltiple.
SelectedItems	Obtiene la colección de elementos seleccionados.
SmallImageList	Especifica el objeto ImageList que contiene iconos pequeños para mostrar.
View	Determina la apariencia de los objetos ListViewItem. Los posibles valores son LargeIcon (se muestra un icono grande, los elementos pueden estar en varias columnas), SmallIcon (se muestra un icono pequeño, los elementos pueden estar en varias columnas), List (se muestran iconos pequeños, los elementos aparecen en una sola columna), Details (igual que List, pero pueden mostrarse varias columnas de información por cada elemento) y Tile (se muestran iconos grandes, la información se proporciona a la derecha del icono, válido sólo en Windows XP o posterior).
<i>Evento común</i>	
ItemActivate	Se genera cuando se activa un elemento en el control ListView. No contiene los detalles específicos acerca de cuál elemento se activó.

Figura 14.29 | Propiedades y evento de ListView. (Parte 2 de 2).

El control ListView le permite definir las imágenes que se usarán como iconos para sus elementos. Para mostrar imágenes se requiere un componente ImageList. Para crear uno, arrástrelo hacia un formulario desde el **Cuadro de herramientas**. Después seleccione la propiedad **Images** en la ventana **Propiedades** para mostrar el **Editor de la colección Imágenes** (figura 14.30). Aquí puede explorar en busca de imágenes que desea agregar al componente ImageList, el cual contiene un arreglo de objetos Image. Una vez definidas las imágenes, establezca la propiedad SmallImageList del control ListView con el nuevo objeto ImageList. La propiedad **SmallImageList** especifica la lista de imágenes para los iconos pequeños. La propiedad **LargeImageList** establece la lista de imágenes para los iconos grandes. Cada uno de los elementos en un control ListView son de tipo **ListViewItem**. Los iconos para los elementos de ListView se seleccionan estableciendo la propiedad **ImageIndex** del elemento con el índice apropiado.

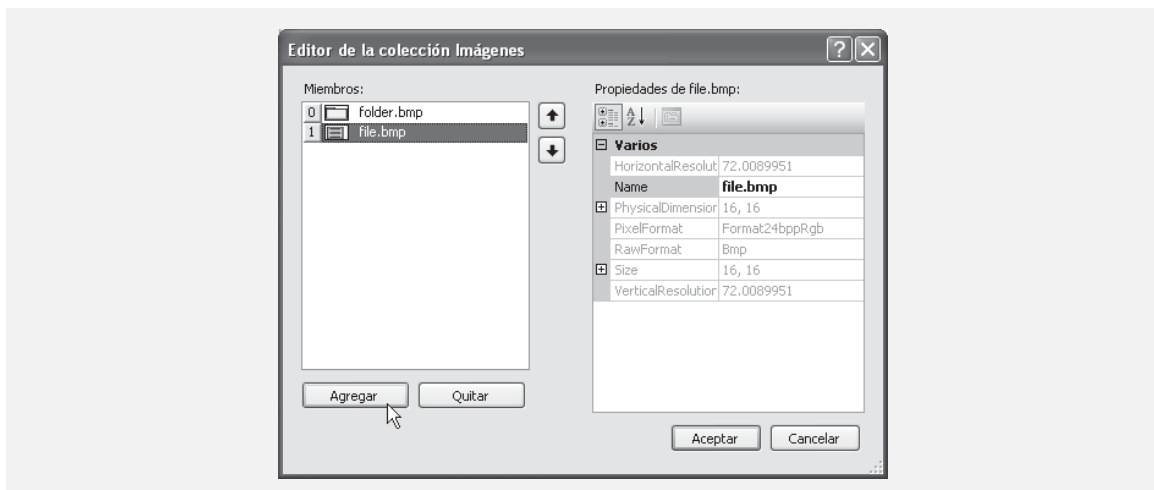


Figura 14.30 | La ventana **Editor de la colección Imágenes** para un componente ImageList.

La clase `PruebaListViewForm` (figura 14.31) muestra los archivos y carpetas en un control `ListView`, junto con pequeños iconos que representan a cada archivo o carpeta. Si un archivo o carpeta es inaccesible debido a las configuraciones de los permisos, aparece un cuadro de diálogo `MessageBox`. El programa explora el contenido del directorio a medida que navega por él, en vez de indexar toda la unidad de una sola vez.

Para mostrar iconos al lado de los elementos de la lista, cree un componente `ImageList` para el control `ListView` `exploradorListView`. Primero, arrastre y suelte un componente `ImageList` en el formulario, y abra el **Editor de la colección Imágenes**. Selecciona nuestras dos imágenes de mapa de bits simples, que se encuentran en la carpeta `bin\Release` de este ejemplo; uno para una carpeta (índice de arreglo 0) y el otro para un archivo (índice de arreglo 1). Después establezca la propiedad `SmallImageList` del objeto `exploradorListView` con el nuevo componente `ImageList` en la ventana **Propiedades**.

El método `CargarArchivosEnDirectorio` (líneas 63-110) llena el control `exploradorListView` con el directorio que recibe como argumento (`valorDirectorioActual`). Borra `exploradorListView` y agrega el elemento "Subir un nivel". Cuando el usuario hace clic en este elemento, el programa trata de subir un nivel (en breve veremos cómo). Después, el método crea un objeto `DirectoryInfo` que se inicializa con el objeto `string` `directorioActual` (líneas 74-75). Si no se otorga el permiso para explorar el directorio, se lanza una excepción (y se atrapa en la línea 104). El método `CargarArchivosEnDirectorio` trabaja en forma distinta al método

```

1  // Fig. 14.31: PruebaListViewForm.cs
2  // Mostrar directorios y su contenido en un control ListView.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6  using System.IO;
7
8  // el formulario contiene un control ListView, el cual muestra
9  // las carpetas y los archivos en un directorio
10 public partial class PruebaListViewForm : Form
11 {
12     // almacena el directorio actual
13     string directorioActual = Directory.GetCurrentDirectory();
14
15     // constructor predeterminado
16     public PruebaListViewForm()
17     {
18         InitializeComponent();
19     } // fin del constructor
20
21     // explora directorio en el que hizo clic el usuario, o sube un nivel
22     private void exploradorListView_Click( object sender, EventArgs e )
23     {
24         // asegura que se seleccione un elemento
25         if ( exploradorListView.SelectedItems.Count != 0 )
26         {
27             // si se seleccionó el primer elemento, sube un nivel
28             if ( exploradorListView.Items[ 0 ].Selected )
29             {
30                 // crea objeto DirectoryInfo para el directorio
31                 DirectoryInfo objetoDirectorio =
32                     new DirectoryInfo( directorioActual );
33
34                 // si el directorio tiene padre, lo carga
35                 if ( objetoDirectorio.Parent != null )
36                     CargarArchivosEnDirectorio( objetoDirectorio.Parent.FullName );
37             } // fin de if

```

Figura 14.31 | Control `ListView` que muestra archivos y carpetas. (Parte 1 de 3).

```

38
39 // seleccionó directorio o archivo
40 else
41 {
42     // se eligió un directorio o un archivo
43     string elegido = exploradorListView.SelectedItems[ 0 ].Text;
44
45     // si el elemento seleccionado es un directorio, carga el directorio
46     // seleccionado
47     if ( Directory.Exists( directorioActual + @"\" + elegido ) )
48     {
49         // si se encuentra actualmente en C:\, no necesita '\'; en caso
50         // contrario sí
51         if ( directorioActual == @"C:\" )
52             CargarArchivosEnDirectorio( directorioActual + elegido );
53         else
54             CargarArchivosEnDirectorio(
55                 directorioActual + @"\" + elegido );
56     } // fin de if
57 } // fin de else
58
59 // actualiza mostrarLabel
60 mostrarLabel.Text = directorioActual;
61 } // fin de if
62 } // fin del método exploradorListView_Click
63
64 // muestra archivos/subdirectorios del directorio actual
65 public void CargarArchivosEnDirectorio( string valorDirectorioActual )
66 {
67     // carga información del directorio y la muestra en pantalla
68     try
69     {
70         // borra control ListView y establece el primer elemento
71         exploradorListView.Items.Clear();
72         exploradorListView.Items.Add( "Subir un nivel" );
73
74         // actualiza directorio actual
75         directorioActual = valorDirectorioActual;
76         DirectoryInfo nuevoDirectorioActual =
77             new DirectoryInfo( directorioActual );
78
79         // coloca archivos y directorios en arreglos
80         DirectoryInfo[] arregloDirectorios =
81             nuevoDirectorioActual.GetDirectories();
82         FileInfo[] arregloArchivos = nuevoDirectorioActual.GetFiles();
83
84         // agrega los nombres de los directorios al control ListView
85         foreach ( DirectoryInfo dir in arregloDirectorios )
86         {
87             // agrega directorio a ListView
88             ListViewItem nuevoElementoDirectorio =
89                 exploradorListView.Items.Add( dir.Name );
90
91             nuevoElementoDirectorio.ImageIndex = 0; // establece imagen del
92             directorio
93         } // fin de foreach
94
95         // agrega los nombres de archivo a ListView

```

Figura 14.31 | Control ListView que muestra archivos y carpetas. (Parte 2 de 3).

```

93     foreach ( FileInfo file in arregloArchivos )
94     {
95         // agrega archivo a ListView
96         ListViewItem nuevoElementoArchivo =
97             exploradorListView.Items.Add( file.Name );
98
99         nuevoElementoArchivo.ImageIndex = 1; // establece imagen del archivo
100     } // fin de foreach
101 } // fin de try
102
103 // acceso denegado
104 catch ( UnauthorizedAccessException )
105 {
106     MessageBox.Show( "Advertencia: tal vez algunos campos no estén " +
107         "visibles debido a la configuración de los permisos",
108         "Atención", 0, MessageBoxIcon.Warning );
109 } // fin de catch
110 } // fin del método CargarArchivosEnDirectorio
111
112 // maneja evento de carga cuando el formulario se muestra por primera vez
113 private void PruebaListViewForm_Load( object sender, EventArgs e )
114 {
115     // establece lista de objetos Image
116     Image imagenCarpeta = Image.FromFile(
117         directorioActual + @"\imagenes\carpeta.bmp" );
118
119     Image imagenArchivo = Image.FromFile(
120         directorioActual + @"\imagenes\archivo.bmp" );
121
122     archivoCarpeta.Images.Add( imagenCarpeta );
123     archivoCarpeta.Images.Add( imagenArchivo );
124
125     // carga el directorio actual en el control exploradorListView
126     CargarArchivosEnDirectorio( directorioActual );
127     mostrarLabel.Text = directorioActual;
128 } // fin del método PruebaListViewForm_Load
129 } // fin de la clase PruebaListViewForm

```

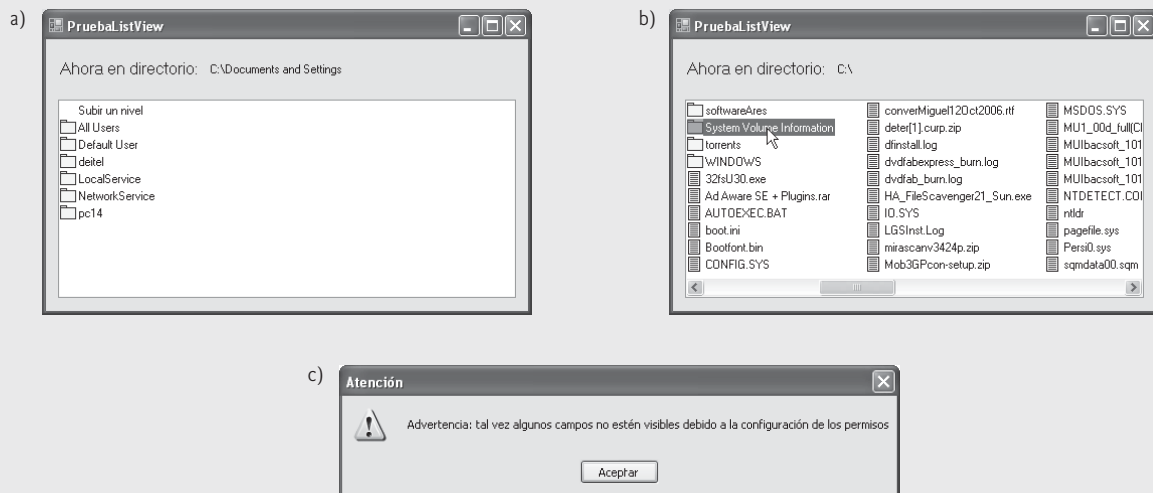


Figura 14.31 | Control ListView que muestra archivos y carpetas. (Parte 3 de 3).

LlenarTreeView en el programa anterior (figura 14.28). En vez de cargar todas las carpetas en el disco duro, el método CargarArchivosEnDirectorio sólo carga las carpetas en el directorio actual.

La clase **DirectoryInfo** (espacio de nombres System.IO) nos permite explorar o manipular la estructura de directorios con facilidad. El método **GetDirectories** (línea 79) devuelve un arreglo de objetos DirectoryInfo, los cuales contienen los subdirectorios del directorio actual. De manera similar, el método **GetFiles** (línea 80) devuelve un arreglo de objetos de la clase **FileInfo**, que contienen los archivos en el directorio actual. La propiedad **Name** (tanto de la clase DirectoryInfo como de la clase FileInfo) sólo contiene el nombre del directorio o archivo, como temp en vez de C:\micarpeta\temp. Para acceder al nombre completo, use la propiedad **FullName**.

Las líneas 83-90 y las líneas 93-100 iteran a través de los subdirectorios y archivos del directorio actual, y los agregan a exploradorListView. Las líneas 89 y 99 establecen las propiedades ImageIndex de los elementos recién creados. Si un elemento es un directorio, establecemos su icono a un icono de directorio (índice 0); si es un archivo, establecemos su icono a un icono de archivo (índice 1);

El método exploradorListView_Click (líneas 22-60) responde cuando el usuario hace clic en el control exploradorListView. La línea 25 comprueba si hay algo seleccionado. Si se hizo una selección, la línea 28 determina si el usuario seleccionó el primer elemento en exploradorListView. El primer elemento siempre es **Subir un nivel**; si está seleccionado, el programa trata de subir un nivel. Las líneas 31-32 crean un objeto DirectoryInfo para el directorio actual. La línea 35 establece la propiedad Parent para asegurar que el usuario no se encuentre en la raíz del árbol de directorios. La propiedad **Parent** indica el directorio padre como un objeto DirectoryInfo; si no existe un directorio padre, Parent devuelve el valor null. Si existe un directorio padre, entonces la línea 36 pasa el nombre completo del directorio padre al método CargarArchivosEnDirectorio.

Si el usuario no seleccionó el primer elemento en exploradorListView, las líneas 40-55 permiten que el usuario continúe navegando a través de la estructura de directorios. La línea 43 crea el objeto string elegido, el cual recibe el texto del elemento seleccionado (el primer elemento en la colección SelectedItems). La línea 46 determina si el usuario seleccionó un directorio válido (en vez de un archivo). El programa combina las variables directorioActual y elegido (el nuevo directorio), separadas por una barra diagonal inversa (\), y pasa este valor al método **Exists** de la clase Directory. Este método devuelve true si su parámetro string es un directorio. Si esto ocurre, el programa pasa el objeto string al método CargarArchivosEnDirectorio. Como el directorio C:\ ya incluye una barra diagonal inversa, no se necesita una para combinar directorioActual y elegido (línea 50). No obstante, otros directorios deben incluir la barra diagonal (líneas 52-53). Por último, mostrarLabel se actualiza con el nuevo directorio (línea 58).

Este programa se carga rápidamente, ya que sólo indexa los archivos en el directorio actual. Esto significa que puede ocurrir un pequeño retraso cuando se carga un nuevo directorio. Además, los cambios en la estructura de directorios pueden mostrarse al recargar un directorio. El programa anterior (figura 14.28) podría tener un largo retraso inicial al cargar toda una estructura de directorios completa. Este tipo de concesión es muy común en el mundo del software.



Observación de ingeniería de software 14.2

Al diseñar aplicaciones que se ejecutan durante periodos extensos de tiempo, podría ser conveniente elegir un largo retraso inicial para mejorar el rendimiento durante el resto del programa. No obstante, en aplicaciones que se ejecutan por periodos cortos, los desarrolladores por lo general prefieren tiempos de carga inicial rápidos y pequeños retrasos después de cada acción.

14.11 Control TabControl

El control **TabControl** crea ventanas con fichas, como las que hemos visto en Visual Studio (figura 14.32). Esto le permite especificar más información en el mismo espacio de un formulario.

Los controles TabControl contienen objetos **TabPage**, que son similares a los controles Panel y GroupBox en cuanto a que los controles TabPage pueden contener controles. Primero se agregan controles a los objetos TabPage y después se agregan los objetos TabPage al control TabControl. Sólo se muestra un objeto TabPage a la vez. Para agregar objetos a TabPage y TabControl, escriba

```
miTabPage.Controls.Add(miControl)
miTabControl.Controls.Add(miTabPage)
```

Estas instrucciones llaman al método `Add` de la colección `Controls`. El ejemplo agrega el control `TabControl` *miControl* al objeto `TabPage` *miTabPage*, y después agrega *miTabPage* a *miTabControl*. De manera alternativa, podemos usar el método `AddRange` para agregar un arreglo de objetos `TabPage` o controles a un control `TabControl` o a un objeto `TabPage`, respectivamente. La figura 14.33 describe el uso de un control `TabControl` de ejemplo.

Para agregar controles `TabControl` en forma visual, arrástrelos y suéltelos en un formulario, en modo de **Diseño**. Para agregar objetos `TabPage` en modo de **Diseño**, haga clic derecho en el control `TabControl` y seleccione **Agregar ficha** (figura 14.34). Como alternativa, haga clic en la propiedad **TabPage** en la ventana **Propiedades** y agregue fichas en el cuadro de diálogo que aparezca. Para modificar la etiqueta de una ficha, establezca la propiedad **Text** del objeto `TabPage`. Observe que al hacer clic en las fichas, se selecciona el control `TabControl`; para seleccionar el objeto `TabPage`, haga clic en el área de control debajo de las fichas. Puede agregar controles al objeto `TabPage`, arrastrando y soltando elementos desde el **Cuadro de herramientas**. Para ver distintos objetos `TabPage`, haga clic en la ficha apropiada (ya sea en modo de diseño, o en modo de ejecución). En la figura 14.35 se describen las propiedades comunes y un evento común de los controles `TabControl`.

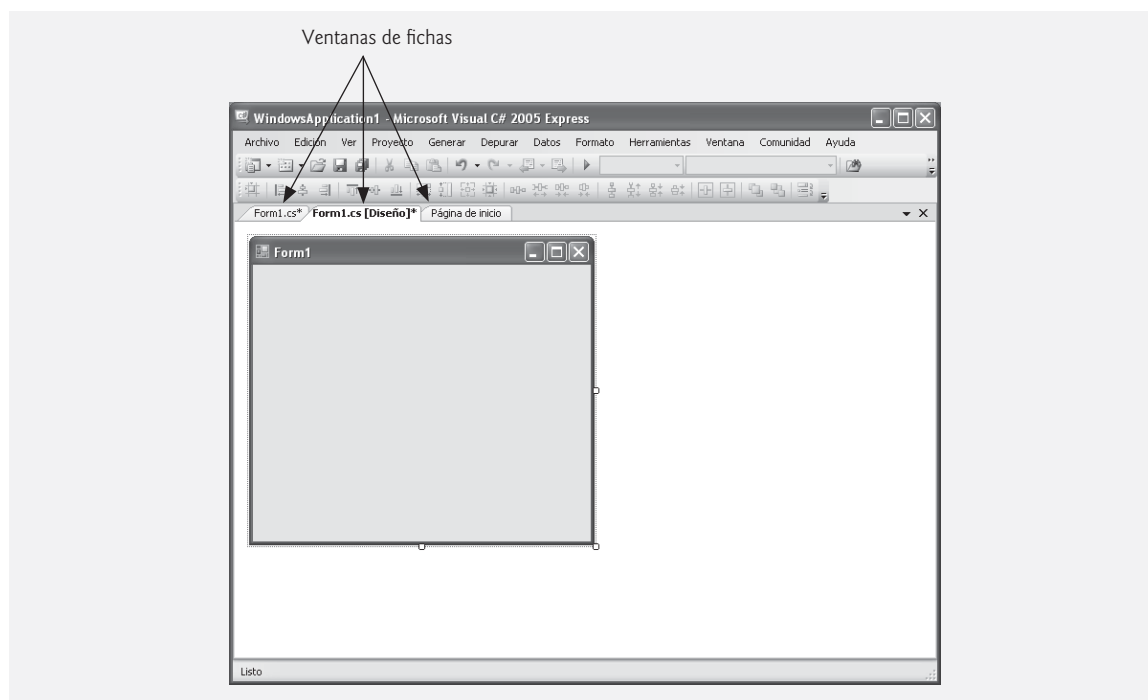


Figura 14.32 | Ventanas con fichas en Visual Studio.

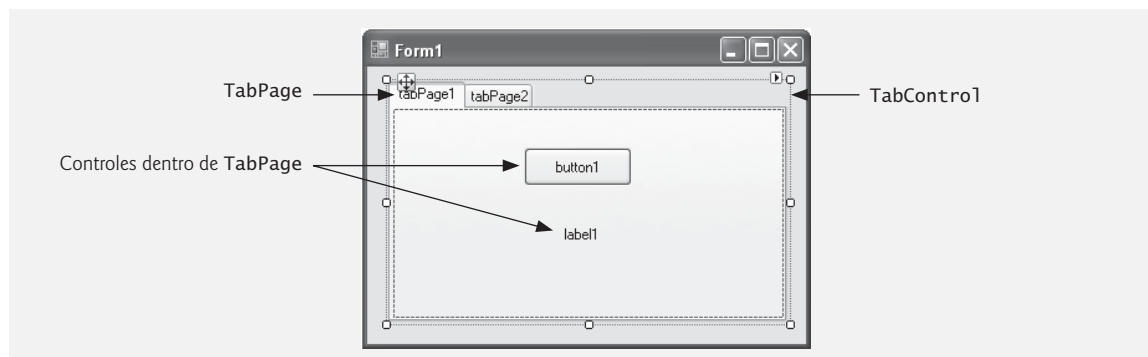


Figura 14.33 | Ejemplo de un control `TabControl` con objetos `TabPage`.

Cada objeto `TabPage` genera un evento `Click` cuando se hace clic en su ficha. Para crear los manejadores de eventos para este evento, haga doble clic en el cuerpo del control `TabPage`.

La clase `UsoDeFichasForm` (figura 14.36) utiliza un control `TabControl` para mostrar varias opciones relacionadas con el texto en una etiqueta (**Color**, **Size** y **Message**). El último objeto `TabPage` muestra un mensaje **Acerca de**, el cual describe el uso de los controles `TabControl`.

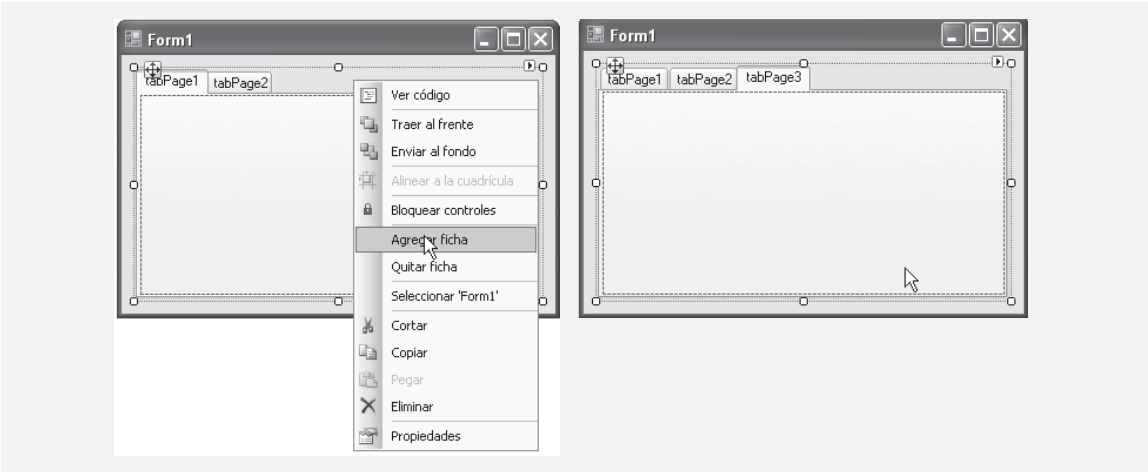


Figura 14.34 | Agregar objetos `TabPage` a un control `TabControl`.

Propiedades y evento de <code>TabControl</code>	Descripción
<i>Propiedades comunes</i>	
<code>ImageList</code>	Especifica las imágenes que se van a mostrar en las fichas.
<code>ItemSize</code>	Especifica el tamaño de las fichas.
<code>Multiline</code>	Indica si pueden mostrarse varias filas de fichas.
<code>SelectedIndex</code>	Índice del objeto <code>TabPage</code> seleccionado.
<code>SelectedTab</code>	El objeto <code>TabPage</code> seleccionado.
<code>TabCount</code>	Devuelve el número de páginas de fichas.
<code>TabPage</code> s	Collección de objetos <code>TabPage</code> s dentro del control <code>TabControl</code> .
<i>Evento común</i>	
<code>SelectedIndexChanged</code>	Se genera cuando cambia la propiedad <code>SelectedIndex</code> (es decir, cuando se selecciona otro objeto <code>TabPage</code>).

Figura 14.35 | Propiedades y evento de `TabControl`.

```
1 // Fig. 14.36: UsoDeFichasForm.cs
2 // Uso del control TabControl para mostrar varias opciones de fuente.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6
```

Figura 14.36 | Uso del control `TabControl` para mostrar varias opciones de fuente. (Parte 1 de 3).

```

7 // el formulario usa fichas y botones de opción para mostrar varias opciones de fuente
8 public partial class UsoDeFichasForm : Form
9 {
10     // constructor predeterminado
11     public UsoDeFichasForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // manejador de eventos para el control RadioButton Negro
17     private void negroRadioButton_CheckedChanged(
18         object sender, EventArgs e )
19     {
20         mostrarLabel.ForeColor = Color.Black; // cambia color de fuente a negro
21     } // fin del método negroRadioButton_CheckedChanged
22
23     // manejador de eventos para el control RadioButton Rojo
24     private void rojoRadioButton_CheckedChanged(
25         object sender, EventArgs e )
26     {
27         mostrarLabel.ForeColor = Color.Red; // cambia color de fuente a rojo
28     } // fin del método rojoRadioButton_CheckedChanged
29
30     // manejador de eventos para el control RadioButton Verde
31     private void verdeRadioButton_CheckedChanged(
32         object sender, EventArgs e )
33     {
34         mostrarLabel.ForeColor = Color.Green; // cambia color de fuente a verde
35     } // fin del método verdeRadioButton_CheckedChanged
36
37     // manejador de eventos para el control RadioButton de 12 puntos
38     private void tamaño12RadioButton_CheckedChanged(
39         object sender, EventArgs e )
40     {
41         // cambia el tamaño de la fuente a 12
42         mostrarLabel.Font = new Font( mostrarLabel.Font.Name, 12 );
43     } // fin del método tamaño12RadioButton_CheckedChanged
44
45     // manejador de eventos para el control RadioButton de 16 puntos
46     private void tamaño16RadioButton_CheckedChanged(
47         object sender, EventArgs e )
48     {
49         // cambia el tamaño de la fuente a 16
50         mostrarLabel.Font = new Font( mostrarLabel.Font.Name, 16 );
51     } // fin del método tamaño16RadioButton_CheckedChanged
52
53     // manejador de eventos para el control RadioButton de 20 puntos
54     private void tamaño20RadioButton_CheckedChanged(
55         object sender, EventArgs e )
56     {
57         // cambia el tamaño de la fuente a 20
58         mostrarLabel.Font = new Font( mostrarLabel.Font.Name, 20 );
59     } // fin del método tamaño20RadioButton_CheckedChanged
60
61     // manejador de eventos para el control RadioButton ¡Hola!
62     private void holaRadioButton_CheckedChanged(
63         object sender, EventArgs e )
64     {

```

Figura 14.36 | Uso del control TabControl para mostrar varias opciones de fuente. (Parte 2 de 3).

```

65     mostrarLabel.Text = "¡Hola!"; // cambia el texto a ¡Hola!
66 } // fin del método holaRadioButton_CheckedChanged
67
68 // manejador de eventos para el control RadioButton ¡Adiós!
69 private void adiosRadioButton_CheckedChanged(
70     object sender, EventArgs e )
71 {
72     mostrarLabel.Text = "¡Adiós!"; // cambia el texto a ¡Adiós!
73 } // fin del método adiosRadioButton_CheckedChanged
74 } // fin de la clase UsoDeFichasForm

```

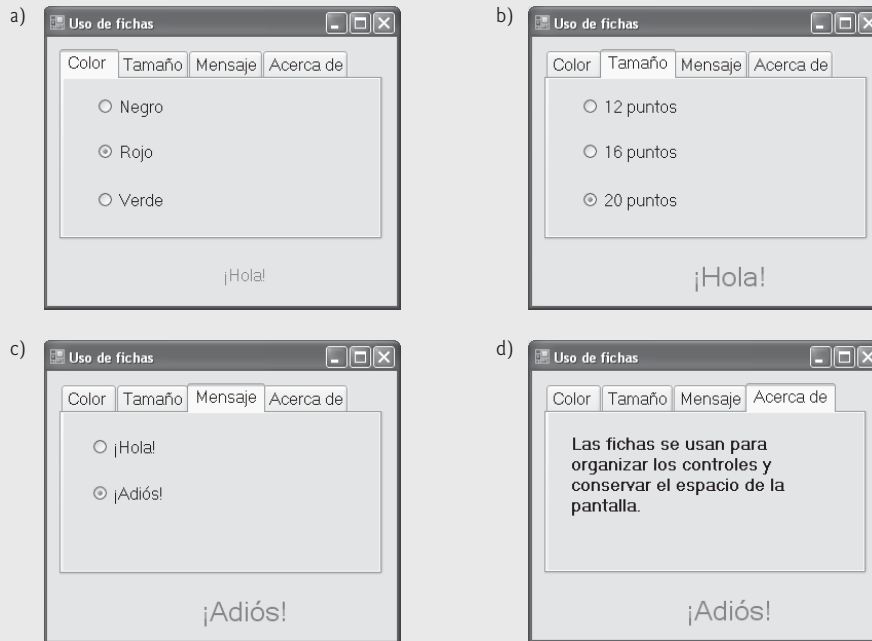


Figura 14.36 | Uso del control TabControl para mostrar varias opciones de fuente. (Parte 3 de 3).

El control opcionesDeTextoTabControl y las fichas colorTabPage, tamañoTabPage, mensajeTabPage y acercaDeTabPage se crean en el diseñador (como describimos antes). La ficha colorTabPage contiene tres controles RadioButton para los colores negro (negroRadioButton), rojo (rojoRadioButton) y verde (verdeRadioButton). Este objeto tabPage se muestra en la figura 14.36(a). El manejador de eventos CheckChanged para cada control RadioButton actualiza el color del texto en mostrarLabel (líneas 20, 27 y 34). La ficha tamañoTabPage (figura 14.36(b)) tiene tres controles RadioButton, los cuales corresponden a los tamaños de fuente de 12 (tamaño12RadioButton), 16 (tamaño16RadioButton) y 20 (tamaño20RadioButton), que modifican el tamaño de la fuente de mostrarLabel (líneas 42, 50 y 58, respectivamente). La ficha mensajeTabPage (figura 14.36(c)) contiene dos controles RadioButton para los mensajes ¡Hola! (holaRadioButton) y ¡Adiós! (adiosRadioButton). Los dos controles RadioButton determinan el texto en mostrarLabel (líneas 65 y 72, respectivamente). La ficha acercaDeTabPage (figura 14.36(d)) contiene un control Label (mensajeLabel) que describe el propósito de los controles TabControl.



Observación de ingeniería de software 14.3

Un objeto tabPage puede actuar como un contenedor para un solo grupo lógico de controles RadioButton, para cumplir con su exclusividad mutua. Para colocar varios grupos de controles RadioButton dentro de un solo objeto tabPage, debe agrupar los controles RadioButton dentro de controles Panel o GroupBox que se encuentren dentro del objeto tabPage.

14.12 Ventanas de la interfaz de múltiples documentos (MDI)

En capítulos anteriores sólo hemos construido aplicaciones con la *interfaz de un solo documento (SDI)*. Tales programas (incluyendo el Bloc de notas y Paint de Microsoft) sólo pueden soportar una ventana o documento abierto a la vez. Por lo general, las aplicaciones SDI tienen habilidades limitadas; por ejemplo, Paint y Bloc de notas tienen características limitadas de edición de imágenes y texto. Para editar varios documentos, el usuario debe ejecutar otra instancia de la aplicación SDI.

Las aplicaciones más recientes son programas con la *interfaz de múltiples documentos (MDI)*, que permiten a los usuarios editar varios documentos a la vez (por ejemplo, los productos de Microsoft Office). Los programas MDI también tienden a ser más complejos; PaintShop Pro y Photoshop tienen un mayor número de características para editar imágenes que Paint.

A la ventana de aplicación principal de un programa MDI se le conoce como *ventana padre*, y a cada ventana dentro de la aplicación se le conoce como *ventana hija*. Aunque una aplicación MDI puede tener muchas ventanas hijas, cada una tiene sólo una ventana padre. Lo que es más, sólo puede haber como máximo una ventana hija activa a la vez. Las ventanas hijas no pueden ser padres y tampoco pueden moverse fuera de su padre. De no ser así, una ventana hija se comporta igual que cualquier otra ventana (con respecto a cerrar, minimizar, cambiar de tamaño, etc.). La funcionalidad de una ventana hija puede ser distinta de la funcionalidad de otras ventanas hijas del padre. Por ejemplo, una ventana hija podría permitir al usuario editar imágenes, otra podría permitir al usuario editar texto y una tercera podría mostrar el tráfico de la red en forma gráfica, pero todas podrían pertenecer al mismo padre MDI. La figura 14.37 ilustra una aplicación MDI de ejemplo.

Para crear un formulario MDI, cree un nuevo formulario y establezca su propiedad `IsMdiContainer` a `true`. El formulario cambiará de apariencia, como en la figura 14.38.

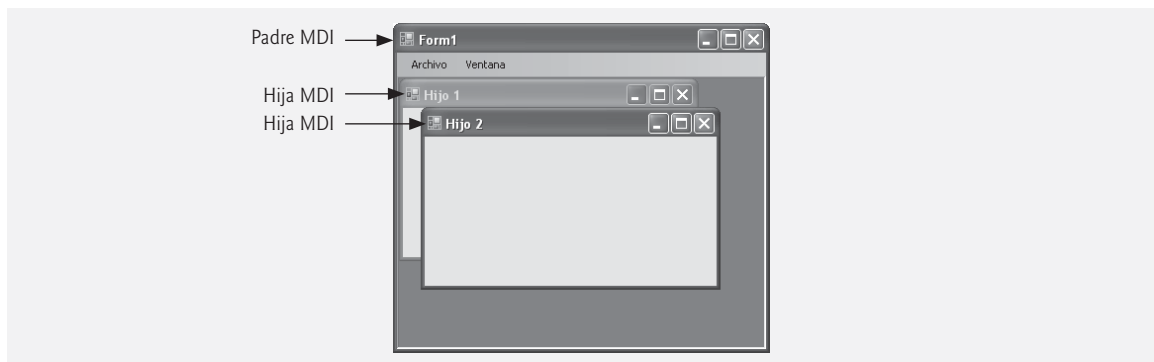


Figura 14.37 | Ventana padre MDI y ventanas hijas MDI.

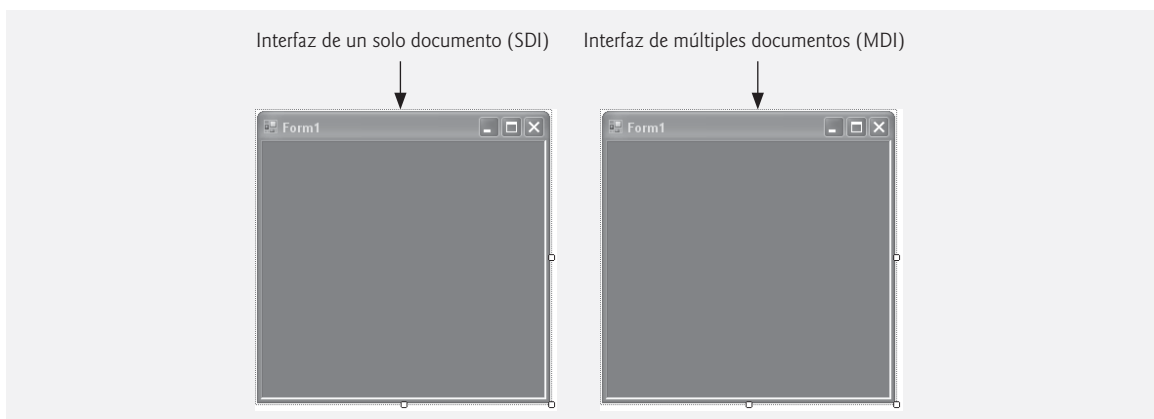


Figura 14.38 | Formularios SDI y MSI.

A continuación, cree una clase `Form` hija para agregarla al formulario. Para ello, haga clic con el botón derecho del ratón en el **Explorador de soluciones** del proyecto, seleccione **Proyecto > Agregar Windows Forms...** y nombre el archivo. Edite el formulario según lo desee. Para agregar e formulario hijo al padre, debemos crear un nuevo objeto `Form` hijo, asignar a la propiedad **`MdiParent`** el formulario padre y llamar al método `Show` del formulario. En general, para agregar un formulario hijo a un padre, escribimos

```
ClaseFormHijo hijoForm = New ClaseFormHijo();
hijoForm.MdiParent = padreForm;
hijoForm.Show();
```

En la mayoría de los casos, el formulario padre crea al hijo, por lo que la referencia *padreForm* es `this`. Por lo general, el código para crear un hijo se encuentra dentro de un manejador de eventos, el cual crea una nueva ventana en respuesta a una acción del usuario. Las selecciones de menús (como **Archivo**, seguido de la opción de submenú **Nuevo**, seguido de la opción de submenú **Ventana**) son técnicas comunes para crear nuevas ventanas hijas.

La propiedad **`MdiChildren`** de la clase `Form` devuelve un arreglo de referencias `Form` hijas. Esto es útil si la ventana padre desea comprobar el estado de todas sus hijas (por ejemplo, asegurar que todas se guarden antes de que cierre la ventana padre). La propiedad **`ActiveMdiChild`** devuelve una referencia a la ventana hija activa; devuelve `Nothing` si no hay ventanas hijas activas. En la figura 14.39 se describen otras características de las ventanas MDI.

Las ventanas hijas se pueden minimizar, maximizar y cerrar en forma independiente unas de otras, y de la ventana padre. La figura 14.40 muestra dos imágenes: una contiene dos ventanas hijas minimizadas y la otra contiene una ventana hija minimizada. Cuando el padre se minimiza o se cierra, las ventanas hijas también se minimizan o se cierran. Observe que la barra de título en la figura 14.40(b) es **Form1 – [Hijo2]**. Cuando se maximiza una ventana hija, el texto de su barra de título se inserta en la barra de título de la ventana padre. Cuando se minimiza o maximiza una ventana hija, su barra de título muestra un icono de restauración, el cual puede usarse para regresar la ventana hija a su tamaño anterior (al que tenía antes de maximizarse o minimizarse).

Propiedades, un método y un evento de los formularios MDI	Descripción
<i>Propiedades comunes de los hijos MDI</i>	
<code>IsMdiChild</code>	Indica si el formulario es un hijo MDI. Si es true, el formulario es un hijo MDI (propiedad de sólo lectura).
<code>MdiParent</code>	Especifica el formulario padre MDI del hijo.
<i>Propiedades comunes de los padres MDI</i>	
<code>ActiveMdiChild</code>	Devuelve el formulario hijo MDI que está activo en un momento dado (devuelve null si no hay hijos activos).
<code>IsMdiContainer</code>	Indica si un formulario puede ser un padre MDI. Si es true, el formulario puede ser un padre MDI. El valor predeterminado es false.
<code>MdiChildren</code>	Devuelve los hijos MDI como un arreglo de objetos <code>Form</code> .
<i>Método común</i>	
<code>LayoutMdi</code>	Determina la distribución de los formularios hijos en un padre MDI. El método recibe como parámetro una enumeración <code>MdiLayout</code> con los posibles valores <code>ArrangeIcons</code> , <code>Cascade</code> , <code>TileHorizontal</code> y <code>TileVertical</code> . La figura 14.42 ejemplifica los efectos de estos valores.
<i>Evento común</i>	
<code>MdiChildActivate</code>	Se genera cuando se cierra o se activa un formulario hijo MDI.

Figura 14.39 | Propiedades, métodos y eventos de las ventanas padre MDI y las ventanas hijas MDI.

C# cuenta con una propiedad que ayuda a llevar la cuenta de cuáles ventanas hijas están abiertas en un contenedor MDI. La propiedad **MdiWindowListItem** de la clase **MenuStrip** especifica qué menú (si lo hay) va a mostrar una lista de ventanas hijas abiertas. Cuando se abre una nueva ventana hija, se agrega una entrada a la lista (como en la primera pantalla de la figura 14.41). Si se abren nueve ventanas hijas o más, la lista incluye la opción **Más ventanas...**, la cual permite al usuario seleccionar una ventana de una lista en un cuadro de diálogo.

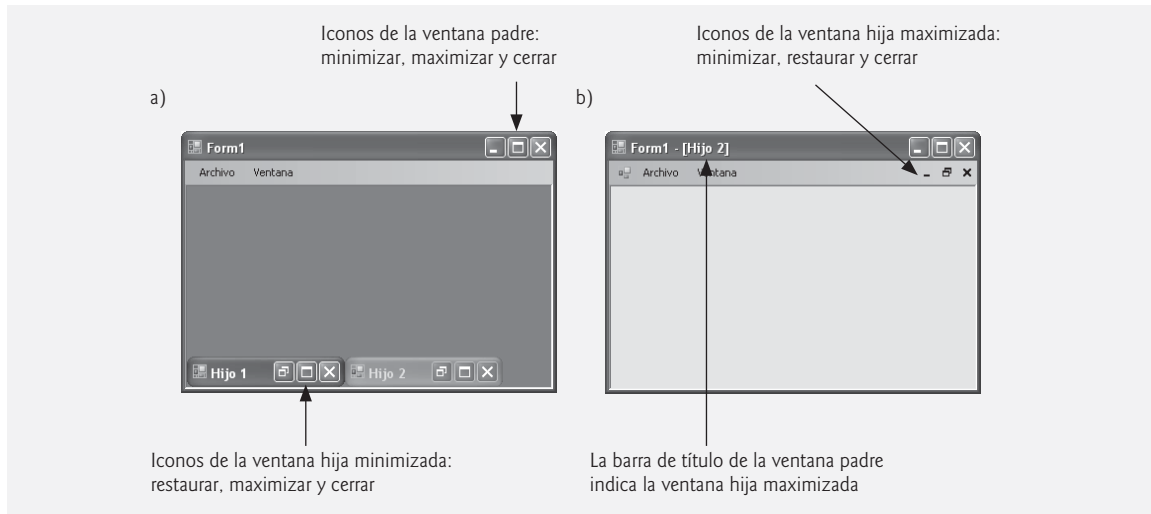


Figura 14.40 | Ventanas hijas minimizadas y maximizadas.

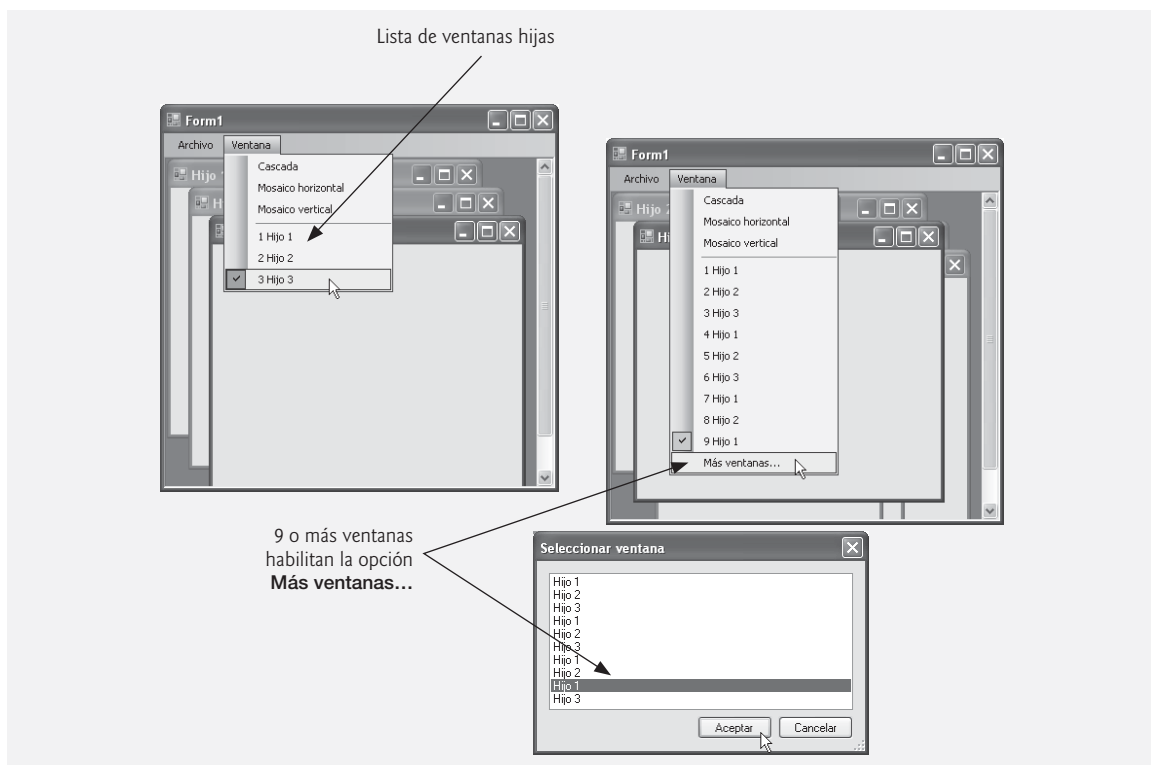


Figura 14.41 | Ejemplo de la propiedad **MdiList** de **MenuItem**.



Buena práctica de programación 14.1

Al crear aplicaciones MDI, incluya un menú que muestre una lista de las ventanas hijas abiertas. Esto ayuda al usuario a seleccionar una ventana hija con rapidez, en vez de tener que buscarla en la ventana padre.

Los contenedores MDI le permiten organizar la colocación de las ventanas hijas. Para ordenar las ventanas hijas en una aplicación MDI, se hace una llamada al método **LayoutMdi** del formulario padre. El método **LayoutMdi** recibe una enumeración **MdiLayout**, la cual puede tener los valores **ArrangeIcons**, **Cascade**, **TileHorizontal** y **TileVertical**. Las *ventanas en mosaico* llenan por completo la ventana padre y no se traslapan; dichas ventanas pueden organizarse en forma horizontal (el valor **TileHorizontal**) o vertical (el valor **TileVertical**). Las *ventanas en cascada* (el valor **Cascade**) se traslapan; cada una es del mismo tamaño y muestra una barra de título visible, si es posible. El valor **ArrangeIcons** ordena los iconos de todas las ventanas hijas minimizadas. Si hay ventanas minimizadas esparcidas alrededor de la ventana padre, el valor **ArrangeIcons** las ordena cerca de la esquina inferior izquierda de la ventana padre. La figura 14.42 ilustra los valores de la enumeración **MdiLayout**.

La clase **UsodeMDIForm** (figura 14.43) demuestra el uso de las ventanas MDI. La clase **UsodeMDIForm** tiene tres instancias del formulario hijo **HijoForm** (figura 14.44), cada una de las cuales contiene un control **PictureBox** que muestra una imagen. El formulario MDI padre contiene un menú que permite a los usuarios crear y ordenar los formularios hijos.

El programa en la figura 14.43 es la aplicación. El formulario MDI padre, que se crea primero, contiene dos menús de nivel superior. El primero de estos menús, llamado **Archivo** (**archivoToolStripMenuItem**), contiene un elemento **Salir** (**salirToolStripMenuItem**) y un submenú **Nuevo** (**nuevoToolStripMenuItem**), que

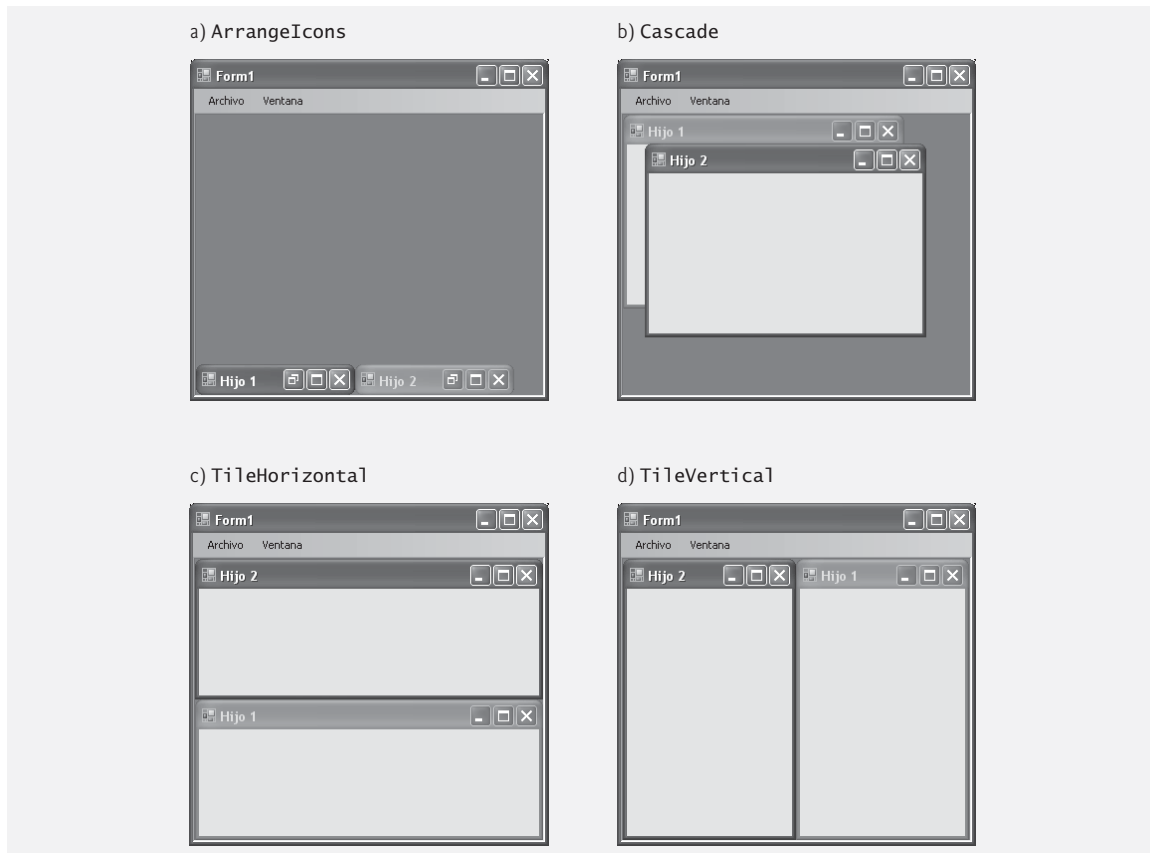


Figura 14.42 | Valores de la enumeración **MdiLayout**.

contiene los elementos para cada ventana hija. El segundo menú, llamado **Ventana** (`ventanaToolStripMenuItem`), proporciona opciones para distribuir en pantalla los formularios MDI hijos, además de una lista de los hijos MDI activos.

```

1  // Fig. 14.43: UsoDeMDIForm.cs
2  // Demostración del uso de ventanas MDI padre e hijas.
3  using System;
4  using System.Windows.Forms;
5
6  // el formulario demuestra el uso de las ventanas MDI padre e hijas
7  public partial class UsoDeMDIForm : Form
8  {
9      // constructor predeterminado
10     public UsoDeMDIForm()
11     {
12         InitializeComponent();
13     } // fin del constructor
14
15     // crea ventana Hijo 1 cuando se hace clic en hijo1ToolStripMenuItem
16     private void hijo1ToolStripMenuItem_Click(
17         object sender, EventArgs e )
18     {
19         // crea nuevo hijo
20         HijoForm formHijo = new HijoForm(
21             "Hijo 1", @"\\imagenes\\csharphttp1.jpg" );
22         formHijo.MdiParent = this; // set parent
23         formHijo.Show(); // display child
24     } // fin del método hijo1ToolStripMenuItem_Click
25
26     // crea ventana Hijo 2 cuando se hace clic en hijo2ToolStripMenuItem
27     private void hijo2ToolStripMenuItem_Click(
28         object sender, EventArgs e )
29     {
30         // crea nuevo hijo
31         HijoForm formHijo = new HijoForm(
32             "Hijo 2", @"\\imagenes\\vbnethttp2.jpg" );
33         formHijo.MdiParent = this; // establece el padre
34         formHijo.Show(); // muestra el hijo
35     } // fin del método hijo2ToolStripMenuItem_Click
36
37     // crea ventana Hijo 3 cuando se hace clic en hijo3ToolStripMenuItem
38     private void hijo3ToolStripMenuItem_Click(
39         object sender, EventArgs e )
40     {
41         // crea nuevo hijo
42         HijoForm formHijo =
43             new HijoForm( "Hijo 3", @"\\imagenes\\pythonhttp1.jpg" );
44         formHijo.MdiParent = this; // establece el padre
45         formHijo.Show(); // muestra el hijo
46     } // fin del método hijo3MenuItem_Click
47
48     // sale de la aplicación
49     private void salirToolStripMenuItem_Click( object sender, EventArgs e )
50     {
51         Application.Exit();
52     } // fin del método salirToolStripMenuItem_Click

```

Figura 14.43 | Clase de ventana MDI padre. (Parte I de 2).

```

53
54 // establece esquema en Cascada
55 private void cascadaToolStripMenuItem_Click(
56     object sender, EventArgs e )
57 {
58     this.LayoutMdi( MdiLayout.Cascade );
59 } // fin del método cascadaToolStripMenuItem_Click
60
61 // establece esquema en Mosaico horizontal
62 private void mosaicoHorizontalToolStripMenuItem_Click(
63     object sender, EventArgs e )
64 {
65     this.LayoutMdi( MdiLayout.TileHorizontal );
66 } // fin del método mosaicoHorizontalToolStripMenuItem_Click
67
68 // establece esquema en Mosaico vertical
69 private void mosaicoVerticalToolStripMenuItem_Click(
70     object sender, EventArgs e )
71 {
72     this.LayoutMdi( MdiLayout.TileVertical );
73 } // fin del método mosaicoVerticalToolStripMenuItem_Click
74 } // fin de la clase UsoDeMDIForm

```



Figura 14.43 | Clase de ventana MDI padre. (Parte 2 de 2).

En la ventana **Propiedades** establecimos la propiedad `IsMdiContainer` del formulario a `true`, convirtiéndolo en un formulario MDI padre. Además, establecimos la propiedad `MdiWindowListItem` de `MenuStrip` a `ventanaToolStripMenuItem`. Esto permite que el menú **Ventana** contenga la lista de ventanas MDI hijas.

El elemento de menú **Cascada** (`cascadaToolStripMenuItem`) tiene un manejador de eventos (`cascadaToolStripMenuItem_Click`, líneas 55-59) que ordena las ventanas hijas en cascada. El manejador de eventos llama al método `LayoutMdi` con el argumento `Cascade` de la enumeración `MdiLayout` (línea 58).

El elemento de menú **Mosaico horizontal** (`mosaicoHorizontalToolStripMenuItem`) tiene un manejador de eventos (`mosaicoHorizontalToolStripMenuItem_Click`, líneas 62-66) que ordena las ventanas hijas en forma horizontal. El manejador de eventos llama al método `LayoutMdi` con el argumento `TileHorizontal` de la enumeración `MdiLayout` (línea 65).

Por último, el elemento de menú **Mosaico vertical** (`mosaicoVerticalToolStripMenuItem`) tiene un manejador de eventos (`mosaicoVerticalToolStripMenuItem_Click`, líneas 69-73) que ordena las ventanas hijas en forma vertical. El manejador de eventos llama al método `LayoutMdi` con el argumento `TileVertical` de la enumeración `MdiLayout` (línea 72).

En este punto, la aplicación aún está incompleta; debemos definir la clase de formulario MDI hijo. Para ello, haga clic con el botón derecho del ratón en el proyecto, dentro del **Explorador de soluciones**, y seleccione **Agregar > Windows Forms...** Después use el nombre `HijoForm` para la nueva clase en el cuadro de diálogo (figura 14.44). A continuación, hay que agregar un control `PictureBox` (`mostrarImag`) a `HijoForm`. En el constructor, la línea 15 establece el texto de la barra de título. Las líneas 18-19 establecen la propiedad `Image` de `HijoForm` a un objeto `Image`, usando el método `FromFile`.

Después de definir la clase de formulario MDI hijo, el formulario MDI padre (figura 14.43) puede crear nuevas ventanas hijas. Los manejadores de eventos en las líneas 16-46 crean un nuevo formulario hijo, que corresponde al elemento del menú en el que se hizo clic. Las líneas 20-21, 31-32 y 42-43 crean nuevas instancias de `HijoForm`. Las líneas 22, 33 y 44 establecen la propiedad `MdiParent` de cada ventana hija, para asignarle el formulario padre. Las líneas 23, 34 y 45 llaman al método `Show` para mostrar cada formulario hijo.

14.13 Herencia visual

En el capítulo 10 vimos cómo crear clases heredando de otras clases. También hemos utilizado la herencia para crear formularios que muestren una GUI, derivando nuestras nuevas clases de formularios de la clase `System.Windows.Forms.Form`. Éste es un ejemplo de *herencia visual*. La clase `Form` derivada contiene la funcionalidad de su clase base `Form`, incluyendo sus propiedades, métodos, variables y controles. La clase derivada también hereda de su clase base todos los aspectos visuales: tamaño, distribución de los componentes, espacio entre los componentes de la GUI, colores y fuentes.

```

1  // Fig. 14.44: HijoForm.cs
2  // Ventana hija del padre MDI.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6  using System.IO;
7
8  public partial class HijoForm : Form
9  {
10     public HijoForm( string titulo, string nombreArchivo )
11     {
12         // Requerido para el soporte del Diseñador de Windows Form
13         InitializeComponent();
14
15         Text = titulo; // establece el texto del título
16
17         // establece imagen a mostrar en control PictureBox
18         mostrarImag.Image = Image.FromFile(
19             Directory.GetCurrentDirectory() + nombreArchivo );
20     } // fin del constructor
21 } // fin de la clase HijoForm

```

Figura 14.44 | El formulario MDI hijo `HijoForm`.

La herencia visual le permite lograr una consistencia visual en todas las aplicaciones. Por ejemplo, podría definir un formulario base que contenga el logotipo de un producto, un color de fondo específico, una barra de menús predefinida y otros elementos. Después podría usar el formulario base a lo largo de toda una aplicación para obtener uniformidad y reconocimiento de marca.

La clase `HerenciaVisualForm` (figura 14.45) se deriva de `Form`. Los resultados ilustran el funcionamiento del programa. La GUI contiene dos controles `Label` con el texto, **Errores, Errores, Errores** y **Copyright 2006, por deitel.com**, así como un control `Button` que muestra el texto **Aprenda más**. Cuando un usuario oprime el botón **Aprenda más**, se invoca el método `aprendaMasButton_Click` (líneas 16-22). Este método muestra un cuadro de diálogo `MessageBox` que proporciona cierto texto informativo.

Para permitir que otros formularios hereden de `HerenciaVisualForm`, debemos empaquetar a `Visual-HerenciaForm` como un archivo `.dll` (biblioteca de clases). Haga clic con el botón derecho del ratón en el nombre del proyecto, dentro del **Explorador de soluciones**, y seleccione **Propiedades**; después seleccione la ficha **Aplicación**. En la lista desplegable **Tipo de resultado**, sustituya **Aplicación para Windows** por **Biblioteca de clases**. Al generar el proyecto se produce el archivo `.dll`.

Para heredar en forma visual de `HerenciaVisualForm`, primero hay que crear una aplicación para Windows. En esta aplicación, agregue una referencia al archivo `.dll` que acaba de crear (se encuentra en la carpeta `bin/Release` de la aplicación anterior). Después abra el archivo que define la GUI de la nueva aplicación y modifique la primera línea de la clase, de manera que herede de la clase `HerenciaVisualForm`. Sólo tendrá que especificar

```

1 // Fig. 14.45: HerenciaVisualForm.cs
2 // Formulario base para usarlo con la herencia visual.
3 using System;
4 using System.Windows.Forms;
5
6 // el formulario base se utiliza para demostrar la herencia visual
7 public partial class HerenciaVisualForm : Form
8 {
9     // constructor predeterminado
10    public HerenciaVisualForm()
11    {
12        InitializeComponent();
13    } // fin del constructor
14
15    // muestra MessageBox cuando se hace clic en el control Button
16    private void aprendaMasButton_Click( object sender, EventArgs e )
17    {
18        MessageBox.Show(
19            "Errores, Errores, Errores es un producto de deitel.com",
20            "Aprenda más", MessageBoxButtons.OK,
21            MessageBoxIcon.Information );
22    } // fin del método aprendaMasButton_Click
23 } // fin de la clase HerenciaVisualForm

```

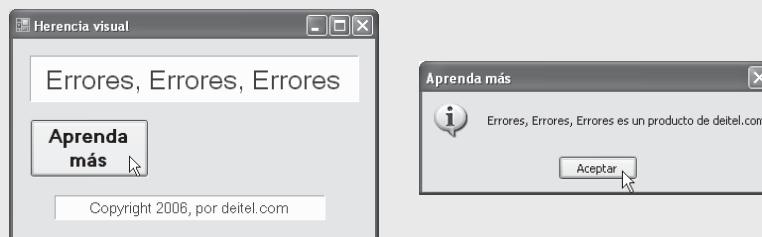


Figura 14.45 | La clase `HerenciaVisualForm`, que hereda de la clase `Form`, contiene un control `Button` (**Aprenda más**).

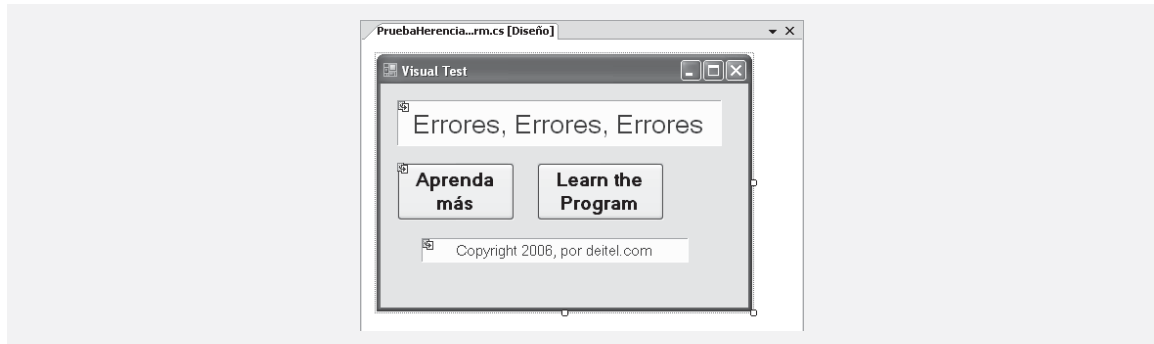


Figura 14.46 | Formulario que demuestra la herencia visual.

el nombre de la clase. En la vista de **Diseño**, el formulario de la nueva aplicación deberá mostrar los controles del formulario base (figura 14.46). Aún podemos agregar más componentes al formulario.

La clase `PruebaHerenciaVisualForm` (figura 14.47) es una clase derivada de `HerenciaVisualForm`. Los resultados ilustran la funcionalidad del programa. La GUI contiene los componentes derivados de la clase `HerenciaVisualForm`, así como también un control `Button` adicional, con el texto **Aprenda el programa**. Cuando un usuario oprime este control `Button`, se invoca el método `aprendaProgramaButton_Click` (líneas 17-22). Este método muestra otro cuadro de diálogo `MessageBox` que proporciona un texto informativo distinto.

La figura 14.47 demuestra que la clase `PruebaHerenciaVisual` hereda los componentes, la distribución de los mismos y la funcionalidad de la clase base `HerenciaVisualForm` (figura 14.45). Si un usuario hace clic en el botón **Aprenda más**, el manejador de eventos `aprendaMasButton_Click` de la clase base muestra un cuadro de diálogo `MessageBox`. `HerenciaVisualForm` utiliza un modificador de acceso `private` para declarar sus controles, por lo que la clase `PruebaHerenciaVisualForm` no puede modificar los controles heredados de la clase `HerenciaVisualForm`.

```

1  // Fig. 14.47: PruebaHerenciaVisualForm.cs
2  // Formulario derivado mediante el uso de la herencia visual.
3  using System;
4  using System.Windows.Forms;
5
6  // formulario derivado mediante la herencia visual
7  public partial class PruebaHerenciaVisualForm :
8      HerenciaVisualForm // código para la herencia
9  {
10     // constructor predeterminado
11     public PruebaHerenciaVisualForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // muestra MessageBox cuando se hace clic en el control Button
17     private void aprendaProgramaButton_Click(object sender, EventArgs e)
18     {
19         MessageBox.Show( "Este programa fue creado por Deitel & Associates",
20                         "Aprenda el programa", MessageBoxButtons.OK,
21                         MessageBoxIcon.Information );
22     } // fin del método aprendaProgramaButton_Click
23 } // fin de la clase PruebaHerenciaVisualForm

```

Figura 14.47 | La clase `PruebaHerenciaVisualForm`, que hereda de la clase `HerenciaVisualForm`, contiene un control `Button` adicional. (Parte 1 de 2).

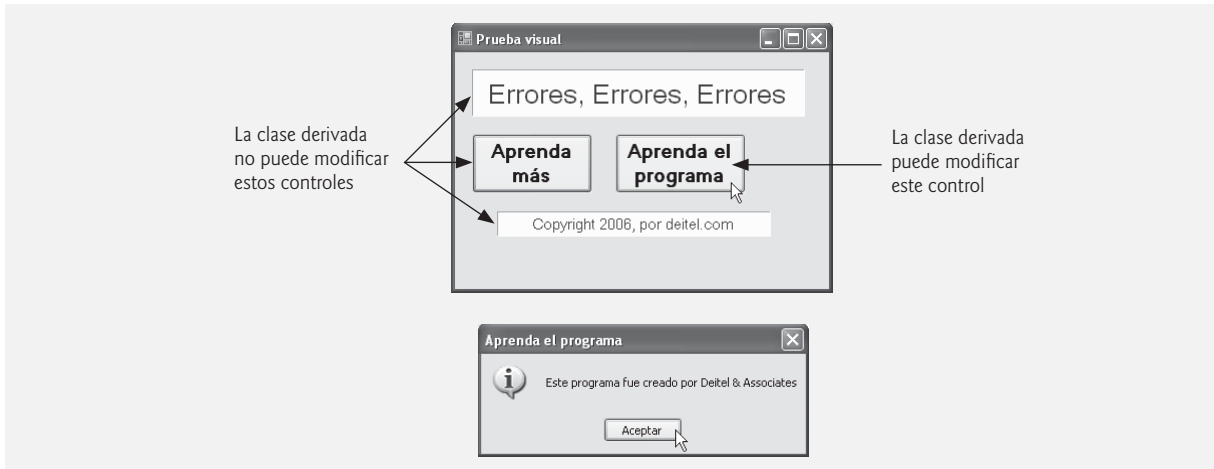


Figura 14.47 | La clase `PruebaHerenciaVisualForm`, que hereda de la clase `HerenciaVisualForm`, contiene un control `Button` adicional. (Parte 2 de 2).

14.14 Controles definidos por el usuario

El .NET Framework le permite crear *controles personalizados*. Estos controles personalizados aparecen en el **Cuadro de herramientas** del usuario y pueden agregarse a controles `Form`, `Panel` o `GroupBox` de la misma forma que se agregan los controles `Button`, `Label` y otros controles predefinidos. La manera más simple de crear un control personalizado es derivar una clase de un control existente, como `Label`. Esto es útil si queremos agregar funcionalidad a un control existente, en vez de tener que volver a implementar el control existente para incluir la funcionalidad deseada. Por ejemplo, puede crear un nuevo tipo de control `Label` que se comporte como un control `Label` normal, pero que tenga una apariencia distinta. Para ello, hay que heredar de la clase `Label` y redefinir el método `OnPaint`.

Todos los controles contienen el método **`OnPaint`**, que el sistema llama cuando un componente debe volver a dibujarse (como cuando se cambia su tamaño). El método `OnPaint` recibe un objeto **`PaintEventArgs`**, el cual contiene información gráfica: la propiedad **`Graphics`** es el objeto gráfico que se utiliza para dibujar, y la propiedad **`ClipRectangle`** define el límite rectangular del control. Cada vez que el sistema produce el evento `Paint`, la clase base de nuestro control atrapa el evento. Mediante el polimorfismo se hace una llamada al método `OnPaint` de nuestro control. Como no se hace la llamada a la implementación de `OnPaint` de nuestra clase base, debemos de llamarla en forma implícita desde nuestra implementación de `OnPaint` antes de ejecutar nuestro código personalizado para pintar en la pantalla. En la mayoría de los casos, es conveniente hacer esto para asegurar que se ejecute el código original para pintar en la pantalla, además del código que usted defina en la clase del control personalizado. Por otro lado, si no deseamos que se ejecute el método `OnPaint` de la clase base, no lo llamamos.

Para crear un nuevo control compuesto de controles existentes, use la clase **`UserControl`**. Los controles que se agregan a un control personalizado se llaman *controles constituyentes*. Por ejemplo, un programador podría crear un control `UserControl` compuesto de un control `Button`, un control `Label` y un control `TextBox`, cada uno de ellos asociado con cierta funcionalidad (por ejemplo, que el control `Button` establezca el texto del control `Label` y le asigne el texto que contiene el control `TextBox`). El control `UserControl` actúa como un contenedor para los controles que se agregan a él. El control `UserControl` contiene controles constituyentes, por lo que no determina la forma en que éstos se muestran en pantalla. El método `OnPaint` del control `UserControl` no puede redefinirse. Para controlar la apariencia de cada control constituyente, debe manejar el evento `Paint` de cada control. El manejador de eventos de `Paint` recibe un objeto `PaintEventArgs`, el cual puede usarse para dibujar gráficos (líneas, rectángulos, etc.) en los controles constituyentes.

Mediante el uso de otra técnica, un programador puede crear un control completamente nuevo, heredando de la clase `Control`. Esta clase no define ningún comportamiento específico; esa tarea se deja para el programador. La clase `Control` maneja los elementos asociados con todos los controles, como los eventos y los controles de tamaño. El método `OnPaint` debe contener una llamada al método `OnPaint` de la clase base, el cual debe llamar

a los manejadores de eventos de `Paint`. Después, usted debe agregar código para dibujar gráficos personalizados dentro del método `OnPaint` redefinido, a la hora de dibujar el control. Esta técnica permite la mayor flexibilidad, pero también requiere la mayor planeación. En la figura 14.48 se muestra un resumen de los tres métodos.

Técnicas para controles personalizado y propiedades de <code>PaintEventArgs</code>	
	Descripción
<i>Técnicas para controles personalizados</i>	
Heredar del control de Windows Forms	Puede hacer esto para agregar funcionalidad a un control ya existente. Si redefine el método <code>OnPaint</code> , llame al método <code>OnPaint</code> de la clase base. Sólo puede agregar elementos a la apariencia del control original, no rediseñarlo.
Crear un control <code>UserControl</code>	Puede crear un control <code>UserControl</code> compuesto de varios controles ya existentes (por ejemplo, para combinar su funcionalidad). No puede redefinir los métodos <code>OnPaint</code> de los controles personalizados. En vez de ello, debe colocar el código de dibujo en un manejador de eventos <code>Paint</code> . De nuevo, hay que tener en cuenta que sólo se pueden agregar elementos a la apariencia original del control, no rediseñarlo.
Heredar de la clase <code>Control</code>	Defina un control completamente nuevo. Redefina el método <code>OnPaint</code> y después llame al método <code>OnPaint</code> de la clase base e incluya métodos para dibujar el control. Con este método puede personalizar la apariencia y la funcionalidad del control.
<i>Propiedades de <code>PaintEventArgs</code></i>	
<code>Graphics</code>	El objeto gráfico del control. Se utiliza para dibujar en el control.
<code>ClipRectangle</code>	Especifica el rectángulo que indica el contorno del control.

Figura 14.48 | Creación de controles personalizados.

En la figura 14.49 creamos un control tipo “reloj”. Éste es un control `UserControl` compuesto de un control `Label` y un control `Timer`; cada vez que el control `Timer` genera un evento, el control `Label` se actualiza para reflejar la hora actual.

Los controles **Timer** (espacio de nombres `System.Windows.Forms`) son componentes invisibles que residen en un formulario, generando eventos **Tick** a un intervalo establecido. Este intervalo se establece mediante la propiedad **Interval** del control `Timer`, la cual define el número de milisegundos (milésimas de un segundo) que deben transcurrir entre cada evento. De manera predeterminada, los temporizadores están deshabilitados y no generan eventos.

Esta aplicación contiene un control de usuario (`RelojUserControl`) y un formulario que muestra el control de usuario. Empezamos por crear una aplicación para Windows. Después, creamos una clase `UserControl` para el proyecto, seleccionando **Proyecto > Agregar control de usuario....** A continuación debe aparecer un cuadro de diálogo, en el que podemos seleccionar el tipo de control que deseamos agregar; ya están seleccionados los controles del usuario. Después asignamos el nombre `RelojUserControl` al archivo (y a la clase). Nuestro control `RelojUserControl` se muestra como un rectángulo de color gris.

Puede tratar a este control como un formulario Windows Forms, lo que significa que puede agregar controles mediante el **Cuadro de herramientas** y establecer propiedades mediante la ventana **Propiedades**. No obstante, en vez de crear una aplicación, sólo estamos creando un nuevo control compuesto de otros controles. Agregue un control `Label` (`mostrarLabel`) y un control `Timer` (`relojTimer`) al control `UserControl`. Establezca el intervalo del control `Timer` a 1000 milisegundos y establezca el texto de `mostrarLabel` con cada evento (líneas 16-20). Para generar eventos, `relojTimer` debe habilitarse estableciendo la propiedad `Enabled` a `true` en la ventana **Propiedades**.

```

1 // Fig. 14.49: RelojUserControl.cs
2 // Control definido por el usuario, con un control Timer y un control Label.
3 using System;
4 using System.Windows.Forms;
5
6 // control UserControl que muestra la hora en un control Label
7 public partial class RelojUserControl : UserControl
8 {
9     // constructor predeterminado
10    public RelojUserControl()
11    {
12        InitializeComponent();
13    } // fin del constructor
14
15    // actualiza el control Label en cada evento Tick
16    private void relojTimer_Tick(object sender, EventArgs e)
17    {
18        // obtiene la hora actual (Now), la convierte en string
19        mostrarLabel.Text = DateTime.Now.ToLongTimeString();
20    } // fin del método relojTimer_Tick
21 } // fin de la clase RelojUserControl

```

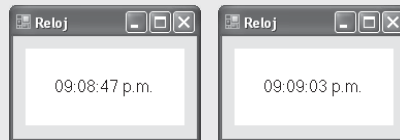


Figura 14.49 | Reloj definido mediante el control UserControl.

La estructura **DateTime** (espacio de nombres System) contiene la propiedad **Now**, que es la hora actual. El método **ToLongTimeString** convierte Now en un objeto string que contiene la hora, los minutos y segundos actuales (junto con el indicador AM o PM). Utilizamos esta información para establecer la hora en **mostrarLabel**, en la línea 19.

Una vez creado, nuestro control tipo reloj aparece como un elemento en el **Cuadro de herramientas**. Tal vez necesite cambiar al formulario de la aplicación para que el elemento aparezca en el **Cuadro de herramientas**. Para usar el control, sólo arrástrelo al formulario y ejecute la aplicación para Windows. Usamos un color de fondo blanco para el objeto **RelojUserControl**, para hacerlo que resalte en el formulario. La figura 14.49 muestra el resultado de **RelojForm**, que contiene nuestro control **RelojUserControl**. No hay manejadores de eventos en **RelojForm**, por lo que sólo mostramos el código para **RelojUserControl**.

Visual Studio le permite compartir controles personalizados con otros desarrolladores. Para crear un control **UserControl** que pueda exportarse a otras soluciones, haga lo siguiente:

1. Cree un nuevo proyecto de **Biblioteca de clases**.
2. Elimine el archivo **Class1.cs** que se incluye al principio con la aplicación.
3. Haga clic con el botón derecho del ratón en el proyecto, dentro del **Explorador de soluciones**, y seleccione **Agregar > Control de usuario....** En el cuadro de diálogo que aparezca, escriba un nombre para el archivo del control de usuario y haga clic en **Agregar**.
4. Dentro del proyecto, agregue controles y funcionalidad al control **UserControl** (figura 14.50).
5. Genere el proyecto. Visual Studio creará un archivo **.dll** para el control **UserControl** en el directorio de salida (**bin/Release**). El archivo no es ejecutable; las bibliotecas de clases se utilizan para definir clases que pueden reutilizarse en otras aplicaciones ejecutables.
6. Cree una nueva aplicación para Windows.

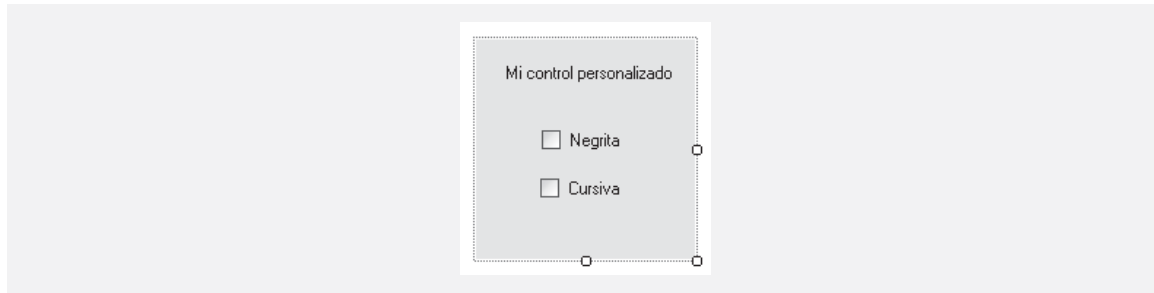


Figura 14.50 | Creación de un control personalizado.

- En la nueva aplicación para Windows, haga clic con el botón derecho del ratón en el **Cuadro de herramientas** y seleccione la opción **Elegir elementos....** En el cuadro de diálogo **Elegir elementos del cuadro de herramientas** que aparezca, haga clic en **Examinar....** Busque el archivo .dll de la biblioteca de clases que creó en los *Pasos del 1 al 5*. Después aparecerá el elemento en el cuadro de diálogo **Elegir elementos del cuadro de herramientas** (figura 14.51). Ahora este control se puede agregar al formulario, como si fuera cualquier otro control (figura 14.52).

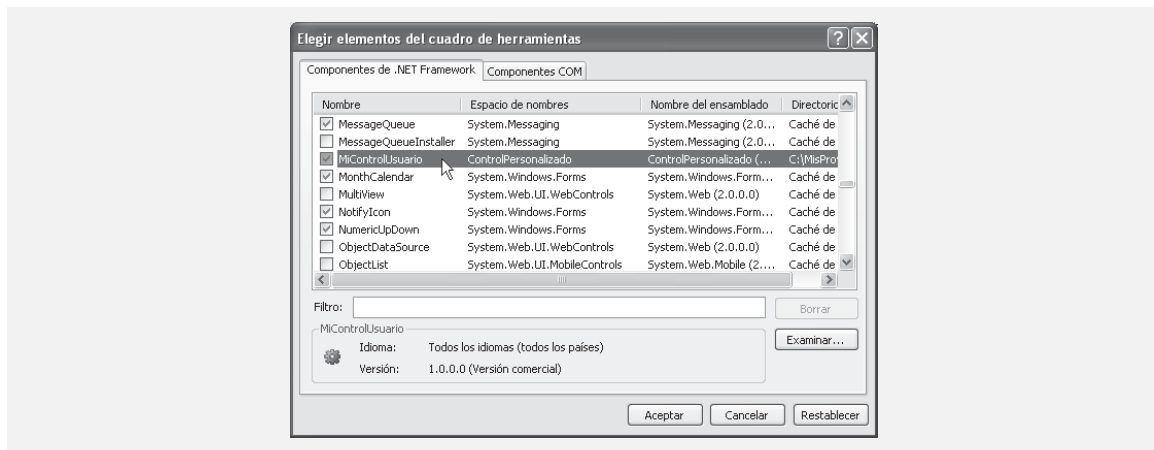


Figura 14.51 | Control personalizado agregado al **Cuadro de herramientas**.

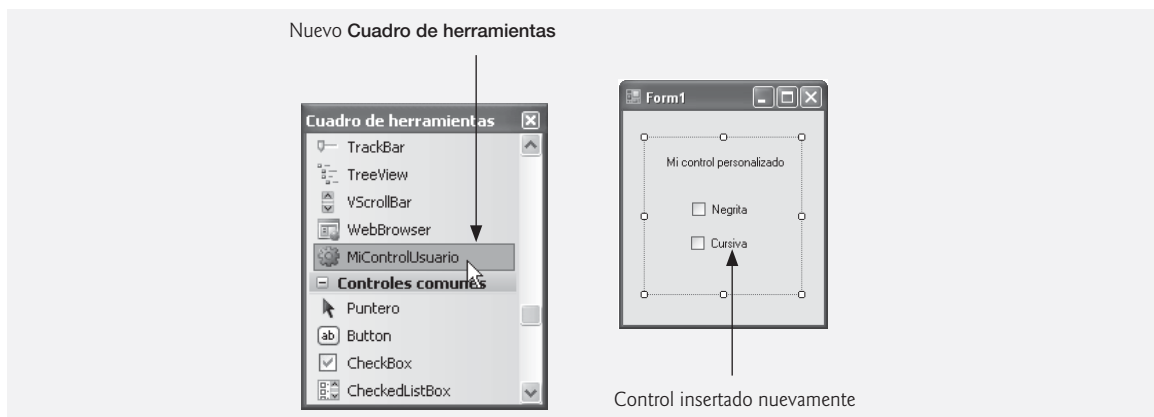


Figura 14.52 | Control personalizado agregado al formulario.

14.15 Conclusión

Muchas de las aplicaciones comerciales de hoy en día cuentan con GUIs que son fáciles de usar y manipular. Debido a esta demanda de GUIs amigables para el usuario, la habilidad de diseñar GUIs sofisticadas es una habilidad de programación esencial. El IDE de Visual Studio ayuda a que el desarrollo de GUIs sea rápido y sencillo. En los capítulos 13 y 14 presentamos técnicas de desarrollo de GUIs básicas. En el capítulo 14 demostramos cómo crear menús, los cuales proporcionan a los usuarios un fácil acceso a la funcionalidad de una aplicación. En este capítulo aprendió a usar los controles `DateTimePicker` y `MonthCalendar`, que permiten a los usuarios introducir valores de fecha y hora. Demostramos los controles `LinkLabel`, que se utilizan para vincular al usuario con una aplicación o página Web. Utilizó varios controles que proporcionan listas de datos al usuario: `ListBox`, `CheckedListBox` y `ListView`. Utilizamos el control `ComboBox` para crear listas desplegables, y el control `TreeView` para mostrar datos en forma jerárquica. Después presentamos GUIs complejas que utilizan ventanas con fichas e interfaces de múltiples documentos. Concluimos este capítulo con demostraciones de la herencia simple y la creación de controles personalizados.

En el siguiente capítulo exploraremos el concepto de subprocesamiento múltiple. En muchos lenguajes de programación puede crear varios subprocesos, para que varias actividades se lleven a cabo en paralelo.

15

Subprocesamiento múltiple

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Qué son los subprocesos y por qué son útiles.
- Cómo nos permiten los subprocesos administrar actividades concurrentes.
- El ciclo de vida de un subproceso.
- Las prioridades y la programación de subprocesos.
- A crear y ejecutar objetos Thread.
- Acerca de la sincronización de subprocesos.
- Qué son las relaciones productor/consumidor y cómo se implementan con el subprocesamiento múltiple.
- A mostrar resultados de múltiples subprocesos en una GUI.

*No bloquee el camino
de la investigación*

—Charles Sanders Peirce

*Una persona con un reloj
sabe qué hora es;
una persona con dos relojes
nunca estará segura.*

—Proverbio

Aprenda a laborar y esperar.

—Henry Wadsworth Longfellow

*La definición más general
de la belleza... múltiple deidad
en la unidad.*

—Samuel Taylor Coleridge

*El mundo avanza tan rápido
estos días que el hombre que
dice que no puede hacer algo, por
lo general se ve interrumpido
por alguien que lo está haciendo.*

—Elbert Hubbard

- 15.1 Introducción
- 15.2 Estados de los subprocesos: ciclo de vida de un subproceso
- 15.3 Prioridades y programación de subprocesos
- 15.4 Creación y ejecución de subprocesos
- 15.5 Sincronización de subprocesos y la clase `Monitor`
- 15.6 Relación productor/consumidor sin sincronización de subprocesos
- 15.7 Relación productor/consumidor con sincronización de subprocesos
- 15.8 Relación productor/consumidor: búfer circular
- 15.9 Subprocesamiento múltiple con GUIs
- 15.10 Conclusión

15.1 Introducción

Sería bueno si pudiéramos realizar una acción a la vez, y realizarla bien, pero por lo general eso es difícil. El cuerpo humano realiza una gran variedad de operaciones *en paralelo*; o, como diremos en este capítulo, *operaciones concurrentes*. Por ejemplo, la respiración, la circulación de la sangre y la digestión pueden ocurrir de manera concurrente. Todos los sentidos (vista, tacto, olfato, gusto y oído) pueden emplearse al mismo tiempo. Las computadoras también pueden realizar operaciones concurrentes. Para su computadora es tarea común compilar un programa, enviar un archivo a una impresora y recibir mensajes de correo electrónico a través de una red, de manera concurrente.

Aunque suene irónico, la mayoría de los lenguajes de programación no permiten a los programadores especificar actividades concurrentes. Por lo general, los lenguajes de programación proporcionan sólo un simple conjunto de instrucciones de control, las cuales permiten a los programadores realizar una acción a la vez, procediendo a la siguiente acción una vez que la anterior haya terminado. Históricamente, el tipo de concurrencia que las computadoras realizan en la actualidad se ha implementado en forma general como funciones “primitivas” del sistema operativo, disponibles sólo para los “programadores de sistemas” altamente experimentados.

El lenguaje de programación Ada, desarrollado por el Departamento de Defensa de los Estados Unidos, hizo que las primitivas de concurrencia estuvieran disponibles ampliamente para los contratistas de defensa dedicados a la construcción de sistemas militares de comando y control. Sin embargo, Ada no se ha utilizado de manera general en las universidades o en la industria comercial.

La Biblioteca de clases del .NET Framework cuenta con primitivas de concurrencia. Usted especifica que las aplicaciones contienen “*subprocesos de ejecución*”, cada uno de los cuales designa una porción de un programa que puede ejecutarse en forma concurrente con otros subprocesos; a esta capacidad se le conoce como subprocesamiento múltiple, y está disponible para todos los lenguajes de programación .NET, incluyendo C#, Visual Basic y Visual C++. La Biblioteca de clases del .NET Framework incluye herramientas de subprocesamiento múltiple en el espacio de nombres **System.Threading**.



Tip de rendimiento 15.1

Un problema con las aplicaciones de un solo subproceso es que las actividades que llevan mucho tiempo deben completarse antes de que puedan comenzar otras. En una aplicación con subprocesamiento múltiple, los subprocesos pueden distribuirse entre varios procesadores (si hay disponibles), de manera que se realicen varias tareas en forma concurrente, lo cual permite a la aplicación operar con más eficiencia. El subprocesamiento múltiple también puede incrementar el rendimiento en sistemas con un solo procesador que simulan la concurrencia; cuando un subproceso no puede proceder, otro puede usar el procesador.

Hablaremos sobre muchas aplicaciones de la *programación concurrente*. Cuando los programas descargan archivos extensos como clips de audio o video desde Internet, los usuarios no desean esperar hasta que se descargue todo un clip completo para empezar a reproducirlo. Para resolver este problema, podemos poner varios subprocesos a trabajar; un subproceso descarga un clip mientras otro lo reproduce. Estas actividades se realizan en forma concurrente para evitar que la reproducción del clip tenga interrupciones, *sincronizamos* los subprocesos

de manera que el subproceso de reproducción no empiece sino hasta que haya una cantidad suficiente del clip en memoria, como para mantener ocupado al subproceso de reproducción.

Otro ejemplo de subprocesamiento múltiple es la recolección automática de basura del CLR. C y C++ requieren que los programadores reclamen en forma explícita la memoria asignada de manera dinámica. El CLR cuenta con un subproceso recolector de basura, el cual reclama la memoria asignada en forma dinámica que ya no se necesita.



Buena práctica de programación 15.1

Establezca a null la referencia a un objeto, cuando el programa ya no necesite ese objeto. Esto permite que el recolector de basura determine lo más pronto posible si el objeto puede recolectarse como basura. Si dicho objeto tiene otras referencias hacia él, no puede recolectarse.

Escribir programas con subprocesamiento múltiple puede ser un proceso complicado. Aunque la mente humana puede realizar funciones de manera concurrente, a las personas se les dificulta saltar de un “tren de pensamiento” paralelo a otro. Para ver por qué el subprocesamiento múltiple puede ser difícil de programar y comprender, intente realizar el siguiente experimento: abra tres libros en la página 1 y trate de leerlos concurrentemente. Lea unas cuantas palabras del primer libro; después unas cuantas del segundo y por último unas cuantas del tercero, luego regrese a leer las siguientes palabras del primer libro, etc. Después de este experimento apreciará los retos que implica el subprocesamiento múltiple: alternar entre los libros, leer brevemente, recordar en dónde se quedó en cada libro, acercarse al libro que está leyendo para poder verlo, hacer a un lado los libros que no está leyendo y, entre todo este caos, ¡tratar de comprender el contenido de cada libro!

15.2 Estados de los subprocesos: ciclo de vida de un subproceso

En cualquier momento dado, se dice que un subproceso se encuentra en uno de varios estados de subproceso, los cuales se muestran en el diagrama de estados de UML en la figura 15.1. En esta sección hablaremos sobre estos estados y las transiciones entre los mismos. Dos clases imprescindibles para las aplicaciones con subprocesamiento múltiple son **Thread** y **Monitor** (espacio de nombres System.Threading). En esta sección hablaremos también sobre varios métodos de las clases Thread y Monitor que producen transiciones de estado. En las siguientes secciones hablaremos sobre algunos de los términos en el diagrama.

Un objeto Thread empieza su ciclo de vida en el estado **Unstarted** (inactivo) cuando el programa crea el objeto y pasa un delegado **ThreadStart** al constructor del objeto. El delegado ThreadStart, que especifica la acción que realizará el subproceso durante su ciclo de vida, debe inicializarse con un método que devuelva void y no reciba argumentos. [Nota: .NET 2.0 también incluye un delegado ParameterizedThreadStart, al cual le puede pasar un método que reciba argumentos. Para obtener más información, visite los sitios msdn2.microsoft.com/en-us/library/xzehzsds (en inglés) y msdn2.microsoft.com/es-es/library/system.threading.parameterizedthreadstart(VS.80).aspx (en español).] El subproceso permanece en el estado **Unstarted** hasta que el programa llama al método **Start** de Thread, el cual coloca el subproceso en el estado **Running** y devuelve de inmediato el control a la parte del programa que llamó a Start. Después, el subproceso que acaba de pasar al estado **Running** y cualquier otro subproceso en el programa se pueden ejecutar en forma concurrente en un sistema con varios procesadores, o pueden compartir el procesador en un sistema con un solo procesador.

Mientras se encuentre en el estado **Running** (en ejecución), tal vez el subproceso no esté en ejecución todo el tiempo. El subproceso se ejecuta en el estado **Running** sólo cuando el sistema operativo asigna un procesador al subproceso. Cuando un subproceso en ejecución recibe un procesador por primera vez, empieza a ejecutar el método especificado por su delegado ThreadStart.

Un subproceso en ejecución entra al estado **Stopped** (detenido) o **Aborted** (abortado) cuando termina su delegado ThreadStart, lo que por lo general indica que el subproceso ha completado su tarea. Observe que un programa puede forzar a un subproceso a que entre en el estado **Stopped**, llamando al método **Abort** de Thread en el objeto Thread apropiado. El método Abort lanza una excepción **ThreadAbortException** en el subproceso, lo que por lo general ocasiona que el subproceso termine. Cuando un subproceso se encuentra en el estado **Stopped** y no hay referencias al objeto subproceso, el recolector de basura puede eliminar el objeto subproceso de la memoria. [Nota: Cuando se llama internamente al método Abort de un subproceso, éste entra en el estado

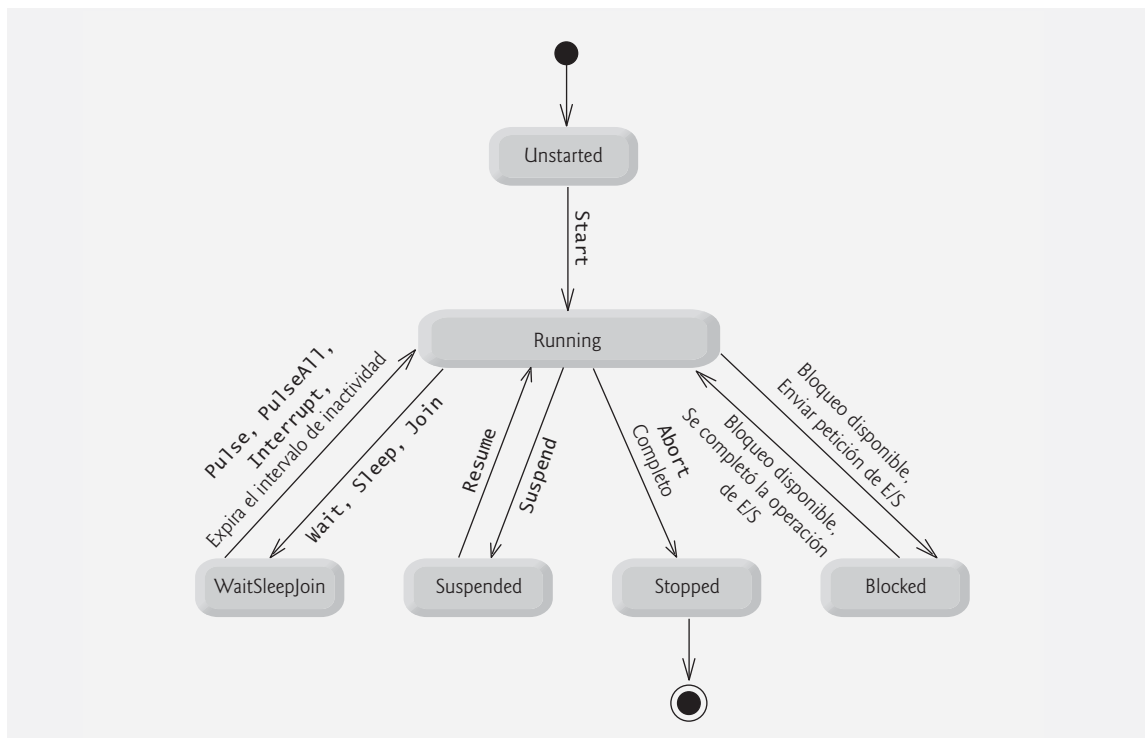


Figura 15.1 | Ciclo de vida de un subproceso.

AbortRequested antes de entrar al estado *Stopped*. El subproceso permanece en el estado *AbortRequested* mientras espera recibir la excepción `ThreadAbortException` que está pendiente. Cuando se hace una llamada a `Abort`, si el subproceso se encuentra en el estado *WaitSleepJoin*, *Suspended* o *Blocked*, el subproceso reside en su estado actual y en el estado *AbortRequested*, por lo que no puede recibir la excepción `ThreadAbortException` sino hasta que deje su estado actual.]

Un subproceso se considera **bloqueado** si no puede utilizar un procesador, aun si hay uno disponible. Por ejemplo, un subproceso se bloquea cuando envía una petición de entrada/salida (E/S). El sistema operativo bloquea la ejecución del subproceso hasta que pueda completarse la petición de E/S por la que el subproceso está esperando. En ese punto, el subproceso regresa al estado *Running*, para poder seguir ejecutándose. Otro caso en el cual se bloquea un subproceso es en la sincronización de subprocesos (sección 15.5). Un subproceso que se sincronizará debe adquirir un bloqueo sobre un objeto, mediante una llamada al método **Enter** de la clase `Monitor`. Si no hay un bloqueo disponible, el subproceso se bloquea hasta que esté disponible el bloqueo deseado. [Nota: el estado *Blocked* en realidad no es un estado en .NET. Es un estado conceptual que describe a un subproceso que no se encuentra en el estado *Running*.]

Hay tres formas en las que un subproceso en el estado *Running* entra al estado *WaitSleepJoin*. Si un subproceso encuentra código que no puede ejecutar todavía (por lo general, debido a que no se cumple cierta condición), puede llamar al método **Wait** de `Monitor` para entrar al estado *WaitSleepJoin*. Una vez en este estado, un subproceso regresa al estado *Running* cuando otro subproceso invoca al método **Pulse** o **PulseAll** de `Monitor`. El método `Pulse` cambia el siguiente subproceso en espera, de vuelta al estado *Running*. El método `PulseAll` cambia todos los subprocesos en espera, de vuelta al estado *Running*.

Un subproceso en ejecución puede llamar al método **Sleep** de `Thread` para entrar al estado *WaitSleepJoin* durante un periodo de varios milisegundos, que se especifican como el argumento para `Sleep`. Un subproceso inactivo regresa al estado *Running* cuando expira su tiempo de inactividad designado. Los subprocesos inactivos no pueden usar un procesador, aunque haya uno disponible.

Cualquier subproceso que entra al estado *WaitSleepJoin* mediante una llamada al método `Wait` de `Monitor`, o mediante una llamada al método `Sleep` de `Thread`, también sale del estado *WaitSleepJoin* y regresa al estado

Running si otro subproceso en el programa hace una llamada al método **Interrupt** del subproceso inactivo o en espera. El método **Interrupt** lanza una excepción **ThreadInterruptedException** en el subproceso que se interrumpe.

Si un subproceso no puede seguir ejecutándose (a éste le llamaremos el subproceso dependiente) a menos que termine otro subproceso, el subproceso dependiente llama al método **Join** del otro subproceso para “unir” los dos subprocesos. Cuando se “unen” dos subprocesos, el subproceso dependiente sale del estado *WaitSleepJoin* y vuelve a entrar al estado *Running* cuando el otro subproceso termina su ejecución (y entra al estado *Stopped*).

Si se llama al método **Suspend** de un subproceso en ejecución, éste entra al estado *Suspended* (suspendido). Un subproceso en el estado *Suspended* regresa al estado *Running* cuando otro subproceso en el programa invoca al método **Resume** del subproceso suspendido. [Nota: Internamente, cuando se hace una llamada al método **Suspend**, el subproceso en realidad entra al estado *SuspendRequested* (suspensión solicitada) antes de entrar al estado *Suspended*. El subproceso permanece en el estado *SuspendRequested* mientras espera a responder a la petición **Suspend**. Si el subproceso se encuentra en el estado *WaitSleepJoin* o si se bloquea cuando se hace la llamada a su método **Suspend**, el subproceso reside en su estado actual y en el estado *SuspendRequest*, y no puede responder a la petición **Suspend** hasta que salga de su estado actual.] Los métodos **Suspend** y **Resume** ahora están obsoletos y no deben usarse. En la sección 15.9 le mostraremos cómo emular estos métodos mediante la sincronización de subprocesos.

Si la propiedad **IsBackground** de un subproceso es **true**, el subproceso reside en el estado *Background* (segundo plano) (no se muestra en la figura 15.1). Un subproceso puede residir en el estado *Background* y en cualquier otro estado al mismo tiempo. Para que un proceso pueda terminar, debe esperar a que todos los *subprocesos en primer plano* (los subprocesos que no se encuentran en el estado *Background*) terminen de ejecutarse y entren en el estado *Stopped* (detenidos) antes de que finalice el proceso. No obstante, si los únicos subprocesos restantes en un proceso son *subprocesos en segundo plano*, el CLR termina cada subproceso invocando a su método **Abort**, y el proceso termina.

15.3 Prioridades y programación de subprocesos

Todo subproceso tiene una prioridad en el rango entre **ThreadPriority.Lowest** y **ThreadPriority.Highest**. Estos valores provienen de la enumeración **ThreadPriority** (espacio de nombres **System.Threading**), la cual consiste de los valores **Lowest**, **BelowNormal**, **Normal**, **AboveNormal** y **Highest**. De manera predeterminada, cada subproceso tiene la prioridad **Normal**.

El sistema operativo Windows soporta un concepto conocido como partición de tiempo, el cual permite a los subprocesos de igual prioridad compartir un procesador. Sin la partición de tiempo, cada subproceso en un conjunto de subprocesos de igual prioridad se ejecuta hasta completarse (a menos que el subproceso salga del estado *Running* y entre al estado *WaitSleepJoin*, *Suspended* o *Blocked*), antes de que otros subprocesos de igual prioridad tengan una oportunidad para ejecutarse. Con la partición de tiempo, cada subproceso recibe un breve lapso de tiempo del procesador, conocido como cuanto, durante el cual puede ejecutarse. Cuando el cuanto expira, incluso aunque el subproceso no haya terminado de ejecutarse, el procesador se quita de ese subproceso y se asigna al siguiente de igual prioridad, si hay uno disponible.

El trabajo del *programador de subprocesos* es mantener el subproceso de mayor prioridad en ejecución en todo momento y, si hay más de un proceso de mayor prioridad, debe asegurarse que todos los subprocesos se ejecuten durante un cuanto cada uno, en forma cíclica (round-robin). En la figura 15.2 se muestra la *cola de prioridades multinivel* para los subprocesos. En la figura, suponiendo que hay una computadora con un solo procesador, los subprocesos A y B se ejecutan cada uno durante un cuanto, en forma cíclica (round-robin) hasta que ambos subprocesos terminan su ejecución. Esto significa que A obtiene un cuanto de tiempo para ejecutarse. Después B obtiene un cuanto. Luego A obtiene otro cuanto; y después B, otro. Esto continúa hasta que un subproceso se completa. Entonces, el procesador dedica todo su poder al subproceso restante (a menos que se inicie otro subproceso de esa prioridad). A continuación, el subproceso C se ejecuta hasta completarse. Los subprocesos D, E y F se ejecutan cada uno durante un cuanto, en forma cíclica hasta que todos terminan de ejecutarse. Este proceso continúa hasta que todos los subprocesos terminan de ejecutarse. Observe que, dependiendo del sistema operativo, los nuevos procesos de mayor prioridad podrían aplazar (tal vez en forma indefinida) la ejecución de los subprocesos de menor prioridad. Comúnmente, a dicho aplazamiento indefinido se le conoce, en forma más colorida, como inanición.

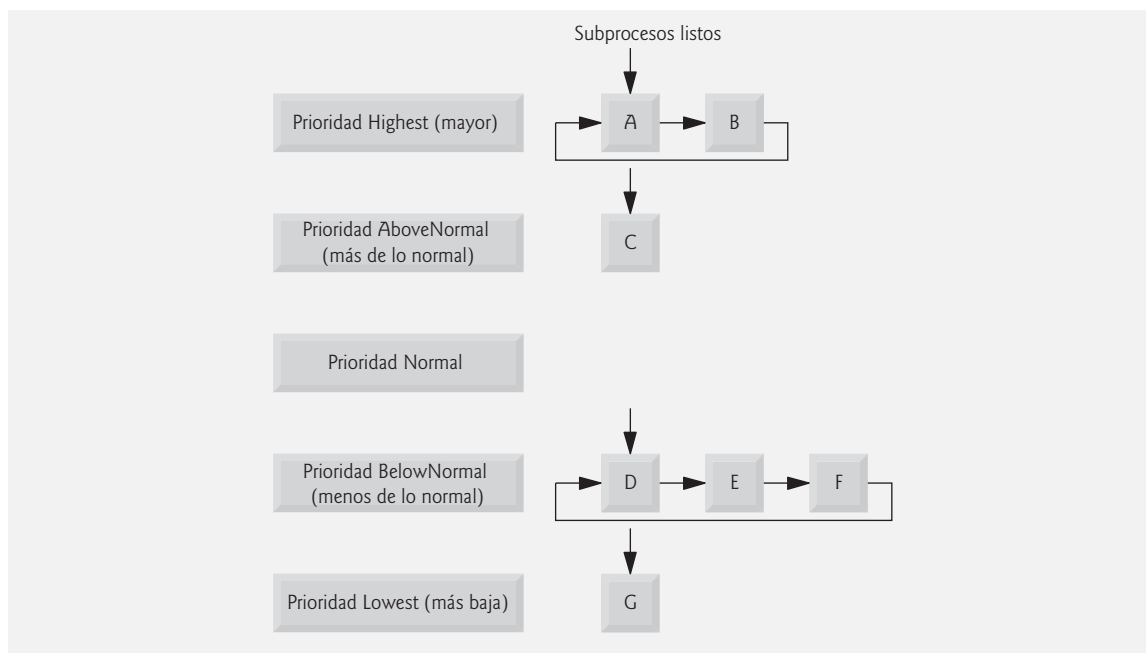


Figura 15.2 | Programación de prioridades de subprocesos.

La prioridad de un subproceso puede ajustarse mediante la propiedad **Priority**, la cual acepta valores de la enumeración **ThreadPriority**. Si el valor especificado no es una de las constantes válidas de prioridad de subprocesos, se produce una excepción **ArgumentException**.

Un subproceso se ejecuta hasta que muere, al *bloquearse* su entrada/salida (o por alguna otra razón), al llamar a **Sleep**, al método **Wait** o **Join** de **Monitor**, cuando es desplazado por un subproceso de mayor prioridad, o al *expirar su cuanto*. Un subproceso con mayor prioridad que el subproceso *en ejecución* puede ponerse *en ejecución* (y por lo tanto desplazar al primer subproceso *en ejecución*) si un subproceso está inactivo y posteriormente se despierta, si la E/S se completa para un subproceso que tenía esa E/S *bloqueada*, o si se llama al método **Pulse** o **PulseAll** en un objeto en el que se haya llamado previamente a **Wait**, si el proceso *reanuda* su ejecución desde el estado *suspendido*, o si se completa un subproceso al que estaba unido el subproceso de mayor prioridad.

15.4 Creación y ejecución de subprocesos

La figura 15.3 demuestra las técnicas básicas de subprocesamiento, como la construcción de objetos **Thread** y el uso del método **static Sleep** de la clase **Thread**. El programa crea tres subprocesos de ejecución, cada uno con la prioridad predeterminada **Normal**. Cada subproceso muestra un mensaje indicando que estará inactivo durante un intervalo aleatorio de 0 a 5000 milisegundos, y luego se vuelve inactivo. Cuando cada subproceso despierta, muestra su nombre, indica que ha dejado de estar inactivo, termina y entra al estado *Stopped* (detenido). En este ejemplo podrá ver que el método **Main** (es decir, el *subproceso de ejecución Main*) termina antes de que la aplicación finalice. El programa consta de dos clases: **ProbadorSubprocesos** (líneas 7-35), la cual crea los tres subprocesos, e **ImpresoraMensajes** (líneas 38-64), que define a un método **Imprimir** que contiene las acciones que cada subproceso llevará a cabo.

Los objetos de la clase **ImpresoraMensajes** (líneas 38-64) controlan el ciclo de vida de cada uno de los tres subprocesos creados en el método **Main** de la clase **ProbadorSubprocesos**. La clase **ImpresoraMensajes** consta de la variable de instancia **tiempoInactividad** (línea 40), la variable **static** **aleatorio** (línea 41), un constructor (líneas 44-48) y un método **Imprimir** (líneas 51-63). La variable **tiempoInactividad** almacena un valor entero aleatorio, el cual se selecciona cuando se hace una llamada al constructor de un nuevo objeto **ImpresoraMensajes**. Cada subproceso controlado por un objeto **ImpresoraMensajes** permanece inactivo durante el tiempo especificado por el valor de **tiempoInactividad** correspondiente a cada objeto **ImpresoraMensajes**.

```

1  // Fig. 15.3: ProbadorSubprocesos.cs
2  // Varios subprocesos imprimiendo en distintos intervalos.
3  using System;
4  using System.Threading;
5
6  // la clase ProbadorSubprocesos demuestra los conceptos básicos del subprocesamiento
7  class ProbadorSubprocesos
8  {
9      static void Main( string[] args )
10     {
11         // Crea y asigna un nombre a cada subproceso. Usa el método Imprimir
12         // de ImpresoraMensajes como argumento para el delegado ThreadStart.
13         ImpresoraMensajes impresora1 = new ImpresoraMensajes();
14         Thread subproceso1 = new Thread ( new ThreadStart( impresora1.Imprimir ) );
15         subproceso1.Name = "subproceso1";
16
17         ImpresoraMensajes impresora2 = new ImpresoraMensajes();
18         Thread subproceso2 = new Thread ( new ThreadStart( impresora2.Imprimir ) );
19         subproceso2.Name = "subproceso2";
20
21         ImpresoraMensajes impresora3 = new ImpresoraMensajes();
22         Thread subproceso3 = new Thread ( new ThreadStart( impresora3.Imprimir ) );
23         subproceso3.Name = "subproceso3";
24
25         Console.WriteLine( "Iniciando subprocesos" );
26
27         // llama al método Start de cada subproceso para colocar a cada
28         // uno de ellos en el estado Running
29         subproceso1.Start();
30         subproceso2.Start();
31         subproceso3.Start();
32
33         Console.WriteLine( "Subprocesos iniciados\n" );
34     } // fin del método Main
35 } // fin de la clase ProbadorSubprocesos
36
37 // el método Imprimir de esta clase se utiliza para controlar los subprocesos
38 class ImpresoraMensajes
39 {
40     private int tiempoInactividad;
41     private static Random aleatorio = new Random();
42
43     // constructor para inicializar un objeto ImpresoraMensajes
44     public ImpresoraMensajes()
45     {
46         // elige tiempo de inactividad aleatorio entre 0 y 5 segundos
47         tiempoInactividad = aleatorio.Next( 5001 ); // 5001 milisegundos
48     } // fin del constructor
49
50     // el método Imprimir controla el subproceso que imprime los mensajes
51     public void Imprimir()
52     {
53         // obtiene la referencia al subproceso que se ejecuta en este momento
54         Thread actual = Thread.CurrentThread;
55
56         // pone subproceso en inactividad durante un tiempo indicado por tiempoInactividad
57         Console.WriteLine( "{0} va a estar inactivo durante {1} milisegundos",
58             actual.Name, tiempoInactividad );

```

Figura 15.3 | Subprocesos inactivos e imprimiendo. (Parte 1 de 2).


```

59      Thread.Sleep( tiempoInactividad ); // inactivo durante tiempoInactividad milisegundos
60
61      // imprime el nombre del subproceso
62      Console.WriteLine( "{0} dejó de estar inactivo", actual.Name );
63  } // fin del método Imprimir
64 } // fin de la clase ImpresoraMensajes

```

Iniciando subprocesos

subproceso1 va a estar inactivo durante 1603 milisegundos
 subproceso2 va a estar inactivo durante 2355 milisegundos
 subproceso3 va a estar inactivo durante 285 milisegundos
 Subprocesos iniciados

subproceso3 dejó de estar inactivo
 subproceso1 dejó de estar inactivo
 subproceso2 dejó de estar inactivo

Iniciando subprocesos

subproceso1 va a estar inactivo durante 4245 milisegundos
 subproceso2 va a estar inactivo durante 1466 milisegundos
 Subprocesos iniciados

subproceso3 va a estar inactivo durante 1929 milisegundos
 subproceso2 dejó de estar inactivo
 subproceso3 dejó de estar inactivo
 subproceso1 dejó de estar inactivo

Figura 15.3 | Subprocesos inactivos e imprimiendo. (Parte 2 de 2).

El constructor de `ImpresoraMensajes` (líneas 44-48) inicializa `tiempoInactividad` con un número aleatorio de milisegundos, desde 0 hasta pero sin incluir a, 5001 (es decir, de 0 a 5000).

El método `Imprimir` empieza por obtener una referencia al subproceso que se está ejecutando en ese momento (línea 54), a través de la propiedad `static CurrentThread` de la clase `Thread`. El subproceso que se ejecuta actualmente es el que invoca al método `Imprimir`. A continuación, las líneas 57-58 muestran un mensaje que indica el nombre del subproceso que se ejecuta actualmente, y también indica que el subproceso permanecerá inactivo durante cierto número de milisegundos. Observe que la línea 58 utiliza la propiedad `Name` del subproceso actual en ejecución, para obtener el nombre del subproceso (el cual se establece en el método `Main`, a la hora de crear cada subproceso). La línea 59 invoca al método `static Sleep` de `Thread` para colocar el subproceso en el estado `WaitSleepJoin`. En este punto, el subproceso pierde el procesador y el sistema permite que se ejecute otro subproceso, si hay uno listo para ejecutarse. Cuando el subproceso despierta, entra nuevamente en el estado `Running` y espera a que el programador de subprocesos le asigne un procesador. Cuando el objeto `ImpresoraMensajes` entra de nuevo al estado `Running`, la línea 62 imprime en pantalla el nombre del subproceso en un mensaje que indica que salió de su inactividad, y el método `Imprimir` termina.

El método `Main` de la clase `ProbadorSubprocesos` (líneas 9-34) crea tres objetos de la clase `ImpresoraMensajes` en las líneas 13, 17 y 21, respectivamente. Las líneas 14, 18 y 22 crean e inicializan tres objetos `Thread`. Observe que el constructor de cada objeto `Thread` recibe un delegado `ThreadStart` como argumento. Un delegado `ThreadStart` representa a un método sin argumentos y con un tipo de valor de retorno `void`, que especifica las acciones que debe realizar un subproceso. La línea 14 inicializa el delegado `ThreadStart` para `subproceso1`, con el método `Imprimir` de `impresora1`. Cuando `subproceso1` entra al estado `Running` por primera vez, invoca al método `Imprimir` de `impresora1` para realizar las tareas especificadas en el cuerpo del método `Imprimir`. Así, `subproceso1` imprimirá su nombre, mostrará el monto del tiempo durante el cual quedará inactivo, permanecerá inactivo durante ese monto de tiempo, despertará y mostrará un mensaje indicando que el subproceso dejó de estar inactivo. En ese punto, el método `Imprimir` terminará. Un subproceso completa su tarea cuando termina el método especificado por su delegado `ThreadStart`, punto en el cual el subproceso entra al estado `Stopped`. Cuando `subproceso2` y `subproceso3` entran al estado `Running` por primera vez, invocan

a los métodos de `impresora2` e `impresora3`, respectivamente; `subproceso2` y `subproceso3` realizan las mismas tareas que `subproceso1` mediante la ejecución de los métodos `Imprimir` de los objetos a los que `impresora2` e `impresora3` hacen referencia (cada uno de los cuales tiene su propio tiempo de inactividad elegido al azar). Las líneas 15, 19 y 23 establecen la propiedad `Name` de cada subproceso, que utilizaremos con el fin de mostrar los resultados.



Tip de prevención de errores 15.1

*Asignar nombres a los subprocesos ayuda en la depuración de un programa con subprocesamiento múltiple. El depurador de Visual Studio .NET cuenta con una ventana llamada **Subprocesos**, que muestra el nombre de cada subproceso y nos permite ver la ejecución de cualquier subproceso en el programa.*

Las líneas 29-31 invocan al método `Start` de cada subproceso, para colocar los subprocesos en el estado *Running*. El método `Start` regresa inmediatamente de cada invocación, y después la línea 33 imprime en pantalla un mensaje, indicando que se iniciaron los subprocesos y el subproceso de ejecución `Main` termina. Sin embargo, el programa en sí no termina, ya que aún quedan subprocesos activos que no están en segundo plano (es decir, los subproceso que están en ejecución y no han llegado al estado *Stopped*). El programa no terminará sino hasta que muera su último subproceso que no esté en segundo plano. Cuando el sistema asigna un procesador a un subproceso, éste entra en el estado *Running* y llama al método especificado por el delegado `ThreadStart` del subproceso. En este programa, cada subproceso invoca al método `Imprimir` del objeto `ImpresoraMensaje` apropiado para realizar las tareas antes descritas.

Observe que los resultados de ejemplo para este programa muestran a cada subproceso, junto con el tiempo de inactividad del subproceso en milisegundos, cuando éste queda inactivo. El subproceso con el menor tiempo de inactividad es el que por lo general se despierta primero; después indica que dejó de estar inactivo y termina. En la sección 15.8 hablaremos sobre cuestiones relacionadas con el subprocesamiento múltiple que podrían evitar que el subproceso con el menor tiempo de inactividad se despierte primero (nada de esto se garantiza). Observe en el segundo conjunto de resultados de ejemplo que `subproceso1` y `subproceso2` pudieron reportar sus tiempos de inactividad antes de que `Main` pudiera imprimir su último mensaje. Esto significa que el cuanto del subproceso principal terminó antes de que pudiera terminar de ejecutar `Main`, y tanto `subproceso1` como `subproceso2` tuvieron la oportunidad de ejecutarse.

15.5 Sincronización de subprocesos y la clase `Monitor`

A menudo, varios subprocesos de ejecución manipulan datos compartidos. Si los subprocesos con acceso a los datos compartidos sólo leen esos datos, entonces cualquier número de subprocesos podría acceder a esos datos en forma simultánea, y no surgirían problemas. No obstante, cuando varios subprocesos comparten datos, y uno o más modifican esos datos, entonces se pueden producir resultados indeterminados. Si un subproceso se encuentra en el proceso de actualizar los datos, y otro trata de actualizarlos también, los datos reflejarán sólo la última actualización. Si los datos son un arreglo, o cualquier otra estructura de datos en la que los subprocesos pudieran actualizar partes separadas de los datos en forma concurrente, es posible que esa parte de los datos refleje la información de un subproceso, mientras que parte de los datos reflejará información de otro subproceso. Cuando esto ocurre, el programa tiene dificultad para determinar si se actualizaron los datos en forma apropiada o no.

Para resolver este problema, es necesario proporcionar a un proceso a la vez el acceso exclusivo al código que manipula los datos compartidos. Durante ese tiempo, otros subprocesos que deseen manipular los datos deben mantenerse en espera. Cuando el subproceso con acceso exclusivo a los datos complete sus manipulaciones de datos, se debe permitir que uno de los subprocesos en espera proceda. De esta forma, cada subproceso que acceda a los datos compartidos excluye a todos los demás de hacerlo en forma simultánea. A esto se le conoce como *exclusión mutua*, o sincronización de subprocesos.

C# utiliza los monitores del .NET Framework para realizar la sincronización. La clase `Monitor` proporciona los métodos para bloquear objetos e implementar el acceso sincronizado a los datos compartidos. Bloquear un objeto significa que sólo un subproceso puede acceder a ese objeto en un momento dado. Cuando un subproceso desea adquirir el control exclusivo sobre un objeto, invoca al método `Enter` de `Monitor` para *adquirir el bloqueo* sobre ese objeto de datos. Cada objeto tiene un bloque de sincronización (`SyncBlock`) que mantiene el estado del

bloqueo de ese objeto. Los métodos de la clase `Monitor` usan los datos en un `SyncBlock` de un objeto para determinar el estado del bloqueo para ese objeto. Después de adquirir el bloqueo para un objeto, un subproceso puede manipular los datos de ese objeto. Mientras el objeto esté bloqueado, todos los demás subprocesos que traten de adquirir el bloqueo sobre ese objeto se bloquearán y no podrán adquirirlo; dichos subprocesos entran en el estado *Blocked*. Cuando el subproceso que bloqueó el objeto compartido ya no requiere el bloqueo, invoca al método `Exit` de `Monitor` para liberar el bloqueo. Eso actualiza el `SyncBlock` del objeto compartido, para indicar que el bloqueo para ese objeto está disponible otra vez. En este punto, si hay un subproceso que estaba bloqueado y no podía adquirir el bloqueo sobre el objeto compartido, adquiere el bloqueo para empezar su procesamiento del objeto. Si todos los subprocesos con acceso a un objeto tratan de adquirir el bloqueo del objeto antes de manipularlo, sólo se permitirá que un subproceso a la vez manipule el objeto. Esto ayuda a asegurar la integridad de los datos.



Error común de programación 15.1

Asegúrese de que todo el código que actualice a un objeto compartido bloquee al objeto antes de actualizarlo. En caso contrario, un subproceso que llame a un método que no bloquee el objeto puede provocar que se vuelva inestable, aun cuando otro subproceso haya adquirido el bloqueo para ese objeto.



Error común de programación 15.2

*El interbloqueo ocurre cuando un subproceso en espera (llamémosle **subproceso1**) no puede proceder, debido a que está esperando (ya sea en forma directa o indirecta) que otro subproceso (llamado **subproceso2**) proceda, mientras que al mismo tiempo, el **subproceso2** no puede proceder ya que está esperando (en forma directa o indirecta) a que el **subproceso1** proceda. Dos subprocesos están esperando uno al otro, por lo que las acciones que permitirían que cualquiera de los continuara su ejecución nunca ocurrirán.*

C# proporciona otro medio para manipular el bloqueo de un objeto: la palabra clave **lock**. Al colocar esta palabra antes de un bloque de código (designado mediante llaves), como en

```
lock ( referenciaObjeto )
{
    // aquí va el código que requiere sincronización
}
```

se obtiene el bloqueo sobre el objeto al cual *referenciaObjeto* hace referencia. Esta referencia es la misma que se pasaría comúnmente a los métodos `Enter`, `Exit`, `Pulse` y `PulseAll` de `Monitor`. Cuando un bloqueo termina por cualquier razón, C# libera el bloqueo sobre el objeto al cual se hace referencia. En la sección 15.8 explicaremos la función de `lock` con más detalle.

Si un subproceso que posee el bloqueo sobre un objeto determina que no puede continuar con su tarea, sino hasta que se cumpla cierta condición, el subproceso debe llamar al método `Wait` de `Monitor` y pasar como argumento el objeto al cual esperará el subproceso, hasta que pueda realizar su tarea. Al llamar al método `Monitor.Wait` desde un subproceso, se libera el bloqueo que éste tiene sobre el objeto que recibe `Wait` como argumento, y se coloca a ese subproceso en el estado *WaitSleepJoin* para ese objeto. Un subproceso en el estado *WaitSleepJoin* de un objeto específico deja ese estado cuando un subproceso separado invoca al método `Pulse` o `PulseAll` de `Monitor`, con ese objeto como argumento. El método `Pulse` hace que el primer subproceso en espera del objeto cambie del estado *WaitSleepJoin* al estado *Running*. El método `PulseAll` hace que todos los subprocesos en el estado *WaitSleepJoin* del objeto cambien al estado *Running*. La transición al estado *Running* permite que el subproceso (o subprocesos) se prepare para continuar su ejecución.

Hay una diferencia entre los subprocesos que esperan adquirir el bloqueo de un objeto y los subprocesos que esperan en el estado *WaitSleepJoin* de un objeto. Los subprocesos que llaman al método `Wait` de `Monitor` con un objeto como argumento, se colocan en el estado *WaitSleepJoin* de ese objeto. Los subprocesos que simplemente están esperando adquirir el bloqueo, entran al estado *Blocked* conceptual y esperan hasta que el bloqueo del objeto está disponible. Por lo tanto, un subproceso *bloqueado* puede adquirir el bloqueo de ese objeto.

Los métodos `Enter`, `Exit`, `Wait`, `Pulse` y `PulseAll` de `Monitor` reciben como argumento una referencia a un objeto (por lo general, la palabra clave `this`).



Error común de programación 15.3

Un subproceso en el estado WaitSleepJoin no puede volver a entrar en el estado Running para continuar su ejecución, sino hasta que un subproceso separado invoque al método `Pulse` o `PulseAll` de `Monitor` con el objeto apropiado como argumento. Si esto no ocurre, el subproceso en espera quedará esperando para siempre; en esencia, esto es equivalente al interbloqueo.



Tip de prevención de errores 15.2

Cuando varios subprocesos manipulan a un objeto compartido mediante el uso de monitores, debemos asegurarnos que, si un subproceso llama al método `Wait` de `Monitor` para entrar al estado WaitSleepJoin para el objeto compartido, haya un subproceso separado que en algún momento dado llame al método `Pulse` de `Monitor` para cambiar al subproceso que espera al objeto compartido, de vuelta al estado Running. Si varios subprocesos están esperando al objeto compartido, un subproceso separado puede llamar al método `PulseAll` de `Monitor` como precaución, para asegurar que todos los subprocesos en espera tengan otra oportunidad para realizar sus tareas. Si no se hace esto, podría ocurrir un aplazamiento indefinido o interbloqueo.



Tip de rendimiento 15.2

La sincronización para lograr que los programas con subprocesamiento múltiple funcionen en forma correcta puede hacer que los programas se ejecuten con más lentitud, como resultado de la sobrecarga de los monitores y las frecuentes transiciones de los subprocesos, entre los estados WaitSleepJoin y Running. Sin embargo, no hay mucho que decir en cuanto a los programas con subprocesamiento múltiple que son altamente eficientes, ¡pero incorrectos!

15.6 Relación productor/consumidor sin sincronización de subprocesos

En una relación productor/consumidor, la porción correspondiente al *productor* de una aplicación genera datos, y la porción correspondiente al *consumidor* utiliza esos datos. En una relación productor/consumidor con subprocesamiento múltiple, un subproceso productor llama a un método “producir” para generar datos y colocarlos en una región compartida de memoria, llamada búfer. Un subproceso consumidor llama a un método “consumir” para leer esos datos. Si el productor desea colocar los siguientes datos en el búfer, pero determina que el consumidor no ha leído todavía los datos anteriores del búfer, el subproceso productor debe llamar a `Wait`. En caso contrario, el consumidor nunca vería los datos anteriores, y la aplicación los perdería. Cuando el subproceso consumidor lee los datos, debe llamar a `Pulse` para permitir que un productor en espera continúe, ya que ahora hay un espacio libre en el búfer. Si un subproceso consumidor descubre que el búfer está vacío, o que ya leyó los datos anteriores, debe llamar a `Wait`. En caso contrario, el consumidor podría leer “basura” del búfer, o podría procesar un elemento de datos anterior más de una vez; cada una de estas posibilidades provoca un error lógico en la aplicación. Cuando el productor coloque los siguientes datos en el búfer, debe llamar a `Pulse` para permitir que continúe el subproceso consumidor y lea esos datos.

Consideraremos cómo pueden surgir errores lógicos si no sincronizamos el acceso entre varios subprocesos que manipulan datos compartidos. Imagine una relación productor/consumidor, en la que un subproceso productor escribe una secuencia de números (utilizamos del 1 al 10) en un búfer compartido; una ubicación de memoria compartida entre varios subprocesos. El subproceso consumidor lee estos datos del búfer compartido y después los muestra en pantalla. En los resultados del programa mostramos los valores que el productor escribe (produce) y que el consumidor lee (consume). Las figuras 15.4-15.8 demuestran cómo un subproceso productor y un subproceso consumidor acceden a una sola variable `int` compartida, sin ningún tipo de sincronización. El subproceso productor escribe en la variable; el subproceso consumidor lee de esta variable. Nos gustaría que cada valor que escribe el subproceso productor en la variable compartida se consumiera sólo una vez por el subproceso consumidor. No obstante, los subprocesos en este ejemplo no están sincronizados. Por lo tanto, los datos pueden perderse si el productor coloca nuevos datos en la variable, antes de que el consumidor consuma los datos anteriores. Además, los datos podrían repetirse incorrectamente si el consumidor consume los datos de nuevo, antes de que el productor produzca el siguiente valor. Si el consumidor trata de leer antes de que el productor produzca el primer valor, el consumidor lee basura. Para mostrar estas posibilidades, el subproceso consumidor en el ejemplo mantiene un total de todos los valores que lee. El subproceso productor produce valores del 1 al 10. Si el consumidor lee sólo una vez cada valor producido, el total sería de 55. No obstante, al ejecutar este programa varias veces,

podrá ver que el total casi nunca será de 55, si es que alguna vez llega a serlo. Además, para enfatizar nuestro punto de vista, tanto el subproceso productor como el subproceso consumidor en el ejemplo quedan inactivos durante intervalos aleatorios de hasta tres segundos, entre cada tarea que realizan. Por lo tanto, no sabemos con exactitud cuándo tratará el subproceso productor de escribir un nuevo valor, ni sabemos cuándo tratará el consumidor de leer un valor.

El programa consta de la interfaz *Bufer* (figura 15.4) y las clases *Productor* (figura 15.5), *Consumidor* (figura 15.6), *BuferSinSincronizacion* (figura 15.7) y *PruebaBuferSinSincronizacion* (figura 15.8). La interfaz *Bufer* declara una propiedad *int* llamada *Bufer*. Cualquier implementación de *Bufer* debe proporcionar un descriptor de acceso *get* y un descriptor de acceso *set* para que esta propiedad pueda permitir al productor y al consumidor el acceso a los datos compartidos.

La clase *Productor* (figura 15.5) consta de la variable de instancia *ubicacionCompartida* (línea 10) de tipo *Bufer*, la variable de instancia *tiempoInactividadAleatorio* (línea 11) de tipo *Random*, un constructor (líneas 14-18) para inicializar las variables de instancia y un método *Producir* (líneas 21-33). El constructor inicializa la variable de instancia *ubicacionCompartida* para hacer referencia al objeto *Bufer* que recibe del método *Main* como el parámetro compartido. El subproceso productor en este programa ejecuta las tareas especificadas en el método *Producir* de la clase *Productor*. La instrucción *for* en el método *Producir* (líneas 25-29) itera 10 veces. Cada iteración del ciclo invoca primero al método *Sleep* de *Thread*, para colocar el subproceso productor en el estado *WaitSleepJoin* durante un intervalo de tiempo aleatorio entre 0 y 3 segundos. Cuando el subproceso despierta, la línea 28 asigna el valor de la variable de control *cuenta* a la propiedad *Bufer* de *ubicacionCompartida*. Cuando termina el ciclo, las líneas 31-32 muestran una línea de texto en la ventana de consola, indicando que el subproceso terminó de producir datos y que terminará. Después, el método *Producir* termina y el subproceso productor entra al estado *Stopped*.

La clase *Consumidor* (figura 15.6) consta de la variable de instancia *ubicacionCompartida* (línea 10) de tipo *Bufer*, la variable de instancia *tiempoInactividadAleatorio* (línea 11) de tipo *Random*, un constructor (líneas 14-18) para inicializar las variables de instancia, y un método *Consumir* (líneas 21-36). El constructor inicializa *ubicacionCompartida* para que haga referencia al objeto *Bufer* que recibió de *Main* como el parámetro compartido. El subproceso consumidor en este programa realiza las tareas especificadas en el método *Consumir* de la clase *Consumidor*. El método contiene una instrucción *for* (líneas 27-31) que itera diez veces. Cada iteración del ciclo invoca al método *Sleep* de *Thread* para colocar al subproceso consumidor en el estado *WaitSleepJoin* durante un intervalo de tiempo aleatorio entre 0 y 3 segundos. A continuación, la línea 30 obtiene el valor de la propiedad *Bufer* de *ubicacionCompartida* y lo suma a la variable *suma*. Cuando se completa el ciclo, las líneas 33-35 muestran una línea en la ventana de consola, la cual indica la suma de todos los valores leídos. Nuevamente, el total debería ser 55, pero debido a que el acceso a los datos compartidos no está sincronizado, es muy probable que esta suma nunca aparezca. Después, el método *Consumir* termina y el subproceso consumidor entra al estado *Stopped*.

Utilizamos el método *Sleep* en los subprocesos de este ejemplo para enfatizar el hecho de que, en las aplicaciones con subprocesamiento múltiple, no se sabe con exactitud cuándo realizará cada subproceso su tarea, ni

```

1 // Fig. 15.4: Bufer.cs
2 // Interfaz para un búfer compartido de valores int.
3 using System;
4
5 // esta interfaz representa a un búfer compartido
6 public interface Bufer
7 {
8     // propiedad Bufer
9     int Bufer
10 {
11     get;
12     set;
13 } // fin de la propiedad Bufer
14 } // fin de la interfaz Bufer

```

Figura 15.4 | La interfaz *Bufer* que se utiliza en los ejemplos de productor/consumidor.

```

1  // Fig. 15.5: Productor.cs
2  // Productor produce 10 valores enteros en el búfer compartido.
3  using System;
4  using System.Threading;
5
6  // el método Producir de la clase Productor controla un subproceso que
7  // almacena valores del 1 al 10 en ubicacionCompartida
8  public class Productor
9  {
10     private Bufer ubicacionCompartida;
11     private Random tiempoInactividadAleatorio;
12
13     // constructor
14     public Productor( Bufer compartido, Random aleatorio )
15     {
16         ubicacionCompartida = compartido;
17         tiempoInactividadAleatorio = aleatorio;
18     } // fin del constructor
19
20     // almacena valores del 1 al 10 en el objeto ubicacionCompartida
21     public void Producir()
22     {
23         // duerme durante intervalo aleatorio de hasta 3000 milisegundos
24         // después establece la propiedad Bufer de ubicacionCompartida
25         for ( int cuenta = 1; cuenta <= 10; cuenta++ )
26         {
27             Thread.Sleep( tiempoInactividadAleatorio.Next( 1, 3001 ) );
28             ubicacionCompartida.Bufer = cuenta;
29         } // fin de for
30
31         Console.WriteLine( "{0} terminó de producir.\nTerminando {0}.",
32                             Thread.CurrentThread.Name );
33     } // fin del método Producir
34 } // fin de la clase Productor

```

Figura 15.5 | Productor representa al subproceso productor en una relación productor/consumidor.

por cuánto tiempo realizará esa tarea cuando tenga el procesador. Por lo general, estas cuestiones relacionadas con la programación de tareas son responsabilidad del sistema operativo de la computadora. En este programa, las tareas de nuestro subproceso son bastante simples: para el productor, iterar 10 veces y ejecutar una instrucción de asignación; para el consumidor, iterar 10 veces y sumar un valor a la variable suma. Sin la llamada al método Sleep, y si el productor se ejecuta primero, es muy probable que complete su tarea antes de que el consumidor tenga siquiera una oportunidad de ejecutarse. Si el consumidor se ejecuta primero, consumiría -1 diez veces y después terminaría antes de que el productor pudiera producir el primer valor real.

```

1  // Fig. 15.6: Consumidor.cs
2  // El consumidor consume 10 valores enteros del búfer compartido.
3  using System;
4  using System.Threading;
5
6  // el método Consumir de la clase Consumidor controla un subproceso que
7  // itera 10 veces y lee un valor de ubicacionCompartida
8  public class Consumidor
9  {
10     private Bufer ubicacionCompartida;

```

Figura 15.6 | Consumidor representa al subproceso consumidor en una relación productor/consumidor. (Parte I de 2).

```

11  private Random tiempoInactividadAleatorio;
12
13  // constructor
14  public Consumidor( Bufer compartido, Random aleatorio )
15  {
16      ubicacionCompartida = compartido;
17      tiempoInactividadAleatorio = aleatorio;
18  } // fin del constructor
19
20  // lee el valor de ubicacionCompartida diez veces
21  public void Consumir()
22  {
23      int suma = 0;
24
25      // duerme durante intervalo aleatorio de hasta 3000 milisegundos, después
26      // suma el valor de la propiedad Bufer de ubicacionCompartida a suma
27      for ( int cuenta = 1; cuenta <= 10; cuenta++ )
28      {
29          Thread.Sleep( tiempoInactividadAleatorio.Next( 1, 3001 ) );
30          suma += ubicacionCompartida.Bufer;
31      } // fin de for
32
33      Console.WriteLine(
34          "{0} leyó valores para un total de: {1}.\nTerminando {0}.",
35          Thread.CurrentThread.Name, suma );
36  } // fin del método Consumir
37  } // fin de la clase Consumidor

```

Figura 15.6 | Consumidor representa al subproceso consumidor en una relación productor/consumidor. (Parte 2 de 2).

La clase BuferSinSincronizacion (figura 15.7) implementa la interfaz Bufer (línea 7) y consta de la variable de instancia bufer (línea 10) y la propiedad Bufer (líneas 13-27), que proporciona los descriptores de acceso get y set. Los descriptores de acceso de la propiedad Bufer no sincronizan el acceso a la variable de instancia bufer. Observe que cada descriptor de acceso utiliza la propiedad static CurrentThread de la clase Thread para obtener una referencia al subproceso actual en ejecución, y después utiliza la propiedad Name de ese subproceso para obtener el nombre del subproceso, con el fin de mostrar los resultados en pantalla.

```

1  // Fig. 15.7: BuferSinSincronizacion.cs
2  // Implementación de un búfer compartido sin sincronización.
3  using System;
4  using System.Threading;
5
6  // esta clase representa a un solo valor int compartido
7  public class BuferSinSincronizacion : Bufer
8  {
9      // bufer compartido por los subprocesos productor y consumidor
10     private int bufer = -1;
11
12     // propiedad Bufer
13     public int Bufer
14     {
15         get
16         {
17             Console.WriteLine( "{0} lee {1}",

```

Figura 15.7 | BuferSinSincronizacion mantiene la variable entera compartida a la que acceden los subprocesos productor y consumidor, a través de la propiedad Bufer. (Parte 1 de 2).

```

18         Thread.CurrentThread.Name, bufer );
19     return bufer;
20 } // fin de get
21 set
22 {
23     Console.WriteLine( "{0} escribe {1}",
24         Thread.CurrentThread.Name, value );
25     bufer = value;
26 } // fin de set
27 } // fin de la propiedad Bufer
28 } // fin de la clase BuferSinSincronizacion

```

Figura 15.7 | BuferSinSincronizacion mantiene la variable entera compartida a la que acceden los subprocesos productor y consumidor, a través de la propiedad Bufer. (Parte 2 de 2).

La clase PruebaBuferSinSincronizacion (figura 15.8) define un método Main (líneas 11-36) que instancia un objeto BuferSinSincronizacion compartido (línea 14) y un objeto Random (línea 17), para generar tiempos de inactividad aleatorios. Éstos se utilizan como argumentos para los constructores de los objetos de las clases Productor (línea 20) y Consumidor (línea 21). El objeto BuferSinSincronizacion contiene los datos que se compartirán entre los subprocesos productor y consumidor. Como BuferSinSincronizacion implementa la interfaz Bufer, los constructores de Productor y de Consumidor pueden tomar un objeto BuferSinSincronizacion y asignarlo a sus respectivas variables Bufer llamadas ubicacionCompartida. Las

```

1  // Fig. 15.8: PruebaBuferSinSincronizacion.cs
2  // Muestra varios subprocesos modificando un objeto compartido sin
3  // sincronización.
4  using System;
5  using System.Threading;
6
7  // esta clase crea los subprocesos productor y consumidor
8  class PruebaBuferSinSincronizacion
9  {
10     // crea los subprocesos productor y consumidor, y los inicia
11     static void Main( string[] args )
12     {
13         // crea objeto compartido utilizado por los subprocesos
14         BuferSinSincronizacion compartido = new BuferSinSincronizacion();
15
16         // Objeto aleatorio usado por cada subproceso
17         Random aleatorio = new Random();
18
19         // crea objetos Productor y Consumidor
20         Productor productor = new Productor( compartido, aleatorio );
21         Consumidor consumidor = new Consumidor( compartido, aleatorio );
22
23         // crea subprocesos para productor y consumidor, y establece
24         // delegados para cada subproceso
25         Thread subprocesoProductor =
26             new Thread( new ThreadStart( productor.Producir ) );
27         subprocesoProductor.Name = "Productor";
28
29         Thread subprocesoConsumidor =
30             new Thread( new ThreadStart( consumidor.Consumir ) );
31         subprocesoConsumidor.Name = "Consumidor";

```

Figura 15.8 | Los subprocesos productor y consumidor acceden a un objeto compartido sin sincronización. (Parte 1 de 2).

```

32
33     // inicia cada subproceso
34     subprocesoProductor.Start();
35     subprocesoConsumidor.Start();
36 } // fin de Main
37 } // fin de la clase PruebaBuferSinSincronizacion

```

```

Consumidor lee -1
Productor escribe 1
Consumidor lee 1
Productor escribe 2
Consumidor lee 2
Consumidor lee 2
Productor escribe 3
Consumidor lee 3
Consumidor lee 3
Productor escribe 4
Consumidor lee 4
Productor escribe 5
Consumidor lee 5
Consumidor lee 5
Productor escribe 6
Consumidor lee 6
Consumidor leyó valores para un total de: 30.
Terminando Consumidor.
Productor escribe 7
Productor escribe 8
Productor escribe 9
Productor escribe 10
Productor terminó de producir.
Terminando Productor.

```

```

Productor escribe 1
Productor escribe 2
Consumidor lee 2
Consumidor lee 2
Productor escribe 3
Productor escribe 4
Consumidor lee 4
Consumidor lee 4
Productor escribe 5
Consumidor lee 5
Productor escribe 6
Consumidor lee 6
Productor escribe 7
Consumidor lee 7
Productor escribe 8
Consumidor lee 8
Productor escribe 9
Productor escribe 10
Productor terminó de producir.
Terminando Productor.
Consumidor lee 10
Consumidor lee 10
Consumidor leyó valores para un total de: 58.
Terminando Consumidor.

```

Figura 15.8 | Los subprocesos productor y consumidor acceden a un objeto compartido sin sincronización.
(Parte 2 de 2).

líneas 25-27 crean y nombran a `subprocesoProductor`. El delegado `ThreadStart` para `subprocesoProductor` especifica que este subproceso ejecutará el método `Producir` del objeto productor. Las líneas 29-31 crean y nombran a `subprocesoConsumidor`. El delegado `ThreadStart` para el `subprocesoConsumidor` especifica que este subproceso ejecutará el método `Consumir` del objeto consumidor. Por último, las líneas 34-35 colocan los dos subprocesos en el estado *Running* mediante una invocación al método `Start` de cada subproceso, y después, el subproceso en ejecución `Main` termina.

Nos gustaría que el objeto `Consumidor` consumiera sólo una vez cada valor producido por el objeto `Productor`. Sin embargo, al estudiar el primer conjunto de resultados de la figura 15.8 podemos ver que el consumidor recuperó un valor (-1) antes de que el productor colocara uno en el búfer compartido, y que el consumidor consumió los valores 2, 3 y 5 dos veces cada uno. El consumidor terminó su ejecución antes de que el productor tuviera una oportunidad de producir los valores 7, 8, 9 y 10. Por lo tanto, se perdieron esos valores y se produjo una suma incorrecta. En el segundo conjunto de resultados, podemos ver que se perdió el valor 1, ya que los valores 1 y 2 se produjeron antes de que el subproceso consumidor pudiera leer el valor 1. Los valores 3 y 9 también se perdieron, y los valores 4 y 10 se consumieron dos veces cada uno, con lo cual también se produjo una suma incorrecta. Este ejemplo demuestra con claridad que el acceso a los datos compartidos por subprocesos concurrentes debe controlarse con mucho cuidado; de no ser así, un programa podría producir resultados incorrectos.

Para resolver los problemas de pérdida de datos y de consumir datos más de una vez, como en el ejemplo anterior, sincronizaremos (en las figuras 15.9-15.10) el acceso que tienen los subprocesos concurrentes productor y consumidor al código que manipula los datos compartidos, mediante el uso de los métodos `Enter`, `Wait`, `Pulse` y `Exit` de la clase `Monitor`. Cuando un subproceso utiliza la sincronización para acceder a un objeto compartido, el objeto se *bloquea*, de manera que ningún otro subproceso puede adquirir el bloqueo para ese objeto compartido al mismo tiempo.

15.7 Relación productor/consumidor con sincronización de subprocesos

Las figuras 15.9 y 15.10 demuestran cómo un productor y un consumidor acceden a una celda compartida de memoria con sincronización, de manera que el consumidor sólo consume una vez que el productor produce un valor, y el productor produce un nuevo valor sólo hasta que el consumidor consume el valor anterior producido. Este ejemplo reutiliza la interfaz `Bufere` (figura 15.4) y las clases `Productor` (figura 15.5) y `Consumidor` (figura 15.6) del ejemplo anterior. [Nota: en este ejemplo demostramos la sincronización mediante los métodos `Enter` y `Exit` de la clase `Monitor`. En el siguiente ejemplo demostraremos los mismos conceptos mediante un bloque `lock`.]

La clase `BufereSincronizado` (figura 15.9) implementa la interfaz `Bufere` y contiene dos variables de instancia: `bufere` (línea 10) y `cuentaBufereOcupado` (línea 13). Además, los descriptores de acceso `get` (líneas 18-61) y `set` (líneas 62-93) ahora utilizan métodos de la clase `Monitor` para sincronizar el acceso a la variable de instancia `bufere`. Por ende, cada objeto de la clase `BufereSincronizado` tiene un `SyncBlock` para mantener la sincronización. La variable de instancia `cuentaBufereOcupado` se conoce como variable de condición; los descriptores de acceso de la propiedad `Bufere` utilizan este valor `int` en condiciones para determinar si es el turno del productor o del consumidor para realizar una tarea. Si `cuentaBufereOcupado` es 0, el descriptor de acceso `set` de la propiedad `Bufere` puede colocar un valor en la variable `bufere`, ya que no contiene información todavía; pero esto significa que el descriptor de acceso `get` de la propiedad `Bufere` no puede leer el valor de `bufere`. Si `cuentaBufereOcupado` es 1, el descriptor de acceso `get` de la propiedad `Bufere` puede leer un valor de la variable `bufere`, debido a que la variable sí contiene información, pero el descriptor de acceso `set` de la propiedad `Bufere` no puede colocar un valor en `bufere`.

```
1 // Fig. 15.9: BufereSincronizado.cs
2 // Implementación de un búfer compartido sincronizado.
3 using System;
4 using System.Threading;
5
```

Figura 15.9 | `BufereSincronizado` sincroniza el acceso a un entero compartido. (Parte 1 de 3).


```

6 // esta clase representa a un solo valor int compartido
7 public class BuferSincronizado : Bufer
8 {
9     // búfer compartido por los subprocesos productor y consumidor
10    private int bufer = -1;
11
12    // cuentaBuferOcupado mantiene la cuenta de los búferes ocupados
13    private int cuentaBuferOcupado = 0;
14
15    // propiedad Bufer
16    public int Bufer
17    {
18        get
19        {
20            // obtiene el bloqueo sobre este objeto
21            Monitor.Enter( this );
22
23            // si no hay datos para leer, coloca el subproceso
24            // invocador en el estado WaitSleepJoin
25            if ( cuentaBuferOcupado == 0 )
26            {
27                Console.WriteLine(
28                    Thread.CurrentThread.Name + " trata de leer." );
29                MostrarEstado( "Búfer vacío. " +
30                    Thread.CurrentThread.Name + " espera." );
31                Monitor.Wait( this ); // entra al estado WaitSleepJoin
32            } // fin de if
33
34            // indica que el productor puede almacenar otro valor
35            // ya que el consumidor está a punto de extraer un valor del búfer
36            --cuentaBuferOcupado;
37
38            MostrarEstado( Thread.CurrentThread.Name + " lee " + bufer );
39
40            // indica al subproceso en espera (si lo hay) que debe
41            // prepararse para ejecutarse (estado Running)
42            Monitor.Pulse( this );
43
44            // Obtiene copia de búfer antes de liberar el bloqueo.
45            // Es posible que al productor se le pudiera
46            // asignar el procesador inmediatamente después de que
47            // se libere el monitor y antes de que se ejecute la
48            // instrucción de retorno. En este caso, el productor
49            // asignaría un nuevo valor al búfer antes de que
50            // la instrucción de retorno devuelva el valor al
51            // consumidor. Por ende, el consumidor recibiría el
52            // nuevo valor. Al hacer una copia de búfer y
53            // devolver la copia se asegura que el
54            // consumidor reciba el valor apropiado.
55            int copiaBufer = bufer;
56
57            // libera el bloqueo sobre este objeto
58            Monitor.Exit( this );
59
60            return copiaBufer;
61        } // fin de get
62        set
63        {

```

Figura 15.9 | BuferSincronizado sincroniza el acceso a un entero compartido. (Parte 2 de 3).

```

64      // adquiere el bloqueo para este objeto
65      Monitor.Enter( this );
66
67      // si no hay ubicaciones vacías, coloca el subproceso
68      // invocador en el estado WaitSleepJoin
69      if ( cuentaBuferOcupado == 1 )
70      {
71          Console.WriteLine(
72              Thread.CurrentThread.Name + " trata de escribir." );
73          MostrarEstado( "Búfer lleno. " +
74              Thread.CurrentThread.Name + " espera." );
75          Monitor.Wait( this );// entra al estado WaitSleepJoin
76      } // fin de if
77
78      // establece nuevo valor de búfer
79      bufer = value;
80
81      // indica que el consumidor puede extraer otro valor
82      // ya que el productor acaba de almacenar un valor en el búfer
83      ++cuentaBuferOcupado;
84
85      MostrarEstado( Thread.CurrentThread.Name + " escribe " + bufer );
86
87      // indica al subproceso en espera (si lo hay) que
88      // se prepare para ejecutarse (estado Running)
89      Monitor.Pulse( this );
90
91      // libera el bloqueo sobre este objeto
92      Monitor.Exit( this );
93  } // fin de set
94 } // fin de la propiedad Bufer
95
96 // muestra la operación actual y el estado del búfer
97 public void MostrarEstado( string operacion )
98 {
99     Console.WriteLine( "{0,-35}{1,-9}{2}\n",
100         operacion, bufer, cuentaBuferOcupado );
101 } // fin del método MostrarEstado
102 } // fin de la clase BuferSincronizado

```

Figura 15.9 | BuferSincronizado sincroniza el acceso a un entero compartido. (Parte 3 de 3).

Al igual que en el ejemplo anterior, el subproceso productor realiza las tareas especificadas en el método `Producir` del objeto productor. Cuando la línea 28 de la figura 15.5 establece el valor de la propiedad `Bufer` de `ubicacionCompartida`, el subproceso productor invoca al descriptor de acceso `set` en las líneas 62-93 de la figura 15.9. La línea 65 invoca al método `Enter` de `Monitor` con el argumento `this` para adquirir el bloqueo sobre el objeto `BuferSincronizado`. La instrucción `if` (líneas 69-76) determina si `cuentaBuferOcupado` es 1. Si esta condición es `true`, las líneas 71-72 imprimen en pantalla un mensaje indicando que el subproceso productor está tratando de escribir un valor, y las líneas 73-74 invocan al método `MostrarEstado` (líneas 97-101) para imprimir en pantalla otro mensaje indicando que el búfer está lleno, y que el subproceso productor espera. La línea 75 invoca al método `Wait` de `Monitor` con el argumento `this`, para colocar el subproceso que está haciendo la llamada (es decir, el productor) en el estado *WaitSleepJoin* para el objeto `BuferSincronizado`. Esto también *libera el bloqueo* sobre el objeto `BuferSincronizado`. El estado *WaitSleepJoin* para un objeto se mantiene mediante el `SyncBlock` de ese objeto. Ahora, otro subproceso puede invocar a un método descriptor de acceso de la propiedad `Bufer` del objeto `BuferSincronizado`.

El subproceso productor permanece en el estado *WaitSleepJoin* hasta que la llamada del consumidor al método `Pulse` de `Monitor` le notifique que puede proceder; en este punto el subproceso regresa al estado *Running* y

espera a que el sistema le asigne un procesador. Cuando el subproceso regresa al estado *Running*, el subproceso vuelve a adquirir en forma implícita el bloqueo sobre el objeto *BufeSincronizado*, y el descriptor de acceso *set* continúa ejecutándose con la siguiente instrucción después de *Wait*. La línea 79 asigna *value* a *bufe*. La línea 83 incrementa el valor de *cuentaBufeOcupado* para indicar que el búfer compartido ahora contiene un valor (es decir, un consumidor puede leer el valor, y un productor no puede todavía colocar otro valor ahí). La línea 85 invoca al método *MostrarEstado* para mostrar en pantalla una línea para la ventana de consola, indicando que el productor está escribiendo un nuevo valor en el búfer. La línea 89 invoca al método *Pulse* de *Monitor* con el objeto *BufeSincronizado* (*this*) como argumento. Si hay subprocesos en espera en el *SyncBlock* de ese objeto, el primer subproceso en espera entra al estado *Running*, indicando que ahora puede tratar de realizar su tarea de nuevo (tan pronto como se le asigne un procesador). El método *Pulse* regresa de inmediato. La línea 92 invoca al método *Exit* de *Monitor* para liberar el bloqueo sobre el objeto *BufeSincronizado*, y el descriptor de acceso *set* regresa al método que lo llamó (es decir, el método *Producir* del *Productor*).



Error común de programación 15.4

Olvidar liberar el bloqueo sobre un objeto cuando ya no se necesita es un error lógico. Esto evitará que los subprocesos en su programa adquieran el bloqueo para proceder con sus tareas. Estos subprocesos se verán obligados a esperar (sin necesidad, ya que el bloqueo no se necesita más). Dicha espera puede producir un interbloqueo y un aplazamiento indefinido.

Los descriptors de acceso *get* y *set* se implementan de manera similar. Como en el ejemplo anterior, el subproceso consumidor realiza las tareas especificadas en el método *Consumir* del objeto consumidor. El subproceso consumidor obtiene el valor de la propiedad *Bufe* del objeto *BufeSincronizado* (línea 30 de la figura 15.6) mediante una invocación al descriptor de acceso *get* en las líneas 18-61 de la figura 15.9. La línea 21 invoca al método *Enter* de *Monitor* para adquirir el bloqueo sobre el objeto *BufeSincronizado*.

La instrucción *if* en las líneas 25-32 determina si *cuentaBufeOcupado* es 0. Si esta condición es *true*, las líneas 27-28 imprimen en pantalla un mensaje indicando que el subproceso consumidor está tratando de leer un valor, y las líneas 29-30 invocan al método *MostrarEstado* para imprimir otro mensaje indicando que el búfer está vacío y que el subproceso consumidor espera. La línea 31 invoca al método *Wait* de *Monitor* con el argumento *this* para colocar al subproceso que hizo la llamada (es decir, el consumidor) en el estado *WaitSleepJoin* para el objeto *BufeSincronizado*, y libera el bloqueo sobre el objeto. Ahora, otro subproceso puede invocar a un método descriptor de acceso de la propiedad *Bufe* del objeto *BufeSincronizado*.

El objeto subproceso consumidor permanece en el estado *WaitSleepJoin* hasta que la llamada del productor al método *Pulse* de *Monitor* le notifique que puede proceder; en ese punto el subproceso regresa al estado *Running* y espera a que el sistema le asigne un procesador. Cuando el subproceso vuelve a entrar al estado *Running*, adquiere otra vez de forma implícita el bloqueo sobre el objeto *BufeSincronizado* y el descriptor de acceso *get* continúa ejecutándose, con la siguiente instrucción después de *Wait*. La línea 36 decrementa *cuentaBufeOcupado* para indicar que el búfer compartido ahora está vacío (es decir, un consumidor no puede leer el valor, pero un productor puede colocar otro valor en el búfer compartido), la línea 38 imprime una línea en la ventana de consola indicando el valor que está leyendo el consumidor, y la línea 42 invoca al método *Pulse* de *Monitor*, con el objeto *BufeSincronizado* como argumento. Si hay subprocesos en espera en el *SyncBlock* de ese objeto, el primer subproceso en espera entra al estado *Running*, indicando que el subproceso ahora puede tratar de realizar su tarea otra vez (tan pronto como se le asigne un procesador). El método *Pulse* regresa de inmediato. La línea 55 obtiene una copia de *Bufe* antes de liberar el bloqueo. Esto es necesario, ya que es posible que al productor se le pudiera asignar el procesador justo después de liberar el bloqueo (línea 58) y antes de que se ejecute la instrucción *return* (línea 60). En este caso, el productor asignaría un nuevo valor a *bufe* antes de que la instrucción *return* devuelva el valor al consumidor, y éste recibe el nuevo valor. Al crear una copia de *bufe* y devolver esta copia, se asegura que el consumidor reciba el valor apropiado. La línea 58 invoca al método *Exit* de *Monitor* para liberar el bloqueo en el objeto *BufeSincronizado*, y el descriptor de acceso *get* devuelve *copiaBufe* al método que lo llamó.

La clase *PruebaBufeSincronizado* (figura 15.10) es casi idéntica a la clase *PruebaBufeSinSincronizacion* (figura 15.8). El método *Main* de *PruebaBufeSincronizado* declara a *compartido* como un objeto de la clase *BufeSincronizado* (línea 14) y también muestra la información de encabezado para los resultados (líneas 20-22).

Estudie los dos conjuntos de resultados de ejemplo en la figura 15.10. Observe que cada entero producido se consume sólo una vez; no se pierden valores y no se consumen valores más de una vez. Esto ocurre debido a que el productor y el consumidor no pueden realizar tareas, a menos que sea “su turno”. El productor debe ejecutarse primero;

```

1  // Fig. 15.10: PruebaBufersincronizado.cs
2  // Muestra varios subprocesos que modifican un objeto compartido con
3  // sincronización.
4  using System;
5  using System.Threading;
6
7  // esta clase crea los subprocesos productor y consumidor
8  class PruebaBufersincronizado
9  {
10     // crea los subprocesos productor y consumidor, y los inicia
11     static void Main( string[] args )
12     {
13         // crea el objeto compartido utilizado por los subprocesos
14         Bufersincronizado compartido = new Bufersincronizado();
15
16         // Objeto aleatorio utilizado por cada subproceso
17         Random aleatorio = new Random();
18
19         // imprime los encabezados de las columnas y el estado inicial del búfer
20         Console.WriteLine( "{0,-35}{1,-9}{2}\n",
21             "Operación", "Búfer", "Cuenta ocupado" );
22         compartido.MostrarEstado( "Estado inicial" );
23
24         // crea los objetos Productor y Consumidor
25         Productor productor = new Productor( compartido, aleatorio );
26         Consumidor consumidor = new Consumidor( compartido, aleatorio );
27
28         // crea los subprocesos productor y consumidor, y establece
29         // delegados para cada subproceso
30         Thread subprocesoProductor =
31             new Thread( new ThreadStart( productor.Producir ) );
32         subprocesoProductor.Name = "Productor";
33
34         Thread subprocesoConsumidor =
35             new Thread( new ThreadStart( consumidor.Consumir ) );
36         subprocesoConsumidor.Name = "Consumidor";
37
38         // inicia cada subproceso
39         subprocesoProductor.Start();
40         subprocesoConsumidor.Start();
41     } // fin de Main
42 } // fin de la clase PruebaBufersincronizado

```

Operación	Búfer	Cuenta ocupado
Estado inicial	-1	0
Productor escribe 1	1	1
Consumidor lee 1	1	0
Consumidor trata de leer. Búfer vacío. Consumidor espera.	1	0
Productor escribe 2	2	1
Consumidor lee 2	2	0

Figura 15.10 | Los subprocesos productor y consumidor acceden a un objeto compartido con sincronización. (Parte I de 3).

Productor escribe 3	3	1
Productor trata de escribir. Búfer lleno. Productor espera.	3	1
Consumidor lee 3	3	0
Productor escribe 4	4	1
Consumidor lee 4	4	0
Consumidor trata de leer. Búfer vacío. Consumidor espera.	4	0
Productor escribe 5	5	1
Consumidor lee 5	5	0
Productor escribe 6	6	1
Consumidor lee 6	6	0
Productor escribe 7	7	1
Consumidor lee 7	7	0
Productor escribe 8	8	1
Productor trata de escribir. Búfer lleno. Productor espera.	8	1
Consumidor lee 8	8	0
Productor escribe 9	9	1
Consumidor lee 9	9	0
Consumidor trata de leer. Búfer vacío. Consumidor espera.	9	0
Productor escribe 10	10	1
Productor terminó de producir. Terminando productor.		
Consumidor lee 10	10	0
Consumidor leyó valores para un total de: 55. Terminando Consumidor.		
Operación	Búfer	Cuenta ocupado
Estado inicial	-1	0
Consumidor trata de leer. Búfer vacío. Consumidor espera.	-1	0

Figura 15.10 | Los subprocesos productor y consumidor acceden a un objeto compartido con sincronización. (Parte 2 de 3).

Productor escribe 1	1	1
Consumidor lee 1	1	0
Consumidor trata de leer. Búfer vacío. Consumidor espera.	1	0
Productor escribe 2	2	1
Consumidor lee 2	2	0
Productor escribe 3	3	1
Consumidor lee 3	3	0
Productor escribe 4	4	1
Productor trata de escribir. Búfer lleno. Productor espera.	4	1
Consumidor lee 4	4	0
Productor escribe 5	5	1
Productor trata de escribir. Búfer lleno. Productor espera.	5	1
Consumidor lee 5	5	0
Productor escribe 6	6	1
Consumidor lee 6	6	0
Productor escribe 7	7	1
Consumidor lee 7	7	0
Productor escribe 8	8	1
Consumidor lee 8	8	0
Consumidor trata de leer. Búfer vacío. Consumidor espera.	8	0
Productor escribe 9	9	1
Consumidor lee 9	9	0
Consumidor trata de leer. Búfer vacío. Consumidor espera.	9	0
Productor escribe 10	10	1
Consumidor lee 10	10	0
Productor terminó de producir. Terminando Productor. Consumidor leyó valores para un total de: 55. Terminando Consumidor.		

Figura 15.10 | Los subprocesos productor y consumidor acceden a un objeto compartido con sincronización.
(Parte 3 de 3).

el consumidor debe esperar si el productor no ha producido nada desde la última vez que consumió; y el productor debe esperar si el consumidor no ha consumido todavía el valor que el productor produjo más recientemente. Ejecute este programa varias veces para confirmar que cada entero producido se consuma sólo una vez. Observe las líneas que indican cuándo deben esperar el productor y el consumidor para realizar sus tareas respectivas.

15.8 Relación productor/consumidor: búfer circular

Las figuras 15.9 y 15.10 utilizan la sincronización de subprocesos para garantizar que dos subprocesos manipulen correctamente los datos en un búfer compartido. Sin embargo, la aplicación tal vez no tenga un rendimiento óptimo. Si los dos subprocesos operan a distintas velocidades, uno invertirá más (o la mayor parte del) tiempo esperando. Por ejemplo, en la figura 15.9 compartimos un entero individual entre los dos subprocesos. Si el subproceso productor produce valores más rápido de lo que el consumidor puede consumirlos, entonces el subproceso productor debe esperar al consumidor, ya que no hay más ubicaciones en memoria para colocar el siguiente valor. De manera similar, si el consumidor consume con más rapidez de la que el productor puede producir valores, el consumidor espera hasta que el productor coloque el siguiente valor en la ubicación compartida de memoria. Incluso cuando tenemos subprocesos que operan a las mismas velocidades relativas, al transcurrir un periodo de tiempo esos subprocesos pueden quedar “fuera de sincronización”, lo que provoca que un subproceso espere al otro. No podemos hacer suposiciones en cuanto a las velocidades relativas de subprocesos que se ejecutan en forma concurrente. Hay demasiadas interacciones que ocurren con el sistema operativo, la red, el usuario y otros componentes, lo cual puede provocar que los subprocesos operen a distintas velocidades. Cuando esto ocurre, los subprocesos esperan. Cuando los subprocesos esperan, los programas se vuelven menos productivos, los programas que interactúan con el usuario responden menos y las aplicaciones de red sufren retrasos mayores, debido a que el procesador no se utiliza con eficiencia.

Para minimizar la espera de los subprocesos que comparten recursos y operan a las mismas velocidades relativas, podemos implementar un búfer circular que proporcione ubicaciones adicionales en las que el productor pueda colocar valores (en caso de que se “adelante” al consumidor), y de las que el consumidor pueda obtener esos valores (si se “pone al corriente” con el productor). Supondremos que el búfer se implementa como un arreglo. El productor y el consumidor trabajan a partir del inicio del arreglo. Cuando cualquiera de los subprocesos llega al final del arreglo, simplemente regresa al inicio del mismo para realizar su siguiente tarea. Si el productor produce temporalmente valores más rápido de lo que el consumidor puede consumirlos, el productor puede escribir valores adicionales en los búferes extra (si hay celdas disponibles; en caso contrario, el consumidor debe, una vez más, esperar). Esto permite al productor realizar su tarea, aun cuando el consumidor no esté listo para recibir el valor que se está produciendo. De manera similar, si el consumidor consume más rápido de lo que el productor produce nuevos valores, el consumidor puede leer valores adicionales del búfer (si los hay; en caso contrario, el consumidor debe, una vez más, esperar) y, por lo tanto, “ponerse al corriente” con el productor. Esto permite al consumidor realizar su tarea, a pesar de que el productor no esté listo para producir valores adicionales.

Hay que tener en cuenta que un búfer circular sería inapropiado si el productor y el consumidor operan a velocidades distintas. Si el consumidor siempre se ejecuta más rápido que el productor, entonces es suficiente tener un búfer con una sola ubicación, ya que las ubicaciones adicionales desperdiciarían memoria. Si el productor siempre se ejecuta más rápido, se requeriría un búfer con un número infinito de ubicaciones para absorber la producción adicional.

La clave para usar un búfer circular es definirlo con suficientes celdas adicionales como para poder manejar la producción “adicional” anticipada. Si, durante un periodo, determinamos que el productor a menudo produce cuando mucho tres valores más de los que el consumidor puede consumir, podemos definir un búfer de por lo menos tres celdas para manejar la producción adicional. No es conveniente que el búfer sea demasiado pequeño, ya que los subprocesos tendrían que esperar más. Por otro lado, tampoco es conveniente que el búfer sea demasiado grande, ya que se desperdiciaría memoria.



Tip de rendimiento 15.3

Aun cuando se utilice un búfer circular, es posible que un subproceso productor pudiera llenar el búfer, lo cual obligaría al subproceso productor a esperar hasta que un consumidor consuma un valor para liberar un elemento en el búfer. De manera similar, si el búfer está vacío, el subproceso consumidor debe esperar hasta que el productor produzca otro valor. La clave para usar un búfer circular es optimizar el tamaño del búfer, para minimizar la cantidad de tiempo de espera de los subprocesos.

Las figuras 15.11 y 15.12 demuestran cómo un productor y un consumidor acceden a un búfer circular (en este caso, un arreglo compartido de tres elementos) con sincronización. En esta versión de la relación productor/consumidor, el consumidor consume un valor sólo cuando el arreglo no está vacío, y el productor produce un valor sólo cuando el arreglo no está lleno. Este ejemplo reutiliza la interfaz `Bufere` (figura 15.4), las clases `Productor` (figura 15.5) y `Consumidor` (figura 15.6). Las instrucciones que crearon e iniciaron los objetos subproceso en los métodos `Main` de la clase `PruebaBufereSinSincronizacion` en la figura 15.8, y `PruebaBufereSincronizado` en la figura 15.10, ahora aparecen en la clase `PruebaBufereCircular` (figura 15.12).

Los cambios más importantes ocurren en la clase `BufereCircular` (figura 15.11), que ahora contiene cuatro variables de instancia. El arreglo `bufere` (línea 10) es un arreglo entero de tres elementos, que representa el búfer circular. La variable `cuentaBufereOcupado` es la variable de condición que puede usarse para determinar si un productor puede escribir en el búfer circular (es decir, si `cuentaBufereOcupado` es menor que el número de elementos en el arreglo `bufere`) y si un consumidor puede leer del búfer circular (es decir, si `cuentaBufereOcupado` es mayor que 0). La variable `ubicacionLectura` (línea 15) indica la posición desde la cual un consumidor puede leer el siguiente valor. La variable `ubicacionEscritura` (línea 16) indica la siguiente ubicación en la cual un productor puede colocar un valor.

El descriptor de acceso `set` (líneas 58-92) de la propiedad `Bufere` realiza las mismas tareas que en la figura 15.9, con unas cuantas modificaciones. En vez de usar los métodos `Enter` y `Exit` de `Monitor` para adquirir y liberar el bloqueo sobre el objeto `BufereCircular`, utilizamos un bloque de código precedido por la palabra clave `lock` para bloquear el objeto `BufereCircular`. Cuando el control del programa entra al bloque `lock`, el subproceso actual en ejecución adquiere el bloqueo (suponiendo que esté disponible en ese momento) sobre el objeto `BufereCircular` (es decir, `this`). Cuando el bloque `lock` termina, el subproceso libera el bloqueo en forma automática.



Error común de programación 15.5

Al utilizar los métodos `Enter` y `Exit` de la clase `Monitor` para manejar el bloqueo de un objeto, hay que llamar a `Exit` en forma explícita para liberar el bloqueo. Si ocurre una excepción en un método antes de poder llamar a `Exit`, y esa excepción no se atrapa, el método podría terminar sin llamarlo. De ser así, el bloqueo no se libera. Para evitar este error, coloque el código que podría lanzar excepciones dentro de un bloque `try`, coloque la llamada a `Exit` en el bloque `finally` correspondiente, para asegurar que se libere el bloqueo.



Observación de ingeniería de software 15.1

Al usar un bloque `lock` para manejar el bloqueo en un objeto sincronizado, se elimina la posibilidad de olvidar liberar el bloqueo con una llamada al método `Exit` de `Monitor`. C# llama en forma implícita al método `Exit` de `Monitor` cuando termina un bloque `lock` por cualquier razón. Por ende, aun cuando ocurra una excepción en el bloque, el bloqueo se liberará.

La instrucción `if` de las líneas 66-71 en el descriptor de acceso `set` determina si el productor debe esperar (es decir, si todos los búferes están llenos). De ser así, las líneas 68-69 imprimen texto indicando que el productor está esperando para realizar su tarea, y la línea 70 invoca al método `Wait` de `Monitor` para colocar el subproceso productor en el estado `WaitSleepJoin`. Cuando la ejecución continúa en la línea 74 después de la instrucción `if`, el valor del productor se coloca en el búfer circular, en la ubicación `ubicacionEscritura`. A continuación, las líneas 76-77 imprimen un mensaje en pantalla que contiene el valor producido. La línea 81 incrementa `cuentaBufereOcupado`, ya que ahora hay por lo menos un valor en el búfer para que lo lea el consumidor. Después, la línea 85 actualiza `ubicacionEscritura` para la siguiente llamada al descriptor de acceso `set` de la propiedad `Bufere`. La salida continúa en la línea 86, en donde se invoca al método `CrearSalidaEstado` (declarado en las líneas 96-135), el cual imprime en pantalla el número de búferes ocupados, el contenido de los búferes y los valores actuales de `ubicacionEscritura` y `ubicacionLectura`. Por último, la línea 90 invoca al método `Pulse` de `Monitor` para indicar que un subproceso en espera en el objeto `BufereCircular` (si hay un subproceso en espera) debe cambiar al estado `Running`. Observe que al llegar a la llave derecha de cierre del bloque `lock` en la línea 91, el subproceso libera el bloqueo sobre el objeto `BufereCircular`.

El descriptor de acceso `get` (líneas 21-57) de la propiedad `Bufere` también realiza las mismas tareas en este ejemplo que las que hizo en la figura 15.9, con unas cuantas modificaciones menores. Una vez más, utilizamos un bloque `lock` para adquirir y liberar el bloqueo sobre el objeto `BufereCircular`, en vez de usar los métodos `Enter` y `Exit` de `Monitor`. La instrucción `if` en las líneas 29-34 en el descriptor de acceso `get` determina si el


```

1  // Fig. 15.11: BuferCircular.cs
2  // Un búfer circular compartido para la relación productor/consumidor.
3  using System;
4  using System.Threading;
5
6  // implementa un arreglo de enteros compartidos con sincronización
7  public class BuferCircular : Bufer
8  {
9      // cada elemento en el arreglo es un búfer
10     private int[] buferes = { -1, -1, -1 };
11
12     // cuentaBuferOcupado mantiene la cuenta de búferes ocupados
13     private int cuentaBuferOcupado = 0;
14
15     private int ubicacionLectura = 0; // ubicación de la siguiente lectura
16     private int ubicacionEscritura = 0; // ubicación de la siguiente escritura
17
18     // propiedad Bufer
19     public int Bufer
20     {
21         get
22         {
23             // bloquea este objeto mientras obtiene el valor
24             // del arreglo buferes
25             lock ( this )
26             {
27                 // si no hay datos para leer, coloca el subproceso
28                 // invocador en el estado WaitSleepJoin
29                 if ( cuentaBuferOcupado == 0 )
30                 {
31                     Console.Write( "\nTodos los búferes vacíos. {0} espera.",
32                                   Thread.CurrentThread.Name );
33                     Monitor.Wait( this ); // entra al estado WaitSleepJoin
34                 } // fin de if
35
36                 // obtiene el valor en ubicacionLectura actual
37                 int valorLeido = buferes[ ubicacionLectura ];
38
39                 Console.Write( "\n{0} lee {1} ",
40                               Thread.CurrentThread.Name, buferes[ ubicacionLectura ] );
41
42                 // acaba de consumir un valor, por lo que se decrementa el
43                 // número de búferes ocupados
44                 --cuentaBuferOcupado;
45
46                 // actualiza ubicacionLectura para una operación futura de lectura,
47                 // después agrega el estado actual a la salida
48                 ubicacionLectura = ( ubicacionLectura + 1 ) % buferes.Length;
49                 Console.Write( CrearSalidaEstado() );
50
51                 // devuelve el subproceso en espera (si hay uno)
52                 // al estado Running
53                 Monitor.Pulse( this );
54
55                 return valorLeido;
56             } // fin de lock
57         } // fin de get
58         set

```

Figura 15.11 | BuferCircular sincroniza el acceso a un búfer circular que contiene tres ubicaciones de almacenamiento. (Parte 1 de 3).

```

59     {
60         // bloquea este objeto mientras establece el valor
61         // en el arreglo buferes
62         lock ( this )
63         {
64             // si no hay ubicaciones vacías, coloca el subproceso
65             // invocador en el estado WaitSleepJoin
66             if ( cuentaBuferOcupado == buferes.Length )
67             {
68                 Console.Write( "\nTodos los búferes llenos. {0} espera.",
69                     Thread.CurrentThread.Name );
70                 Monitor.Wait( this ); // entra al estado WaitSleepJoin
71             } // fin de if
72
73             // coloca el valor en ubicacionEscritura de buferes
74             buferes[ ubicacionEscritura ] = value;
75
76             Console.Write( "\n{0} escribe {1} ",
77                 Thread.CurrentThread.Name, buferes[ ubicacionEscritura ] );
78
79             // acaba de producir un valor, por lo que se incrementa el
80             // número de búferes ocupados
81             ++cuentaBuferOcupado;
82
83             // actualiza ubicacionEscritura para una operación futura de escritura,
84             // después agrega el estado actual a la salida
85             ubicacionEscritura = ( ubicacionEscritura + 1 ) % buferes.Length;
86             Console.Write( CrearSalidaEstado() );
87
88             // devuelve el subproceso en espera (si hay uno)
89             // al estado Running
90             Monitor.Pulse( this );
91         } // fin de lock
92     } // fin de set
93 } // fin de la propiedad Bufer
94
95 // crea salida de estado
96 public string CrearSalidaEstado()
97 {
98     // muestra primera línea de información de estado
99     string salida = "(búferes ocupados: " +
100         cuentaBuferOcupado + ")\nbúferes: ";
101
102     for ( int i = 0; i < buferes.Length; i++ )
103         salida += " " + string.Format( "{0,2}", buferes[ i ] ) + " ";
104
105     salida += "\n";
106
107     // muestra segunda línea de información de estado
108     salida += "          ";
109
110     for ( int i = 0; i < buferes.Length; i++ )
111         salida += "---- ";
112
113     salida += "\n";
114
115     // muestra tercera línea de información de estado
116     salida += "          ";

```

Figura 15.11 | BuferCircular sincroniza el acceso a un búfer circular que contiene tres ubicaciones de almacenamiento. (Parte 2 de 3).

```

117
118 // muestra indicadores de ubicacionLectura (R) y ubicacionEscritura (W)
119 // debajo de las ubicaciones de búfer apropiadas
120 for ( int i = 0; i < buferes.Length; i++ )
121 {
122     if ( i == ubicacionEscritura &&
123         ubicacionEscritura == ubicacionLectura )
124         salida += " WR ";
125     else if ( i == ubicacionEscritura )
126         salida += " W ";
127     else if ( i == ubicacionLectura )
128         salida += " R ";
129     else
130         salida += " ";
131 } // fin de for
132
133 salida += "\n";
134 return salida;
135 } // fin del método CrearSalidaEstado
136 } // fin de la clase BuferCircular

```

Figura 15.11 | BuferCircular sincroniza el acceso a un búfer circular que contiene tres ubicaciones de almacenamiento. (Parte 3 de 3).

consumidor debe esperar (es decir, todos los búferes están vacíos). Si el subproceso consumidor debe esperar, las líneas 31-32 indican que el consumidor está esperando realizar su tarea, y la línea 33 invoca al método `Wait` de `Monitor` para colocar el subproceso consumidor en el estado *WaitSleepJoin*. Cuando la ejecución continúa en la línea 37 después de la instrucción `if`, a `valorLeído` se le asigna el valor en la ubicación `ubicacionLectura` en el búfer circular. Las líneas 39-40 imprimen en pantalla el valor consumido. La línea 44 decreuenta la cuenta `BuferOcupado`, ya que el búfer ahora contiene una posición más en la que el subproceso productor puede colocar un valor. Después, la línea 48 actualiza `ubicacionLectura` para la siguiente llamada al descriptor de acceso `get` de `Bufer`. La línea 49 invoca al método `CrearSalidaEstado` para imprimir en pantalla el número de búferes ocupados, el contenido de los búferes y los valores actuales de `ubicacionEscritura` y `ubicacionLectura`. Por último, la línea 53 invoca al método `Pulse` para cambiar el siguiente subproceso que espera al objeto `BuferCircular` al estado *Running*, y la línea 55 devuelve el valor consumido al método que hizo la llamada.

En la figura 15.12, la línea 13 ahora declara a `compartido` como un objeto `BuferCircular`, y la línea 20 muestra el estado inicial del espacio del búfer compartido. Los resultados para este ejemplo incluyen el valor actual de `cuentaBuferOcupado`, el contenido de los búferes y el valor actual de `ubicacionEscritura` y `ubicacionLectura`. En la salida, las letras `W` y `R` representan el valor actual de `ubicacionEscritura` y `ubicacionLectura`, respectivamente. Observe que, después de colocar el tercer valor en el tercer elemento del búfer, el cuarto valor se inserta al inicio del arreglo. Esto proporciona el efecto del búfer circular.

```

1 // Fig. 15.12: PruebaBuferCircular.cs
2 // Implementa la relación productor/consumidor con un
3 // búfer circular.
4 using System;
5 using System.Threading;
6
7 class PruebaBuferCircular
8 {
9     // crea subprocesos productor y consumidor, y los inicia
10    static void Main( string[] args )
11    {
12        // crea el objeto compartido usado por los subprocesos

```

Figura 15.12 | Los subprocesos productor y consumidor acceden a un búfer circular. (Parte 1 de 4).

```

13   BufereCircular compartido = new BufereCircular();
14
15   // objeto aleatorio usado por cada subproceso
16   Random aleatorio = new Random();
17
18   // muestra estado compartido antes de que los subprocesos
19   // productor y consumidor empiecen a ejecutarse
20   Console.Write( compartido.CrearSalidaEstado() );
21
22   // crea objetos Productor y Consumidor
23   Productor productor = new Productor( compartido, aleatorio );
24   Consumidor consumidor = new Consumidor( compartido, aleatorio );
25
26   // crea subprocesos para productor y consumidor, y establece
27   // delegados para cada subproceso
28   Thread subprocesoProductor =
29       new Thread( new ThreadStart( productor.Producir ) );
30   subprocesoProductor.Name = "Productor";
31
32   Thread subprocesoConsumidor =
33       new Thread( new ThreadStart( consumidor.Consumir ) );
34   subprocesoConsumidor.Name = "Consumidor";
35
36   // inicia cada subproceso
37   subprocesoProductor.Start();
38   subprocesoConsumidor.Start();
39   } // fin de Main
40 } // fin de la clase PruebaBufereCircular

```

```

(buferes ocupados: 0)
buferes:  -1  -1  -1
-----
WR

Todos los búferes vacíos. Consumidor espera.
Productor escribe 1 (buferes ocupados: 1)
buferes:   1  -1  -1
-----
R   W

Consumidor lee 1 (buferes ocupados: 0)
buferes:   1  -1  -1
-----
WR

Productor escribe 2 (buferes ocupados: 1)
buferes:   1   2  -1
-----
R   W

Consumidor lee 2 (buferes ocupados: 0)
buferes:   1   2  -1
-----
WR

Todos los búferes vacíos. Consumidor espera.
Productor escribe 3 (buferes ocupados: 1)
buferes:   1   2   3
-----
W           R

```

Figura 15.12 | Los subprocesos productor y consumidor acceden a un búfer circular. (Parte 2 de 4).

```

Consumidor lee 3 (búferes ocupados: 0)
búferes:  1   2   3
          ----
          WR

Todos los búferes vacíos. Consumidor espera.
Productor escribe 4 (búferes ocupados: 1)
búferes:  4   2   3
          ----
          R   W

Productor escribe 5 (búferes ocupados: 2)
búferes:  4   5   3
          ----
          R       W

Consumidor lee 4 (búferes ocupados: 1)
búferes:  4   5   3
          ----
              R   W

Productor escribe 6 (búferes ocupados: 2)
búferes:  4   5   6
          ----
          W   R

Productor escribe 7 (búferes ocupados: 3)
búferes:  7   5   6
          ----
              WR

Todos los búferes llenos. Productor espera.
Consumidor lee 5 (búferes ocupados: 2)
búferes:  7   5   6
          ----
              W   R

Consumidor lee 6 (búferes ocupados: 1)
búferes:  7   5   6
          ----
          R   W

Productor escribe 8 (búferes ocupados: 2)
búferes:  7   8   6
          ----
          R       W

Consumidor lee 7 (búferes ocupados: 1)
búferes:  7   8   6
          ----
              R   W

Consumidor lee 8 (búferes ocupados: 0)
búferes:  7   8   6
          ----
              WR

Productor escribe 9 (búferes ocupados: 1)
búferes:  7   8   9
          ----
          W       R

```

Figura 15.12 | Los subprocesos productor y consumidor acceden a un búfer circular. (Parte 3 de 4).

```

Productor escribe 10 (búferes ocupados: 2)
búferes:  10    8    9
          ----
           W    R

Productor terminó de producir.
Terminando Productor.

Consumidor lee 9 (búferes ocupados: 1)
búferes:  10    8    9
          ----
           R    W

Consumidor lee 10 (búferes ocupados: 0)
búferes:  10    8    9
          ----
           WR

Consumidor leyó valores para un total de: 55.
Terminando Consumidor.

```

Figura 15.12 | Los subprocesos productor y consumidor acceden a un búfer circular. (Parte 4 de 4).

15.9 Subprocesamiento múltiple con GUIs

La naturaleza de la programación con subprocesamiento múltiple evita que usted sepa con exactitud cuándo se ejecutará un subproceso. Los componentes Windows Forms no son *a prueba de subprocesos*; si varios subprocesos manipulan un componente de la GUI de Windows, los resultados tal vez no sean correctos. Para asegurar que los subprocesos manipulen componentes de la GUI en forma segura, todas las interacciones con estos componentes deben realizarse mediante el *subproceso de interfaz de usuario* (también conocido como el *subproceso UI*): el subproceso que crea y mantiene la GUI. La clase `Control` proporciona el método ***Invoke*** para ayudar con este proceso. Este método especifica instrucciones de procesamiento de la GUI que el subproceso UI debe ejecutar. El método recibe como argumentos un delegado (`delegate`) que representa a un método que modificará la GUI, y un arreglo opcional de objetos `object` que representan los parámetros para el método. En algún punto después de llamar a `Invoke`, el subproceso UI ejecutará el método representado por el objeto `delegate`, y pasará el contenido del arreglo de objetos `object` como argumentos para el método.

Nuestro siguiente ejemplo (figuras 15.13 y 15.14) utiliza subprocesos separados para modificar el contenido que se muestra en una GUI de Windows. Este ejemplo también demuestra cómo utilizar la sincronización de subprocesos para *suspender* un subproceso (es decir, evitar temporalmente que se ejecute) y para *reanudar* un subproceso suspendido. La GUI para la aplicación contiene tres controles `Label` y tres controles `CheckBox`. Cada subproceso en el programa muestra caracteres aleatorios en un control `Label` específico. El usuario puede suspender temporalmente un subproceso, haciendo clic en el objeto `CheckBox` apropiado, y puede reanudar la ejecución del subproceso haciendo clic en el mismo objeto `CheckBox` otra vez.

La clase `LetrasAleatorias` (figura 15.13) contiene el método `GenerarCaracteresAleatorios` (líneas 33-58), el cual no recibe argumentos y devuelve `void`. La línea 36 utiliza la propiedad `static currentThread` de `Thread` para determinar el subproceso actual en ejecución, y después utiliza la propiedad `Name` del subproceso para obtener su nombre. A cada subproceso en ejecución se le asigna un nombre que incluye el número del subproceso en el método `Main` (consulte los resultados de la figura 15.14). Las líneas 38-57 son un ciclo infinito. [Nota: en capítulos anteriores, hemos dicho que los ciclos infinitos son una mala práctica de programación, ya que la aplicación nunca terminará. En este caso, el ciclo infinito se encuentra en un subproceso separado del subproceso principal. Cuando se cierra la ventana de la aplicación en este ejemplo, todos los subprocesos creados por el subproceso principal también se cierran, incluyendo los subprocesos (como éste) que ejecutan ciclos infinitos.] En cada iteración del ciclo, el subproceso permanece inactivo (`sleep`) durante un intervalo aleatorio, de 0 a 1 segundo (línea 41).

Cuando el subproceso despierta, la línea 43 bloquea (`lock`) este objeto `LetrasAleatorias`, por lo que podemos determinar si se ha suspendido el subproceso (es decir, si el usuario hizo clic en el control `CheckBox`

correspondiente). Las líneas 45-48 iteran mientras la variable `bool` `suspendido` permanezca en `true`. La línea 47 llama al método `Wait` de `Monitor` en este objeto `LetrasAleatorias` para liberar temporalmente el bloqueo y colocar el subproceso en el estado *WaitSleepJoin*. Cuando se llama al método `Pulse` en este subproceso (es decir, el usuario hace clic de nuevo en el control `CheckBox` correspondiente), regresa al estado *Running*. Si `suspendido` es `false`, el subproceso termina su ejecución. Si `suspendido` sigue siendo `true`, el ciclo se ejecuta otra vez y el subproceso regresa al estado *WaitSleepJoin*.

```

1  // Fig. 15.13: LetrasAleatorias.cs
2  // Escribe una letra aleatoria en una etiqueta
3  using System;
4  using System.Windows.Forms;
5  using System.Drawing;
6  using System.Threading;
7
8  public class LetrasAleatorias
9  {
10     private static Random generador = new Random(); // para las letras aleatorias
11     private bool suspendido = false; // true si se suspende el subproceso
12     private Label salida; // control Label para mostrar la salida
13     private string nombreSubproceso; // nombre del subproceso actual
14
15     // constructor de LetrasAleatorias
16     public LetrasAleatorias( Label etiqueta )
17     {
18         salida = etiqueta;
19     } // fin del constructor de LetrasAleatorias
20
21     // delegado que permite llamar al método MostrarCaracter
22     // en el subproceso que crea y mantiene la GUI
23     private delegate void DisplayDelegate( char mostrarChar );
24
25     // el método MostrarCaracter establece la propiedad Text del control Label
26     private void MostrarCaracter( char mostrarChar )
27     {
28         // imprime caracter en el control Label
29         salida.Text = nombreSubproceso + ": " + mostrarChar;
30     } // fin del método MostrarCaracter
31
32     // coloca caracteres aleatorios en la GUI
33     public void GenerarCaracteresAleatorios()
34     {
35         // obtiene el nombre del subproceso en ejecución
36         nombreSubproceso = Thread.CurrentThread.Name;
37
38         while ( true ) // ciclo infinito; se termina desde el exterior
39         {
40             // permanece inactivo hasta por 1 segundo
41             Thread.Sleep( generador.Next( 1001 ) );
42
43             lock ( this ) // obtiene el bloqueo
44             {
45                 while ( suspendido ) // itera hasta que no esté suspendido
46                 {
47                     Monitor.Wait( this ); // suspende la ejecución del subproceso
48                 } // fin de while
49             } // fin de lock
50         }

```

Figura 15.13 | La clase `LetrasAleatorias` imprime en pantalla letras aleatorias, y puede suspenderse. (Parte 1 de 2).

```

51      // selecciona la letra mayúscula aleatoria
52      char mostrarChar = ( char ) ( generador.Next( 26 ) + 65 );
53
54      // muestra el caracter en el control Label correspondiente
55      salida.Invoke( new DisplayDelegate( MostrarCaracter ),
56                    new object[] { mostrarChar } );
57  } // fin de while
58 } // fin del método GenerarCaracteresAleatorios
59
60 // cambia el estado suspended/running
61 public void Alterna()
62 {
63     suspendido = !suspendido; // alterna el estado de control bool
64
65     // cambia el color de la etiqueta en suspender/reanudar
66     salida.BackColor = suspendido ? Color.Red : Color.LightGreen;
67
68     lock ( this ) // obtiene el bloqueo
69     {
70         if ( !suspendido ) // si se reanudó el subproceso
71             Monitor.Pulse( this );
72     } // fin de lock
73 } // fin del método Alterna
74 } // fin de la clase LetrasAleatorias

```

Figura 15.13 | La clase LetrasAleatorias imprime en pantalla letras aleatorias, y puede suspenderse. (Parte 2 de 2).

La línea 52 genera un carácter aleatorio en mayúscula. Las líneas 55-56 llaman al método `Invoke` y le pasan un nuevo objeto `DisplayDelegate` que contiene el método `MostrarCaracter`, y un nuevo arreglo de objetos `object` que contiene la letra generada al azar. La línea 23 declara un tipo `delegate` llamado `DisplayDelegate`, el cual representa los métodos que reciben un argumento `char` y no devuelven un valor. El método `MostrarCaracter` (líneas 26-30) cumple esos requerimientos: recibe un parámetro `char` llamado `mostrarChar` y no devuelve un valor. La llamada a `Invoke` en las líneas 55-56 provocan que el subproceso UI llame a `MostrarCaracter` con la letra generada al azar como argumento. En ese momento, la línea 29 sustituirá el texto en el control `Label` asociado con este objeto `LetrasAleatorias` con el nombre del subproceso que ejecuta el método `GenerarCaracteresAleatorios` de este objeto `LetrasAleatorias` y la letra generada al azar.

Cuando el usuario hace clic en el control `CheckBox` a la derecha de un control `Label` específico, el subproceso correspondiente debe suspenderse (evitar su ejecución temporalmente) o reanudarse (permitir que siga ejecutándose). Las acciones de suspender y reanudar un subproceso pueden implementarse mediante el uso de la sincronización de subprocesos y los métodos `Wait` y `Pulse` de `Monitor`. Las líneas 61-73 declaran el método `Alterna`, el cual modifica el estado suspendido/reanudado del subproceso actual. La línea 63 invierte el valor de la variable `bool` `suspendido`. La línea 66 cambia el color de fondo del control `Label`, asignando un color a la propiedad `BackColor` de `Label`. Si el subproceso se suspende, el color de fondo será `Color.Red`. Si el subproceso se encuentra en ejecución, el color de fondo será `Color.LightGreen`. El método `Alterna` se llama desde el manejador de eventos en la figura 15.14, por lo que sus tareas se realizarán en el subproceso UI; por ende, no hay necesidad de utilizar `Invoke` para la línea 66. La línea 68 bloquea (mediante `lock`) este objeto `LetrasAleatorias`, por lo que podemos determinar si el subproceso debe continuar su ejecución. De ser así, la línea 71 llama al método `Pulse` en este objeto `LetrasAleatorias` para alertar al subproceso que se colocó en el estado *WaitSleepJoin* mediante la llamada al método `Wait` en la línea 47.

Observe que la instrucción `if` en la línea 70 no tiene un `else` asociado. Si esta condición falla, significa que el subproceso acaba de suspenderse. Cuando esto ocurre, un subproceso en ejecución en la línea 45 entrará al ciclo `while` y la línea 47 suspenderá al subproceso con una llamada al método `Wait`.

La clase `SubprocesosGUI` (figura 15.14) muestra tres controles `Label` y tres controles `CheckBox`. Un subproceso separado de ejecución se asocia con cada par de controles `Label` y `CheckBox`. Cada subproceso muestra letras del alfabeto al azar en su correspondiente objeto `Label`. Las líneas 21, 28 y 35 crean tres nuevos objetos

LetrasAleatorias. Las líneas 22-23, 29-30 y 36-37 crean tres nuevos subprocesos que ejecutarán los métodos GenerarCaracteresAleatorios de los objetos LetrasAleatorias. Las líneas 24, 31 y 38 asignan un nombre a cada subproceso y las líneas 25, 32 y 39 inician (Start) los subprocesos.

```

1  // Fig. 15.14: SubprocesosGUI.cs
2  // Demuestra el uso de subprocesos en una GUI
3  using System;
4  using System.Windows.Forms;
5  using System.Threading;
6
7  public partial class SubprocesosGUIForm : Form
8  {
9      public SubprocesosGUIForm()
10     {
11         InitializeComponent();
12     } // fin del constructor
13
14     private LetrasAleatorias letra1; // primer objeto LetrasAleatorias
15     private LetrasAleatorias letra2; // segundo objeto LetrasAleatorias
16     private LetrasAleatorias letra3; // tercer objeto LetrasAleatorias
17
18     private void SubprocesosGUIForm_Load( object sender, EventArgs e )
19     {
20         // crea el primer subproceso
21         letra1 = new LetrasAleatorias( subproceso1Label );
22         Thread primerSubproceso = new Thread(
23             new ThreadStart( letra1.GenerarCaracteresAleatorios ) );
24         primerSubproceso.Name = "Subproceso 1";
25         primerSubproceso.Start();
26
27         // crea el segundo subproceso
28         letra2 = new LetrasAleatorias( subproceso2Label );
29         Thread segundoSubproceso = new Thread(
30             new ThreadStart( letra2.GenerarCaracteresAleatorios ) );
31         segundoSubproceso.Name = "Subproceso 2";
32         segundoSubproceso.Start();
33
34         // crea el tercer subproceso
35         letra3 = new LetrasAleatorias( subproceso3Label );
36         Thread tercerSubproceso = new Thread(
37             new ThreadStart( letra3.GenerarCaracteresAleatorios ) );
38         tercerSubproceso.Name = "Subproceso 3";
39         tercerSubproceso.Start();
40     } // fin del método SubprocesosGUIForm_Load
41
42     // cierra todos los subprocesos asociados con esta aplicación
43     private void SubprocesosGUIForm_FormClosing( object sender,
44         FormClosingEventArgs e )
45     {
46         System.Environment.Exit( System.Environment.ExitCode );
47     } // fin del método SubprocesosGUIForm_FormClosing
48
49     // suspende o reanuda el subproceso correspondiente
50     private void subprocesoCheckBox_CheckedChanged( object sender,
51         EventArgs e )
52     {
53         if ( sender == subproceso1CheckBox )

```

Figura 15.14 | SubprocesosGUI demuestra el subprocesamiento múltiple en una aplicación de GUI. (Parte I de 2).

```

54     letra1.Alterna();
55     else if ( sender == subproceso2CheckBox )
56         letra2.Alterna();
57     else if ( sender == subproceso3CheckBox )
58         letra3.Alterna();
59 } // fin del método subprocesoCheckBox_CheckedChanged
60 } // fin de la clase SubprocesosGUIForm

```



Figura 15.14 | SubprocesosGUI demuestra el subprocesamiento múltiple en una aplicación de GUI. (Parte 2 de 2).

Si el usuario hace clic en el control **CheckBox** llamado **Suspendido** enseguida de un control **Label** específico, el manejador de eventos `subprocesoCheckBox_CheckedChanged` (líneas 50-59) determina cuál control **CheckBox** generó el evento y llama al método `Alterna` de su objeto `LetrasAleatorias` asociado para suspender o reanudar el subproceso.

Las líneas 43-47 definen el manejador de eventos `SubprocesosGUIForm_FormClosing`, el cual llama al método `Exit` de la clase `System.Environment` con la propiedad `ExitCode` como argumento. Esto hace que todos los demás subprocesos en la aplicación terminen. De otra forma, sólo el subproceso UI terminaría cuando el usuario cerrara la aplicación; `Subproceso1`, `Subproceso2` y `Subproceso3` continuarían ejecutándose para siempre.

15.10 Conclusión

En este capítulo aprendió acerca de las herramientas básicas del .NET Framework que le permiten especificar tareas concurrentes en sus programas. Vimos cómo crear subprocesos de ejecución mediante el uso de los delegados de las clases `Thread` y `ThreadStart`; ambas del espacio de nombres `System.Threading`.

Hablamos sobre diversas aplicaciones de la programación concurrente. En especial, aprendió acerca de los problemas que pueden surgir cuando varios subprocesos comparten los mismos datos. Le demostramos cómo sincronizar subprocesos mediante el uso de las herramientas de la clase `Monitor` para asegurar que varios subprocesos accedan a los datos y los manipulen de manera apropiada. También le mostramos cómo implementar datos compartidos como un búfer circular, para permitir que los subprocesos operen con más eficiencia.

Después, aprendió que los componentes de la GUI no son a prueba de subprocesos, por lo que todos los cambios a estos componentes deben realizarse en el subproceso de interfaz de usuario que crea y mantiene la GUI. Le mostramos cómo usar el método `Invoke` de `Control` y un delegado para permitir que un subproceso especifique tareas que debe realizar el subproceso de interfaz de usuario con los componentes de la GUI. Esto permitió que varios subprocesos modificaran los componentes de la GUI en forma segura. En el siguiente capítulo, aprenderá acerca de las herramientas de procesamiento de cadenas, caracteres y expresiones regulares del .NET Framework.

16

Cadenas, caracteres y expresiones regulares

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Crear y manipular objetos inmutables de cadenas de caracteres de la clase `string`.
- Crear y manipular objetos mutables de cadenas de caracteres de la clase `StringBuilder`.
- Manipular objetos tipo carácter de la estructura `Char`.
- Usar expresiones regulares en conjunto con las clases `Regex` y `Match`.

*El principal defecto
del Rey Enrique
era masticar pequeños
trozos de cuerda.*

—Hillaire Belloc

*La escritura vigorosa es concisa.
Una oración no debe contener
palabras innecesarias,
un párrafo no debe contener
oraciones innecesarias.*

—William Strunk, Jr.

*Hice esta carta más extensa
de lo usual, ya que no tengo
el tiempo para hacerla más corta.*

—Blaise Pascal

*La diferencia entre la palabra
casi correcta y la palabra correcta es
en realidad una cuestión extensa;
es la diferencia entre el insecto
del rayo y el rayo.*

—Mark Twain

Ni una sola palabra.

—Miguel de Cervantes

- 16.1 Introducción
- 16.2 Fundamentos de los caracteres y las cadenas
- 16.3 Constructores de `string`
- 16.4 Indexador de `string`, propiedad `Length` y método `CopyTo`
- 16.5 Comparación de objetos `string`
- 16.6 Localización de caracteres y subcadenas en objetos `string`
- 16.7 Extracción de subcadenas de objetos `string`
- 16.8 Concatenación de objetos `string`
- 16.9 Métodos varios de la clase `string`
- 16.10 La clase `StringBuilder`
- 16.11 Las propiedades `Length` y `Capacity`, el método `EnsureCapacity` y el indexador de la clase `StringBuilder`
- 16.12 Los métodos `Append` y `AppendFormat` de la clase `StringBuilder`
- 16.13 Los métodos `Insert`, `Remove` y `Replace` de la clase `StringBuilder`
- 16.14 Métodos de `char`
- 16.15 Simulación para barajar y repartir cartas
- 16.16 Expresiones regulares y la clase `Regex`
 - 16.16.1 Ejemplo de expresión regular
 - 16.16.2 Validación de la entrada del usuario con expresiones regulares
 - 16.16.3 Los métodos `Replace` y `Split` de `Regex`
- 16.17 Conclusión

16.1 Introducción

Este capítulo introduce las herramientas de procesamiento de cadenas y caracteres de la FCL, y muestra cómo utilizar expresiones regulares para buscar patrones en el texto. Las técnicas que presentamos en este capítulo pueden emplearse en editores de texto, procesadores de palabras, software de diseño de páginas, sistemas de tipografía computarizados y otros tipos de software para procesamiento de texto. En los capítulos anteriores presentamos algunas herramientas básicas de procesamiento de cadenas. En este capítulo, hablaremos con detalle sobre las herramientas de procesamiento de texto de la clase `string` y el tipo `Char`, del espacio de nombres `System`, y de la clase `StringBuilder` del espacio de nombres `System.Text`.

Empezaremos con las generalidades sobre los fundamentos de los caracteres y las cadenas, en donde hablaremos sobre las literales de cadena y las constantes tipo carácter. Después proporcionaremos diversos ejemplos de constructores y métodos de la clase `String`. Los ejemplos muestran cómo trabajar con las cadenas, es decir, determinar su longitud, copiarlas, acceder a caracteres individuales, buscar, obtener subcadenas a partir de cadenas más grandes, comparar cadenas, concatenarlas, sustituir caracteres en cadenas y convertirlas a letras mayúsculas o minúsculas.

Luego introduciremos la clase `StringBuilder`, que se utiliza para crear cadenas en forma dinámica. Demostraremos las herramientas de `StringBuilder` para determinar y especificar el tamaño de un objeto `StringBuilder`, así como anexas, insertar, eliminar y sustituir caracteres en un objeto `StringBuilder`. Después introduciremos los métodos de prueba de caracteres de la estructura `Char`, que permiten que un programa determine si un carácter es un dígito, una letra, una letra minúscula, mayúscula, un signo de puntuación o un símbolo distinto. Tales métodos son útiles para validar caracteres individuales en la entrada del usuario. Además, el tipo `Char` cuenta con métodos para convertir un carácter en letra mayúscula o minúscula.

El capítulo concluye con una discusión acerca de las expresiones regulares. Hablaremos sobre las clases `Regex` y `Match` del espacio de nombres `System.Text.RegularExpressions`, así como de los símbolos que se utilizan para formar expresiones regulares. Luego demostraremos cómo buscar patrones en una cadena, relacionar cadenas completas con los patrones, sustituir caracteres en una cadena que concuerden con un patrón y dividir las cadenas en los delimitadores especificados como un patrón en una expresión regular.

16.2 Fundamentos de los caracteres y las cadenas

Los caracteres son los bloques fundamentales del código fuente de C#. Todo programa está compuesto de caracteres que, cuando se agrupan en modo significativo, crean una secuencia que el compilador interpreta como instrucciones que describen cómo realizar una tarea. Además de los caracteres normales, un programa puede contener también *constantes tipo carácter*. Esta constante es un carácter que se representa como un valor entero, conocido como *código de carácter*. Por ejemplo, el valor entero 122 corresponde a la constante tipo carácter 'z'. El valor entero 10 corresponde al carácter de nueva línea '\n'. Las constantes tipo carácter se establecen de acuerdo con el *conjunto de caracteres Unicode*, un conjunto internacional de caracteres que contiene muchos más símbolos y letras que el conjunto de caracteres ASCII (que se lista en el apéndice D). Para aprender más acerca de Unicode, consulte el apéndice E.

Una cadena es una serie de caracteres que se tratan como una unidad. Estos caracteres pueden ser letras mayúsculas, minúsculas, dígitos y varios *caracteres especiales*: +, -, *, /, \$ y otros. Una cadena es un objeto de la clase `string` en el espacio de nombres `System`.¹ Escribimos las *literales de cadena*, también conocidas como *constantes de cadena*, como secuencias de caracteres entre comillas dobles, como se muestra a continuación:

```
"John Q. Doe"
"9999 Main Street"
"Waltham, Massachusetts"
"(201) 555-1212"
```

Una declaración puede asignar una literal `string` a una referencia `string`. La declaración

```
string color = "azul";
```

inicializa la referencia `color` tipo `string` para que haga referencia al objeto literal `string` "azul".



Tip de rendimiento 16.1

Si hay varias ocurrencias del mismo objeto literal `string` en una aplicación, se hará referencia a una sola copia del objeto literal `string` desde cada ubicación en el programa que utilice esa literal. Es posible compartir el objeto de esta forma, ya que los objetos literales `string` son constantes de manera implícita. Ese tipo de compartición conserva la memoria.

En ocasiones, un objeto `string` contiene varios caracteres de barra diagonal inversa (esto ocurre con frecuencia en el nombre de un archivo). Para evitar un uso excesivo de los caracteres de barra diagonal inversa, es posible excluir secuencias de escape e interpretar a todos los caracteres en un objeto `string` de manera literal, usando el carácter @. Los caracteres de barra diagonal inversa dentro de las comillas dobles que van después del carácter @ no se consideran secuencias de escape, sino caracteres comunes de barra diagonal inversa. A menudo esto simplifica la programación y mejora la legibilidad del código. Por ejemplo, considere la cadena "C:\MiCarpeta\MiSubCarpeta\MiArchivo.txt" con la siguiente asignación:

```
string archivo = "C:\\MiCarpeta\\MiSubCarpeta\\MiArchivo.txt";
```

Usando la sintaxis de cadena textual, la asignación podría alterarse así:

```
string archivo = @"C:\MiCarpeta\MiSubCarpeta\MiArchivo.txt";
```

Este método tiene también la ventaja de permitir que las cadenas abarquen varias líneas, preservando todos los caracteres de nueva línea, espacios y tabuladores.

16.3 Constructores de string

La clase `string` cuenta con ocho constructores para inicializar objetos `string` de varias formas. La figura 16.1 demuestra el uso de tres de los constructores.

1. C# cuenta con la palabra clave `string` como un alias para la clase `String`. En este libro utilizamos el término `string`.

```

1  // Fig. 16.1: ConstructorString.cs
2  // Demostración de los constructores de la clase string.
3  using System;
4
5  class ConstructorString
6  {
7      public static void Main()
8      {
9          string stringOriginal, string1, string2,
10             string3, string4;
11          char[] arregloCaracteres =
12              { 'c', 'u', 'm', 'p', 'l', 'e', ' ', 'a', 'ñ', 'o', 's' };
13
14          // inicialización de las cadenas
15          stringOriginal = "¡Bienvenido a la programación en C#!";
16          string1 = stringOriginal;
17          string2 = new string( arregloCaracteres );
18          string3 = new string( arregloCaracteres, 7, 4 );
19          string4 = new string( 'C', 5 );
20
21          Console.WriteLine( "string1 = " + "\"" + string1 + "\"\n" +
22                             "string2 = " + "\"" + string2 + "\"\n" +
23                             "string3 = " + "\"" + string3 + "\"\n" +
24                             "string4 = " + "\"" + string4 + "\"\n" );
25      } // fin del método Main
26 } // fin de la clase ConstructorString

```

```

string1 = "¡Bienvenido a la programación en C#!"
string2 = "cumple años"
string3 = "años"
string4 = "CCCCC"

```

Figura 16.1 | Constructores de string.

Las líneas 9-10 declaran los objetos string `stringOriginal`, `string1`, `string2`, `string3` y `string4`. Las líneas 11-12 asignan el arreglo char llamado `arregloCaracteres`, el cual contiene once caracteres. La línea 15 asigna la literal string "¡Bienvenido a la programación en C#!" a la referencia string `stringOriginal`. La línea 16 establece `string1` para que haga referencia a la misma literal string.

La línea 17 asigna un nuevo objeto string a `string2`, usando el constructor de string que recibe un arreglo de caracteres como argumento. El nuevo objeto string contiene una copia de los caracteres en el arreglo `arregloCaracteres`.



Observación de ingeniería de software 16.1

En la mayoría de los casos, no es necesario crear una copia de un objeto string existente. Todos los objetos string son inmutables: su contenido de caracteres no puede modificarse una vez que se crean. Además, si hay una o más referencias a un objeto string (o a cualquier otro objeto, en lo que respecta), el recolector de basura no puede reclamar ese objeto.

La línea 18 asigna un nuevo objeto string a `string3`, usando el constructor de string que recibe un arreglo char y dos argumentos int. El segundo argumento especifica la posición del índice inicial (el *desplazamiento*), a partir de la cual se copiarán caracteres en el arreglo. El tercer argumento especifica el número de caracteres (la *cuenta*) a copiar desde la posición inicial especificada en el arreglo. El nuevo objeto string contiene una copia de los caracteres especificados en el arreglo. Si el desplazamiento o la cuenta especificados indican que el programa debe acceder a un elemento fuera de los límites del arreglo de caracteres, se lanza una excepción `ArgumentOutOfRangeException`.

La línea 19 asigna un nuevo objeto `string` a `string4`, usando el constructor de `string` que recibe como argumentos un carácter y un valor `int` que especifica el número de veces que se va a repetir el carácter en el objeto `string`.

16.4 Indexador de string, propiedad Length y método CopyTo

La aplicación de la figura 16.2 presenta el indexador de `string`, el cual facilita la extracción de cualquier carácter en el objeto `string`, y la propiedad `Length` de `string`, que devuelve la longitud del objeto `string`. El método `CopyTo` de `string` copia un número especificado de caracteres de un objeto `string` hacia un arreglo `char`.

Esta aplicación determina la longitud de un objeto `string`, muestra sus caracteres en orden inverso y copia una serie de caracteres del objeto `string` hacia un arreglo de caracteres.

La línea 20 usa la propiedad `Length` de `string` para determinar el número de caracteres en `string1`. Al igual que los arreglos, los objetos `string` siempre conocen su propio tamaño.

```

1  // Fig. 16.2: MetodosString.cs
2  // Uso del indexador, la propiedad Length y el método CopyTo
3  // de la clase string.
4  using System;
5
6  class MetodosString
7  {
8      public static void Main()
9      {
10         string string1;
11         char[] arregloCaracteres;
12
13         string1 = "hola a todos";
14         arregloCaracteres = new char[ 5 ];
15
16         // imprime string1
17         Console.WriteLine( "string1: \"\" + string1 + \"\" );
18
19         // prueba la propiedad Length
20         Console.WriteLine( " Longitud de string1: " + string1.Length );
21
22         // itera a través de los caracteres en string1 y los muestra invertidos
23         Console.Write( "La cadena invertida es: " );
24
25         for ( int i = string1.Length - 1; i >= 0; i-- )
26             Console.Write( string1[ i ] );
27
28         // copia caracteres de string1 hacia arregloCaracteres
29         string1.CopyTo( 0, arregloCaracteres, 0, arregloCaracteres.Length );
30         Console.Write( "\nEl arreglo de caracteres es: " );
31
32         for ( int i = 0; i < arregloCaracteres.Length; i++ )
33             Console.Write( arregloCaracteres[ i ] );
34
35         Console.WriteLine( "\n" );
36     } // fin del método Main
37 } // fin de la clase MetodosString

```

```

string1: "hola a todos"
Longitud de string1: 12
La cadena invertida es: sodot a aloh
El arreglo de caracteres es: hola

```

Figura 16.2 | El indexador de `string`, la propiedad `Length` y el método `CopyTo`.

Las líneas 25-26 escriben los caracteres de `string1` en orden inverso, usando el indexador de `string`. Este indexador trata a un objeto `string` como un arreglo de valores `char` y devuelve el carácter en una posición específica en el objeto `string`. El indexador recibe un argumento entero como el *número de posición* y devuelve el carácter en esa posición. Al igual que con los arreglos, el primer elemento de un objeto `string` se considera que está en la posición 0.



Error común de programación 16.1

Tratar de acceder a un carácter que está fuera de los límites de un objeto `string` (es decir, un índice menor que 0 o un índice mayor o igual que la longitud del objeto `string`) produce una excepción `IndexOutOfRangeException`.

La línea 29 usa el método `CopyTo` de `string` para copiar los caracteres de `string1` a un arreglo de caracteres (`arregloCaracteres`). El primer argumento que recibe el método `CopyTo` es el índice a partir del cual el método empieza a copiar caracteres en el objeto `string`. El segundo argumento es el arreglo de caracteres hacia el que se copiarán los caracteres. El tercer argumento es el índice que especifica la ubicación inicial en la que empieza el método a colocar los caracteres copiados en el arreglo de caracteres. El último argumento es el número de caracteres del objeto `string` que va a copiar el método. Las líneas 32-33 imprimen el contenido del arreglo `char`, un carácter a la vez.

16.5 Comparación de objetos `string`

Los siguientes dos ejemplos demuestran varios métodos para comparar objetos `string`. Para comprender cómo un objeto `string` puede ser “mayor que” o “menor que” otro objeto `string`, considere el proceso de alfabetizar una serie de apellidos. Sin duda, usted colocaría “Jones” antes de “Smith” ya que la primera letra de “Jones” va antes que la primera letra de “Smith” en el alfabeto. Este alfabeto es más que sólo un conjunto de 26 letras; es una lista ordenada de caracteres, en donde cada letra ocurre en una posición específica. Por ejemplo, Z es más que sólo una letra del alfabeto: específicamente, Z es la letra 26.

Las computadoras pueden ordenar caracteres en forma alfabética, debido a que se representan de manera interna como códigos numéricos de Unicode. Al comparar dos objetos `string`, C# sólo compara los códigos numéricos de los caracteres en los objetos `string`.

La clase `string` proporciona varias formas de comparar objetos `string`. La aplicación de la figura 16.3 demuestra el uso del método `Equals`, el método `CompareTo` y el operador de igualdad (`==`).

```

1  // Fig. 16.3: ComparacionString.cs
2  // Comparación de objetos string
3  using System;
4
5  class ComparacionString
6  {
7      public static void Main()
8      {
9          string string1 = "hola";
10         string string2 = "adiós";
11         string string3 = "Feliz Cumpleaños";
12         string string4 = "feliz cumpleaños";
13
14         // imprime los valores de los cuatro objetos string
15         Console.WriteLine( "string1 = \"" + string1 + "\"" +
16             "\nstring2 = \"" + string2 + "\"" +
17             "\nstring3 = \"" + string3 + "\"" +
18             "\nstring4 = \"" + string4 + "\"\n" );
19     }

```

Figura 16.3 | Prueba de objetos `string` para determinar la igualdad. (Parte 1 de 2).

```

20 // prueba la igualdad usando el método Equals
21 if ( string1.Equals( "hola" ) )
22     Console.WriteLine( "string1 es igual a \"hola\"" );
23 else
24     Console.WriteLine( "string1 no es igual a \"hola\"" );
25
26 // prueba la igualdad con ==
27 if ( string1 == "hola" )
28     Console.WriteLine( "string1 es igual a \"hola\"" );
29 else
30     Console.WriteLine( "string1 no es igual a \"hola\"" );
31
32 // prueba la igualdad comparando mayúsculas/minúsculas
33 if ( String.Equals( string3, string4 ) ) // método static
34     Console.WriteLine( "string3 es igual a string4" );
35 else
36     Console.WriteLine( "string3 no es igual a string4" );
37
38 // prueba con CompareTo
39 Console.WriteLine ( "\nstring1.CompareTo( string2 ) es " +
40     string1.CompareTo( string2 ) + "\n" +
41     "string2.CompareTo( string1 ) es " +
42     string2.CompareTo( string1 ) + "\n" +
43     "string1.CompareTo( string1 ) es " +
44     string1.CompareTo( string1 ) + "\n" +
45     "string3.CompareTo( string4 ) es " +
46     string3.CompareTo( string4 ) + "\n" +
47     "string4.CompareTo( string3 ) es " +
48     string4.CompareTo( string3 ) + "\n\n" );
49 } // fin del método Main
50 } // fin de la clase ComparacionString

```

```

string1 = "hola"
string2 = "adiós"
string3 = "Feliz Cumpleaños"
string4 = "feliz cumpleaños"

string1 es igual a "hola"
string1 es igual a "hola"
string3 no es igual a string4

string1.CompareTo( string2 ) es 1
string2.CompareTo( string1 ) es -1
string1.CompareTo( string1 ) es 0
string3.CompareTo( string4 ) es 1
string4.CompareTo( string3 ) es -1

```

Figura 16.3 | Prueba de objetos string para determinar la igualdad. (Parte 2 de 2).

La condición en la instrucción `if` (línea 21) utiliza el método `Equals` de `string` para comparar a `string1` y la literal `string "hola"`, para determinar si son iguales. El método `Equals` (heredado de `object` y redefinido en `string`) prueba la igualdad entre dos objetos cualesquiera (es decir, comprueba si los objetos tienen un contenido idéntico). El método devuelve `true` si los objetos son iguales, y `false` en caso contrario. En esta instancia, la condición anterior devuelve `true`, ya que `string1` hace referencia al objeto literal `string "hola"`. El método `Equals` utiliza una *comparación lexico-gráfica*: se comparan los valores Unicode enteros que representan a cada uno de los caracteres de cada objeto `string`. Una comparación del objeto `string "hola"` con el objeto `string "HOLA"` devolvería `false`, ya que las representaciones numéricas de las letras minúsculas son distintas de las representaciones numéricas de las letras mayúsculas.

La condición en la línea 27 utiliza el operador de igualdad (==) para comparar la igualdad entre el objeto `string string1` con la literal `string "hola"`. En C#, el operador de igualdad también utiliza una comparación lexicográfica para comparar dos objetos `string`. Por ende, la condición en la instrucción `if` se evalúa como `true`, ya que los valores de `string1` y `"hola"` son iguales.

Presentamos el texto para la igualdad de objetos `string` entre `string3` y `string4` (línea 33), para ilustrar que las comparaciones son, sin duda, sensibles al uso de mayúsculas. Aquí, el método `static Equals` se utiliza para comparar los valores de dos objetos `string`. "Feliz Cumpleaños" no es igual a "feliz cumpleaños", por lo que la condición de instrucción `if` falla, y se imprime en pantalla el mensaje "string3 no es igual a string4" (línea 36).

Las líneas 40- 48 usan el método `CompareTo` de `string` para comparar objetos `string`. El método `CompareTo` devuelve 0 si los objetos `string` son iguales, un valor negativo si el objeto `string` que invoca a `CompareTo` es menor que el que recibe como argumento, y un valor positivo si el objeto `string` que invoca a `CompareTo` es mayor que el que se pasa como argumento. El método `CompareTo` utiliza una comparación léxico-gráfica.

Observe que `CompareTo` considera que `string3` es mayor que `string4`. La única diferencia entre estos dos objetos es que `string3` contiene dos letras mayúsculas en posiciones en donde `string4` contiene letras minúsculas.

La aplicación en la figura 16.4 muestra cómo evaluar si una instancia de `string` empieza o termina con un objeto `string` dado. El método `StartsWith` determina si una instancia de `string` empieza con el texto `string` que recibe como argumento. El método `EndsWith` determina si una instancia de `string` termina con el texto `string` que recibe como argumento. El método `Main` de la clase `stringInicioFin` define un arreglo de objetos `string` (llamado `strings`), el cual contiene "iniciado", "iniciando", "terminado" y "terminando".

```

1  // Fig. 16.4: StringInicioFin.cs
2  // Demostración de los métodos StartsWith y EndsWith.
3  using System;
4
5  class StringInicioFin
6  {
7      public static void Main()
8      {
9          string[] strings =
10             { "iniciado", "iniciando", "terminado", "terminando" };
11
12         // evalúa cada objeto string, para ver si empieza con "in"
13         for ( int i = 0; i < strings.Length; i++ )
14             if ( strings[ i ].StartsWith( "in" ) )
15                 Console.WriteLine( "\"" + strings[ i ] + "\"" +
16                     " empieza con \"in\"" );
17
18         Console.WriteLine( "" );
19
20         // evalúa cada objeto string, para ver si termina con "ado"
21         for ( int i = 0; i < strings.Length; i++ )
22             if ( strings[ i ].EndsWith( "ado" ) )
23                 Console.WriteLine( "\"" + strings[ i ] + "\"" +
24                     " termina con \"ado\"" );
25
26         Console.WriteLine( "" );
27     } // fin del método Main
28 } // fin de la clase StringInicioFin

```

```

"iniciado" empieza con "in"
"iniciando" empieza con "in"

"iniciado" termina con "ado"
"terminado" termina con "ado"

```

Figura 16.4 | Los métodos `StartsWith` y `EndsWith`.

El resto del método `Main` evalúa los elementos del arreglo para determinar si inician o terminan con un conjunto específico de caracteres.

La línea 14 utiliza el método `StartsWith`, que recibe un argumento `string`. La condición en la instrucción `if` determina si el objeto `string` en el índice `i` del arreglo inicia con los caracteres `"in"`. Si es así, el método devuelve `true`, y se imprime `strings[i]` junto con un mensaje.

La línea 22 usa el método `EndsWith`, el cual también recibe un argumento `string`. La condición en la instrucción `if` determina si el objeto `string` en el índice `i` del arreglo termina con los caracteres `"ado"`. Si es así, el método devuelve `true`, y se muestra `strings[i]` junto con un mensaje.

16.6 Localización de caracteres y subcadenas en objetos string

En muchas aplicaciones, es necesario buscar un carácter o un conjunto de caracteres en un objeto `string`. Por ejemplo, un programador que crea un procesador de palabras desea proporcionar herramientas para buscar a través de los documentos. La aplicación en la figura 16.5 demuestra algunas de las diversas versiones de los métodos `IndexOf`, `IndexOfAny`, `LastIndexOf` y `LastIndexOfAny` de `string`, los cuales buscan un carácter o subcadena específica en un objeto `string`. En este ejemplo, realizamos todas las búsquedas en el objeto `string` `letras` (inicializado con `"abcdefghijklmabcdefghijklm"`), ubicado en el método `Main` de la clase `MetodosIndexString`.

Las líneas 14, 16 y 18 utilizan el método `IndexOf` para localizar la primera ocurrencia de un carácter o subcadena en un objeto `string`. Si encuentra un carácter, `IndexOf` devuelve el índice del carácter especificado en el objeto `string`; en caso contrario, `IndexOf` devuelve `-1`. La expresión en la línea 16 utiliza una versión del método `IndexOf` que recibe dos argumentos: el carácter a buscar y el índice inicial en donde debe empezar la búsqueda del objeto `string`. El método no examina caracteres que ocurran antes del índice inicial (en este caso, 1). La expresión en la línea 18 utiliza otra versión del método `IndexOf` que recibe tres argumentos: el carácter a buscar, el índice en el que se debe iniciar la búsqueda y el número de caracteres a buscar.

Las líneas 22, 24 y 26 utilizan el método `LastIndexOf` para localizar la última ocurrencia de un carácter en un objeto `string`. El método `LastIndexOf` realiza la búsqueda, desde el final del objeto `string` hasta el principio del mismo. Si encuentra el carácter, `LastIndexOf` devuelve el índice del carácter especificado en el objeto `string`; en caso contrario, `LastIndexOf` devuelve `-1`. Hay tres versiones de `LastIndexOf`. La expresión en la línea 22 utiliza la versión del método `LastIndexOf` que recibe como argumento el carácter que se va a buscar. La expresión en la línea 24 utiliza la versión del método `LastIndexOf` que recibe dos argumentos: el carácter que se va a buscar y el índice más alto a partir del cual se va a iniciar la búsqueda del carácter, de atrás hacia adelante. La expresión en la línea 26 utiliza una tercera versión del método `LastIndexOf` que recibe tres argumentos: el carácter que se va a buscar, el índice inicial a partir del cual se va a iniciar la búsqueda hacia atrás, y el número de caracteres (la porción del objeto `string`) a buscar.

Las líneas 29-44 utilizan versiones de `IndexOf` y `LastIndexOf` que reciben un objeto `string` en vez de un carácter como el primer argumento. Estas versiones de los métodos funcionan en forma idéntica a los antes descritos, con la excepción de que buscan secuencias de caracteres (o subcadenas) que se especifican mediante sus argumentos `string`.

```

1  // Fig. 16.5: MetodosIndexString.cs
2  // Uso de métodos para buscar en objetos string.
3  using System;
4
5  class MetodosIndexString
6  {
7      public static void Main()
8      {
9          string letras = "abcdefghijklmabcdefghijklm";
10         char[] letrasBusqueda = { 'c', 'a', '$' };
11
12         // prueba IndexOf para localizar un caracter en un objeto string

```

Figura 16.5 | Búsqueda de caracteres y subcadenas en objetos string. (Parte I de 3).

```

13 Console.WriteLine( "La primera 'c' se encuentra en el índice " +
14     letras.IndexOf( 'c' ) );
15 Console.WriteLine( "La primera 'a' que empieza en 1 se encuentra en el índice " +
16     letras.IndexOf( 'a', 1 ) );
17 Console.WriteLine( "El primer '$' en las 5 posiciones a partir de 3 " +
18     "se encuentra en el índice " + letras.IndexOf( '$', 3, 5 ) );
19
20 // prueba LastIndexOf para buscar un caracter en una cadena
21 Console.WriteLine( "\nLa última 'c' se encuentra en el índice " +
22     letras.LastIndexOf( 'c' ) );
23 Console.WriteLine( "La última 'a' hasta la posición 25 se encuentra en " +
24     "el índice " + letras.LastIndexOf( 'a', 25 ) );
25 Console.WriteLine( "El último '$' en las 5 posiciones a partir de 15 " +
26     "se encuentra en el índice " + letras.LastIndexOf( '$', 15, 5 ) );
27
28 // prueba IndexOf para localizar una subcadena en un objeto string
29 Console.WriteLine( "\nLa primera \"def\" se encuentra en el índice " +
30     letras.IndexOf( "def" ) );
31 Console.WriteLine( "La primera \"def\" a partir de 7 se encuentra en " +
32     "el índice " + letras.IndexOf( "def", 7 ) );
33 Console.WriteLine( "La primera \"hola\" en las 15 posiciones " +
34     "a partir de 5 se encuentra en el índice " +
35     letras.IndexOf( "hola", 5, 15 ) );
36
37 // prueba LastIndexOf para buscar una subcadena en un objeto string
38 Console.WriteLine( "\nLa última \"def\" se encuentra en el índice " +
39     letras.LastIndexOf( "def" ) );
40 Console.WriteLine( "La última \"def\" hasta la posición 25 se encuentra " +
41     "en el índice " + letras.LastIndexOf( "def", 25 ) );
42 Console.WriteLine( "La última \"hola\" en las 15 posiciones " +
43     "terminando en 20 se encuentra en el índice " +
44     letras.LastIndexOf( "hola", 20, 15 ) );
45
46 // prueba IndexOfAny para buscar la primera ocurrencia del carácter en el arreglo
47 Console.WriteLine( "\nLa primera 'c', 'a' o '$' se " +
48     "encuentra en el índice " + letras.IndexOfAny( letrasBusqueda ) );
49 Console.WriteLine( "La primera 'c', 'a' o '$' a partir de 7 se " +
50     "encuentra en el índice " + letras.IndexOfAny( letrasBusqueda, 7 ) );
51 Console.WriteLine( "La primera 'c', 'a' o '$' en las 5 posiciones " +
52     "a partir de 7 se encuentra en el índice " +
53     letras.IndexOfAny( letrasBusqueda, 7, 5 ) );
54
55 // prueba LastIndexOfAny para buscar la última ocurrencia del caracter
56 // en el arreglo
57 Console.WriteLine( "\nLa última 'c', 'a' o '$' se " +
58     "encuentra en el índice " + letras.LastIndexOfAny( letrasBusqueda ) );
59 Console.WriteLine( "La última 'c', 'a' o '$' hasta la posición 1 se " +
60     "encuentra en el índice " +
61     letras.LastIndexOfAny( letrasBusqueda, 1 ) );
62 Console.WriteLine( "La última 'c', 'a' o '$' en las 5 posiciones " +
63     "terminando en 25 se encuentra en el índice " +
64     letras.LastIndexOfAny( letrasBusqueda, 25, 5 ) );
65     } // fin del método Main
66 } // fin de la clase MetodosIndexString

```

La primera 'c' se encuentra en el índice 2
 La primera 'a' que empieza en 1 se encuentra en el índice 13
 El primer '\$' en las 5 posiciones a partir de 3 se encuentra en el índice -1

Figura 16.5 | Búsqueda de caracteres y subcadenas en objetos string. (Parte 2 de 3).

```

La última 'c' se encuentra en el índice 15
La última 'a' hasta la posición 25 se encuentra en el índice 13
El último '$' en las 5 posiciones a partir de 15 se encuentra en el índice -1

La primera "def" se encuentra en el índice 3
La primera "def" a partir de 7 se encuentra en el índice 16
La primera "hola" en las 15 posiciones a partir de 5 se encuentra en el índice -1

La última "def" se encuentra en el índice 16
La última "def" hasta la posición 25 se encuentra en el índice 16
La última "hola" en las 15 posiciones terminando en 20 se encuentra en el índice -1

La primera 'c', 'a' o '$' se encuentra en el índice 0
La primera 'c', 'a' o '$' a partir de 7 se encuentra en el índice 13
La primera 'c', 'a' o '$' en las 5 posiciones a partir de 7 se encuentra en el índice -1

La última 'c', 'a' o '$' se encuentra en el índice 15
La última 'c', 'a' o '$' hasta la posición 1 se encuentra en el índice 0
La última 'c', 'a' o '$' en las 5 posiciones terminando en 25 se encuentra en el índice -1

```

Figura 16.5 | Búsqueda de caracteres y subcadenas en objetos string. (Parte 3 de 3).

Las líneas 47-64 utilizan los métodos `IndexOfAny` y `LastIndexOfAny`, que reciben un arreglo de caracteres como el primer argumento. Estas versiones de los métodos también funcionan en forma idéntica a las antes descritas, con la excepción de que devuelven el índice de la primera ocurrencia de cualquiera de los caracteres en el argumento tipo arreglo de caracteres.



Error común de programación 16.2

En los métodos sobrecargados `LastIndexOf` y `LastIndexOfAny` que reciben tres parámetros, el segundo argumento debe ser mayor o igual que el tercero. Esto podría ser contraintuitivo, pero recuerde que la búsqueda avanza desde el final de la cadena, hasta el principio de la misma.

16.7 Extracción de subcadenas de objetos string

La clase `string` cuenta con dos métodos `Substring`, los cuales se utilizan para crear un nuevo objeto `string`, copiando parte de un objeto `string` existente. Cada método devuelve un nuevo objeto `string`. La aplicación en la figura 16.6 demuestra el uso de ambos métodos.

```

1 // Fig. 16.6: SubString.cs
2 // Demostración del método Substring de la clase string.
3 using System;
4
5 class SubString
6 {
7     public static void Main()
8     {
9         string letras = "abcdefghijklmabcdefghijklm";
10
11         // invoca al método Substring y le pasa un parámetro
12         Console.WriteLine("La subcadena del índice 20 al final es \"" +
13             letras.Substring( 20 ) + "\"");
14
15         // invoca al método Substring y le pasa dos parámetros

```

Figura 16.6 | Subcadenas generadas a partir de objetos string. (Parte 1 de 2).

```

16      Console.WriteLine( "La subcadena desde el índice 0 de longitud 6 es \"\" +
17          letras.Substring( 0, 6 ) + "\"\" );
18  } // fin del método Main
19  } // fin de la clase SubString

```

La subcadena del índice 20 al final es "hijklm"
 La subcadena desde el índice 0 de longitud 6 es "abcdef"

Figura 16.6 | Subcadenas generadas a partir de objetos `string`. (Parte 2 de 2).

La instrucción en la línea 13 utiliza el método `Substring` que recibe un argumento `int`. El argumento especifica el índice inicial, a partir del cual el método va a copiar los caracteres en el objeto `string` original. La subcadena devuelta contiene una copia de los caracteres, a partir del índice inicial, hasta el final del objeto `string`. Si el índice especificado en el argumento se encuentra fuera de los límites del objeto `string`, el programa lanza una excepción `ArgumentOutOfRangeException`.

La segunda versión del método `Substring` (línea 17) recibe dos argumentos `int`. El primer argumento especifica el índice inicial, a partir del cual el método copia los caracteres del objeto `string` original. El segundo argumento especifica la longitud de la subcadena a copiar. La subcadena devuelta contiene una copia de los caracteres especificados del objeto `string` original.

16.8 Concatenación de objetos `string`

El operador `+` (que vimos en el capítulo 3, Introducción a las aplicaciones de C#) no es la única forma de realizar la concatenación de objetos `string`. El método `static` `Concat` de la clase `string` (figura 16.7) concatena dos objetos `string` y devuelve un objeto `string` que contiene los caracteres combinados a partir de ambos objetos `string` originales. La línea 16 adjunta los caracteres de `string2` al final de una copia de `string1`, usando el método `Concat`. La instrucción en la línea 16 no modifica los objetos `string` originales.

```

1  // Fig. 16.7: SubConcatenacion.cs
2  // Demostración del método Concat de la clase string.
3  using System;
4
5  class SubConcatenacion
6  {
7      public static void Main( string[] args )
8      {
9          string string1 = "Feliz ";
10         string string2 = "Cumpleaños";
11
12         Console.WriteLine( "string1 = \"\" + string1 + "\"\n" +
13             "string2 = \"\" + string2 + "\"\" );
14         Console.WriteLine(
15             "\nResultado de string.Concat( string1, string2 ) = " +
16             string.Concat( string1, string2 ) );
17         Console.WriteLine( "string1 después de la concatenación = " + string1 );
18     } // fin del método Main
19 } // fin de la clase SubConcatenacion

```

string1 = "Feliz "
 string2 = "Cumpleaños"

Resultado de string.Concat(string1, string2) = Feliz Cumpleaños
 string1 después de la concatenación = Feliz

Figura 16.7 | El método `static` `Concat`.

16.9 Métodos varios de la clase string

La clase `string` proporciona varios métodos que devuelven copias modificadas de objetos `string`. La aplicación en la figura 16.8 demuestra el uso de estos métodos, incluyendo los métodos `Replace`, `ToLower`, `ToUpper` y `Trim` de `string`.

La línea 21 utiliza el método `Replace` de `string` para devolver un nuevo objeto `string`, sustituyendo todas las ocurrencias en `string1` del carácter 'u' con 'U'. El método `Replace` recibe dos argumentos: un objeto `string` en el que se va a buscar y otro objeto `string` con el que se van a sustituir todas las ocurrencias del primer argumento. El objeto `string` original permanece sin cambios. Si no hay ocurrencias del primer argumento en el objeto `string`, el método devuelve el objeto `string` original.

```

1  // Fig. 16.8: MetodosString2.cs
2  // Demostración de los métodos Replace, ToLower, ToUpper, Trim,
3  // y ToString de string.
4  using System;
5
6  class MetodosString2
7  {
8      public static void Main()
9      {
10         string string1 = "saludos!";
11         string string2 = "ADIOS ";
12         string string3 = "    espacios    ";
13
14         Console.WriteLine( "string1 = \"" + string1 + "\"\n" +
15                             "string2 = \"" + string2 + "\"\n" +
16                             "string3 = \"" + string3 + "\"");
17
18         // llama al método Replace
19         Console.WriteLine(
20             "\nSustitución de \"u\" con \"U\" en string1: \"" +
21             string1.Replace( 'u', 'U' ) + "\"");
22
23         // llama a ToLower y ToUpper
24         Console.WriteLine( "\nstring1.ToUpper() = \"" +
25                             string1.ToUpper() + "\"\nstring2.ToLower() = \"" +
26                             string2.ToLower() + "\"");
27
28         // llama al método Trim
29         Console.WriteLine( "\nstring3 después de Trim = \"" +
30                             string3.Trim() + "\"");
31
32         Console.WriteLine( "\nstring1 = \"" + string1 + "\"");
33     } // fin del método Main
34 } // fin de la clase MetodosString2

```

```

string1 = "saludos!"
string2 = "ADIOS "
string3 = "    espacios    "

Sustitución de "u" con "U" en string1: "salUUDos!"

string1.ToUpper() = "SALUUDOS!"
string2.ToLower() = "adios "

string3 después de Trim = "espacios"

string1 = "saludos!"

```

Figura 16.8 | Los métodos `Replace`, `ToLower`, `ToUpper` y `Trim` de `string`.

El método `ToUpper` de `string` genera un nuevo objeto `string` (línea 25) que sustituye cualquier letra minúscula en `string1` con su equivalente en mayúscula. El método devuelve un nuevo objeto `string` que contiene el objeto `string` convertido; el objeto `string` original permanece sin cambios. Si no hay caracteres para convertir, se devuelve el objeto `string` original. La línea 26 utiliza el método `ToLower` de `string` para devolver un nuevo objeto `string`, en el que se sustituyen todas las letras mayúsculas en `string2` por sus equivalentes en minúsculas. El objeto `string` original no se modifica. Al igual que con `ToUpper`, si no hay caracteres que convertir a minúscula, el método `ToLower` devuelve el objeto `string` original.

La línea 30 utiliza el método `Trim` de `string` para eliminar todos los caracteres de espacios en blanco que aparecen al principio y al final de un objeto `string`. Sin alterar de otra forma el objeto `string` original, el método devuelve un nuevo objeto `string` que contiene al objeto `string` original, pero omite los caracteres de espacio en blanco a la izquierda o a la derecha. Otra versión del método `Trim` recibe un arreglo de caracteres y devuelve un objeto `string` que no contiene los caracteres incluidos en el argumento tipo arreglo.

16.10 La clase `StringBuilder`

La clase `string` proporciona muchas herramientas para procesar objetos `string`. No obstante, el contenido de un objeto `string` nunca puede cambiar. Las operaciones que parecen concatenar objetos `string`, en realidad asignan referencias `string` a objetos `string` recién creados (por ejemplo, el operador `+=` crea un nuevo objeto `string` y asigna la referencia `string` inicial al objeto `string` recién creado).

En las siguientes secciones hablaremos sobre las características de la clase `StringBuilder` (espacio de nombres `System.Text`), que se utiliza para crear y manipular información de cadenas dinámicas (es decir, objetos `string` mutables). Cada objeto `StringBuilder` puede almacenar cierto número de caracteres que se especifican con base en su capacidad. Si se excede la capacidad de un objeto `StringBuilder`, su capacidad se expande para dar cabida a los caracteres adicionales. Como veremos, los miembros de la clase `StringBuilder` tales como los métodos `Append` y `AppendFormat`, pueden usarse para la concatenación, de igual forma que los operadores `+` y `+=` para la clase `string`.



Tip de rendimiento 16.2

Los objetos de la clase `string` son inmutables (es decir, cadenas constantes), mientras que los objetos de la clase `StringBuilder` son mutables. C# puede realizar ciertas optimizaciones que involucran objetos `string` (tales como compartir un objeto `string` entre varias referencias), debido a que sabe que estos objetos nunca cambiarán.

La clase `StringBuilder` proporciona seis constructores sobrecargados. La clase `ConstructorStringBuilder` (figura 16.9) demuestra tres de estos constructores sobrecargados.

La línea 12 emplea el constructor de `StringBuilder` sin parámetros para crear un objeto `StringBuilder` que no contiene caracteres y tiene una capacidad inicial predeterminada de 16 caracteres. La línea 13 utiliza el constructor de `StringBuilder` que recibe un argumento `int` para crear un objeto `StringBuilder` que no contiene caracteres y tiene la capacidad inicial especificada en el argumento `int` (es decir, 10). La línea 14 utiliza el constructor de `StringBuilder` que recibe un argumento `string` para crear un objeto `StringBuilder` que contiene los caracteres del argumento `string`. La capacidad inicial es la potencia de dos más pequeña, que sea mayor o igual al número de caracteres en el argumento `string`, con un mínimo de 16. Las líneas 16-18 utilizan en forma implícita el método `ToString` de `StringBuilder` para obtener las representaciones `string` del contenido de los objetos `StringBuilder`.

16.11 Las propiedades `Length` y `Capacity`, el método `EnsureCapacity` y el indexador de la clase `StringBuilder`

La clase `StringBuilder` cuenta con las propiedades `Length` y `Capacity` para devolver el número de caracteres actuales en un objeto `StringBuilder`, y el número de caracteres que puede almacenar sin asignar más memoria, respectivamente. Estas propiedades también pueden incrementar o decrementar la longitud o la capacidad del objeto `StringBuilder`.

El método `EnsureCapacity` le permite reducir el número de veces que debe incrementarse la capacidad de un objeto `StringBuilder`. Este método duplica la capacidad actual de una instancia de `StringBuilder`. Si este valor duplicado es mayor que el valor que el programador desea asegurar, ese valor se convierte en la nueva capacidad. En

```

1 // Fig. 16.9: ConstructorStringBuilder.cs
2 // Demostración de los constructores de la clase StringBuilder.
3 using System;
4 using System.Text;
5
6 class ConstructorStringBuilder
7 {
8     public static void Main()
9     {
10         StringBuilder bufer1, bufer2, bufer3;
11
12         bufer1 = new StringBuilder();
13         bufer2 = new StringBuilder( 10 );
14         bufer3 = new StringBuilder( "hola" );
15
16         Console.WriteLine( "bufer1 = \"\" + bufer1 + "\"\" );
17         Console.WriteLine( "bufer2 = \"\" + bufer2 + "\"\" );
18         Console.WriteLine( "bufer3 = \"\" + bufer3 + "\"\" );
19     } // fin del método Main
20 } // fin de la clase ConstructorStringBuilder

```

```

bufer1 = ""
bufer2 = ""
bufer3 = "hola"

```

Figura 16.9 | Constructores de la clase StringBuilder.

caso contrario, `EnsureCapacity` altera la capacidad para hacer que sea igual al número solicitado. Por ejemplo, si la capacidad actual es 17 y deseamos hacerla 40, 17 multiplicado por 2 no es mayor que 40, por lo que la llamada resultará en una nueva capacidad de 40. Si la capacidad actual es 23 y deseamos hacerla 40, 23 se multiplicará por 2 para producir una nueva capacidad de 46. Tanto 40 como 46 son mayores o iguales que 40, por lo que definitivamente el método `EnsureCapacity` asegura una capacidad de 40. El programa en la figura 16.10 demuestra el uso de estos métodos y propiedades.

```

1 // Fig. 16.10: CaracteristicasStringBuilder.cs
2 // Demostración de algunas características de la clase StringBuilder.
3 using System;
4 using System.Text;
5
6 class CaracteristicasStringBuilder
7 {
8     public static void Main()
9     {
10         StringBuilder bufer =
11             new StringBuilder( "Hola, ¿cómo estás?" );
12
13         // usa las propiedades Length y Capacity
14         Console.WriteLine( "bufer = " + bufer +
15             "\nLength = " + bufer.Length +
16             "\nCapacity = " + bufer.Capacity );
17
18         bufer.EnsureCapacity( 75 ); // asegura una capacidad de por lo menos 75
19         Console.WriteLine( "\nNueva capacidad = " +
20             bufer.Capacity );
21

```

Figura 16.10 | Manipulación del tamaño de objetos StringBuilder. (Parte I de 2).

```

22      // trunca StringBuilder estableciendo la propiedad Length
23      bufer.Length = 10;
24      Console.Write( "\nNueva longitud = " +
25          bufer.Length + "\nbufer = " );
26
27      // usa indexador de StringBuilder
28      for ( int i = 0; i < bufer.Length; i++ )
29          Console.Write( bufer[ i ] );
30
31      Console.WriteLine( "\n" );
32  } // fin del método Main
33 } // fin de la clase CaracteristicasStringBuilder

```

```

bufer = Hola, ¿cómo estás?
Length = 18
Capacity = 32

Nueva capacidad = 75

Nueva longitud = 10
bufer = Hola, ¿cóm

```

Figura 16.10 | Manipulación del tamaño de objetos `StringBuilder`. (Parte 2 de 2).

El programa contiene un objeto `StringBuilder`, llamado `bufer`. Las líneas 10-11 del programa utilizan el constructor de `StringBuilder` que recibe un argumento `string` para instanciar el objeto `StringBuilder` e inicializar su valor a "Hola, ¿cómo estás?". Las líneas 14-16 muestran en pantalla el contenido, la longitud y la capacidad del objeto `StringBuilder`. En la ventana de salida, observe que al principio, la capacidad del objeto `StringBuilder` es 32. Recuerde que el constructor de `StringBuilder` que recibe un argumento `string` crea un objeto `StringBuilder` con una capacidad inicial que es la potencia más pequeña de 2, mayor o igual al número de caracteres en el objeto `string` que recibe como argumento.

La línea 18 expande la capacidad del objeto `StringBuilder` hasta un mínimo de 75 caracteres. La capacidad actual (32) multiplicada por 2 es menor que 75, por lo que el método `EnsureCapacity` aumenta la capacidad a 75. Si se agregan nuevos caracteres a un objeto `StringBuilder` de manera que su longitud exceda a su capacidad, ésta crece para dar cabida a los caracteres adicionales, de la misma forma que si se hubiera llamado al método `EnsureCapacity`.

La línea 23 utiliza la propiedad `Length` para establecer la longitud del objeto `StringBuilder` a 10. Si la longitud especificada es menor que el número actual de caracteres en el objeto `StringBuilder`, el contenido del objeto `StringBuilder` se trunca a la longitud especificada. Si la longitud especificada es mayor que el número de caracteres actuales en el objeto `StringBuilder`, se adjuntan caracteres de espacio al objeto `StringBuilder` hasta que el número total de caracteres en el objeto `StringBuilder` sea igual a la longitud especificada.



Error común de programación 16.3

Asignar `null` a una referencia `string` puede producir errores lógicos si usted trata de comparar `null` con un objeto `string` vacío. La palabra clave `null` es un valor que representa una referencia nula (es decir, una referencia que no se refiere a un objeto), y no un objeto `string` vacío (el cual es un objeto `string` que tiene longitud 0 y no contiene caracteres).

16.12 Los métodos `Append` y `AppendFormat` de la clase `StringBuilder`

La clase `StringBuilder` cuenta con 19 métodos `Append` sobrecargados, que permiten agregar varios tipos de valores al final de un objeto `StringBuilder`. La FCL proporciona versiones para cada uno de los tipos simples y para arreglos de caracteres, objetos `string` y `object`. (Recuerde que el método `ToString` produce una repre-

sentación string de cualquier objeto object.) Cada uno de los métodos recibe un argumento, lo convierte en un objeto string y lo adjunta al objeto StringBuilder. La figura 16.11 demuestra el uso de varios métodos Append.

Las líneas 22-40 utilizan 10 métodos Append sobrecargados distintos, para adjuntar las representaciones de los objetos creados en las líneas 10-18 al final del objeto StringBuilder. Append se comporta de manera similar al operador +, el cual se utiliza para concatenar objetos string.

La clase StringBuilder también proporciona el método AppendFormat, el cual convierte un objeto string a un formato especificado, y después lo adjunta al objeto StringBuilder. El ejemplo en la figura 16.12 demuestra el uso de este método.

```

1  // Fig. 16.11: AppendStringBuilder.cs
2  // Demostración de los métodos Append de StringBuilder.
3  using System;
4  using System.Text;
5
6  class AppendStringBuilder
7  {
8      public static void Main()
9      {
10         object valorObjeto = "hola";
11         string valorCadena = "adiós";
12         char[] arregloCaracteres = { 'a', 'b', 'c', 'd', 'e', 'f' };
13         bool valorBooleano = true;
14         char valorCaracter = 'Z';
15         int valorEntero = 7;
16         long valorLong = 1000000;
17         float valorFloat = 2.5F; // el sufijo F indica que 2.5 es un float
18         double valorDouble = 33.333;
19         StringBuilder bufer = new StringBuilder();
20
21         // usa el método Append para adjuntar valores al bufer
22         bufer.Append( valorObjeto );
23         bufer.Append( " " );
24         bufer.Append( valorCadena );
25         bufer.Append( " " );
26         bufer.Append( arregloCaracteres );
27         bufer.Append( " " );
28         bufer.Append( arregloCaracteres, 0, 3 );
29         bufer.Append( " " );
30         bufer.Append( valorBooleano );
31         bufer.Append( " " );
32         bufer.Append( valorCaracter );
33         bufer.Append( " " );
34         bufer.Append( valorEntero );
35         bufer.Append( " " );
36         bufer.Append( valorLong );
37         bufer.Append( " " );
38         bufer.Append( valorFloat );
39         bufer.Append( " " );
40         bufer.Append( valorDouble );
41
42         Console.WriteLine( "bufer = " + bufer.ToString() + "\n" );
43     } // fin del método Main
44 } // fin de la clase AppendStringBuilder

```

```
bufer = hola adiós abcdef abc True Z 7 1000000 2.5 33.333
```

Figura 16.11 | Los métodos Append de StringBuilder.

```

1  // Fig. 16.12: AppendFormatStringBuilder.cs
2  // Demostración del método AppendFormat.
3  using System;
4  using System.Text;
5
6  class AppendFormatStringBuilder
7  {
8      public static void Main()
9      {
10         StringBuilder bufer = new StringBuilder();
11         string string1, string2;
12
13         // string con formato
14         string1 = "Este {0} cuesta: {1:C}.\n";
15
16         // arreglo de argumentos de string1
17         object[] arregloObjetos = new object[ 2 ];
18
19         arregloObjetos[ 0 ] = "auto";
20         arregloObjetos[ 1 ] = 1234.56;
21
22         // adjunta al búfer el objeto string con formato y con argumento
23         bufer.AppendFormat( string1, arregloObjetos );
24
25         // string con formato
26         string2 = "Número:{0:d3}.\n" +
27                 "Número alineado a la derecha con espacios:{0, 4}.\n" +
28                 "Número alineado a la izquierda con espacios:{0, -4}.";
29
30         // adjunta al búfer el objeto string con formato y con argumento
31         bufer.AppendFormat( string2, 5 );
32
33         // muestra los objetos string con formato
34         Console.WriteLine( bufer.ToString() );
35     } // fin del método Main
36 } // fin de la clase AppendFormatStringBuilder

```

Este auto cuesta: \$1,234.56.

Número:005.

Número alineado a la derecha con espacios: 5.

Número alineado a la izquierda con espacios:5 .

Figura 16.12 | El método AppendFormat de StringBuilder.

La línea 14 crea un objeto `string` que contiene información sobre el formato. La información encerrada entre llaves especifica cómo dar formato a una pieza específica de datos. Los formatos tienen la forma `{X[,Y][:CadenaFormato]}`, en donde `X` es el número del argumento al que se va a dar formato, partiendo desde cero. `Y` es un argumento opcional, el cual puede ser positivo o negativo, e indica cuántos caracteres debe haber en el resultado. Si el objeto `string` resultante es menor que el número `Y`, el objeto `string` se rellenará con espacios para compensar la diferencia. Un entero positivo alinea el objeto `string` a la derecha; un entero negativo lo alinea a la izquierda. La `CadenaFormato` opcional aplica un formato específico al argumento: moneda, decimal o científico, entre otros. En este caso, “{0}” significa que el primer argumento se va a imprimir en pantalla. “{1:C}” especifica que el segundo argumento recibirá un formato como valor de moneda.

La línea 23 muestra una versión de `AppendFormat` que recibe dos parámetros: un objeto `string` especifica el formato y un arreglo de objetos que sirven como argumento para el objeto `string` de formato. El argumento al que se hace referencia como “{0}” se encuentra en el arreglo de objetos, en el índice 0.

Las líneas 26-28 definen otro objeto `string` utilizado para el formato. El primer formato “{0:d3}” especifica que al primer argumento se le aplicará un formato como decimal de tres dígitos, lo que significa que a

cualquier número que tenga menos de tres dígitos se le colocarán ceros a la derecha para compensar la diferencia. El siguiente formato, "{0, 4}", especifica que el objeto `string` con formato debe tener cuatro caracteres y estar alineado a la derecha. El tercer formato, "{0, -4}", especifica que los objetos `string` deben estar alineados a la izquierda. Para obtener más opciones de formato, consulte la documentación de ayuda en línea.

La línea 31 usa una versión de `AppendFormat` que recibe dos parámetros: un objeto `string` que contienen un formato y un objeto al que se le aplica este formato. En este caso, el objeto es el número 5. El despliegue en pantalla de la figura 16.12 muestra el resultado de aplicar estas dos versiones de `AppendFormat` con sus respectivos argumentos.

16.13 Los métodos Insert, Remove y Replace de la clase StringBuilder

La clase `StringBuilder` cuenta con 18 métodos `Insert` sobrecargados para permitir insertar varios tipos de datos en cualquier posición de un objeto `StringBuilder`. La clase proporciona versiones para cada uno de los tipos simples y para los arreglos de caracteres, objetos `string` y `object`. Cada método recibe su segundo argumento, lo convierte en un objeto `string` e inserta ese objeto `string` en el objeto `StringBuilder`, enfrente del carácter en la posición especificada por el primer argumento. El índice especificado por el primer argumento debe ser mayor o igual a 0 y menor que la longitud del objeto `StringBuilder`; en caso contrario, el programa lanza una excepción `ArgumentOutOfRangeException`.

La clase `StringBuilder` también proporciona el método `Remove` para eliminar cualquier porción de un objeto `StringBuilder`. El método `Remove` recibe dos argumentos: el índice en el cual empieza la eliminación y el número de caracteres a eliminar. La suma del índice inicial y el número de caracteres a eliminar siempre debe ser menor que la longitud del objeto `StringBuilder`; en caso contrario, el programa lanza una excepción `ArgumentOutOfRangeException`. En la figura 16.13 se demuestra el uso de los métodos `Insert` y `Remove`.

```

1 // Fig. 16.13: InsertRemoveStringBuilder.cs
2 // Demostración de los métodos Insert y Remove de
3 // la clase StringBuilder.
4 using System;
5 using System.Text;
6
7 class InsertRemoveStringBuilder
8 {
9     public static void Main()
10    {
11        object valorObjeto = "hola";
12        string valorCadena = "adiós";
13        char[] arregloCaracteres = { 'a', 'b', 'c', 'd', 'e', 'f' };
14        bool valorBooleano = true;
15        char valorCaracter = 'K';
16        int valorEntero = 7;
17        long valorLong = 10000000;
18        float valorFloat = 2.5F; // el sufijo F indica que 2.5 es un float
19        double valorDouble = 33.333;
20        StringBuilder bufer = new StringBuilder();
21
22        // inserta los valores en el bufer
23        bufer.Insert( 0, valorObjeto );
24        bufer.Insert( 0, " " );
25        bufer.Insert( 0, valorCadena );
26        bufer.Insert( 0, " " );
27        bufer.Insert( 0, arregloCaracteres );
28        bufer.Insert( 0, " " );
29        bufer.Insert( 0, valorBooleano );

```

Figura 16.13 | Inserción y eliminación de texto en objetos `StringBuilder`. (Parte I de 2).

```

30     bufer.Insert( 0, " " );
31     bufer.Insert( 0, valorCaracter );
32     bufer.Insert( 0, " " );
33     bufer.Insert( 0, valorEntero );
34     bufer.Insert( 0, " " );
35     bufer.Insert( 0, valorLong );
36     bufer.Insert( 0, " " );
37     bufer.Insert( 0, valorFloat );
38     bufer.Insert( 0, " " );
39     bufer.Insert( 0, valorDouble );
40     bufer.Insert( 0, " " );
41
42     Console.WriteLine( "bufer después de Insert: \n" + bufer + "\n" );
43
44     bufer.Remove( 10, 1 ); // elimina el 2 en 2.5
45     bufer.Remove( 4, 4 ); // elimina el .333 en 33.333
46
47     Console.WriteLine( "bufer después de Remove:\n" + bufer );
48 } // fin del método Main
49 } // fin de la clase InsertRemoveStringBuilder

```

```

bufer después de Insert:
33.333 2.5 10000000 7 K True abcdef adiós hola

bufer después de Remove:
33 .5 10000000 7 K True abcdef adiós hola

```

Figura 16.13 | Inserción y eliminación de texto en objetos `StringBuilder`. (Parte 2 de 2).

Otra método útil que se incluye con `StringBuilder` es `Replace`; que busca un objeto `string` o un carácter especificado y lo sustituye por otro objeto `string` o carácter. La figura 16.14 demuestra este método.

La línea 18 usa el método `Replace` para sustituir todas las instancias del objeto `string` "Jane" con el objeto `string` "Adalberto" en `builder1`. Otra sobrecarga de este método recibe dos caracteres como parámetro y sustituye cada ocurrencia del primer carácter con el segundo carácter. La línea 19 usa una sobrecarga de `Replace` que recibe cuatro parámetros, de los cuales los primeros dos son caracteres y los otros dos son `int`. Este método sustituye todas las instancias del primer carácter con el segundo carácter, empezando en el índice especificado por el primer `int` y continuando en base a una cuenta especificada por el segundo `int`. Así, en este caso, `Replace` busca sólo a través de cinco caracteres, empezando con el carácter en el índice 0. Como se muestra en los resultados, esta versión de `Replace` sustituye a por A en la palabra "adiós", pero no en "Adalberto". Esto se debe a que las letras a en "adalberto" no se encuentran en el rango indicado por los argumentos `int` (es decir, entre los índices 0 y 4).

```

1  // Fig. 16.14: ReplaceStringBuilder.cs
2  // Demostración del método Replace.
3  using System;
4  using System.Text;
5
6  class ReplaceStringBuilder
7  {
8      public static void Main()
9      {
10         StringBuilder builder1 =
11             new StringBuilder( "Feliz cumpleaños Jane" );
12         StringBuilder builder2 =

```

Figura 16.14 | Sustitución de texto en objetos `StringBuilder`. (Parte 1 de 2).


```

13         new StringBuilder( "adiós adalberto" );
14
15         Console.WriteLine( "Antes de las sustituciones:\n" +
16             builder1.ToString() + "\n" + builder2.ToString() );
17
18         builder1.Replace( "Jane", "Adalberto" );
19         builder2.Replace( 'a', 'A', 0, 5 );
20
21         Console.WriteLine( "\nDespués de las sustituciones:\n" +
22             builder1.ToString() + "\n" + builder2.ToString() );
23     } // fin del método Main
24 } // fin de la clase ReplaceStringBuilder

```

Antes de las sustituciones:
 Feliz cumpleaños Jane
 adiós adalberto

Después de las sustituciones:
 Feliz cumpleaños Adalberto
 Adiós adalberto

Figura 16.14 | Sustitución de texto en objetos `StringBuilder`. (Parte 2 de 2).

16.14 Métodos de char

C# cuenta con un tipo llamado **struct** (abreviación de estructura), el cual es similar a una clase. Aunque los tipos `struct` y las clases son comparables en muchas formas, los tipos `struct` representan tipos de valores. Al igual que las clases, los tipos `struct` pueden tener métodos y propiedades, y pueden utilizar los modificadores de acceso `public` y `private`. Además, se puede acceder a los miembros `struct` a través del operador de acceso a miembros (`.`).

En realidad, los tipos simples son alias para los tipos `struct`. Por ejemplo, un `int` se define mediante el tipo `struct system.Int32`, un `long` mediante `System.Int64`, y así en lo sucesivo. Todos los tipos `struct` se derivan de la clase **ValueType**, la cual a su vez se deriva de `object`. Además, todos los tipos `struct` son sellados (`sealed`) de manera implícita, por lo que no soportan métodos `virtual` o `abstract`, y sus miembros no pueden declararse como `protected` o `protected internal`.

En esta sección, presentaremos la estructura **Char**,² que viene siendo la estructura para los caracteres. La mayoría de los métodos de `Char` son `static`, reciben cuando menos un argumento tipo carácter y realizan una prueba o una manipulación sobre ese carácter. En el siguiente ejemplo presentaremos varios de estos métodos. La figura 16.15 demuestra el uso de métodos `static` que evalúan caracteres para determinar si son de un tipo carácter específico, y métodos `static` que realizan conversiones de mayúsculas/minúsculas en los caracteres.

Esta aplicación Windows contiene un indicador, un objeto `TextBox` en el que el usuario puede introducir un carácter, un botón que el usuario puede oprimir después de introducir un carácter y un segundo objeto `TextBox` que muestra el resultado de nuestro análisis. En cuando el usuario hace clic en el botón **Analizar carácter**, se invoca el manejador de eventos `analizarButton_Click` (líneas 16-40). Este manejador de eventos convierte la entrada de un objeto `string` en un objeto `char`, mediante el uso del método `Convert.ToChar` (línea 19).

La línea 23 utiliza el método `IsDigit` de `Char` para determinar si `character` está definido como un dígito. Si es así, el método devuelve `true`; en caso contrario, devuelve `false` (observe de nuevo que los valores `bool` se imprimen en pantalla con la primera letra mayúscula). La línea 25 utiliza el método `IsLetter` de `Char` para determinar si el carácter `character` es una letra. La línea 27 utiliza el método `IsLetterOrDigit` de `Char` para determinar si el carácter `character` es una letra o un dígito.

2. Al igual que la palabra clave `string` es un alias para la clase `String`, la palabra clave `char` es un alias para el tipo `struct Char`. En este libro, utilizaremos el término `Char` cuando llamemos a un método `static` de la estructura `Char`, y el término `char` en cualquier otra parte.


```

1  // Fig. 16.15: MetodosStaticChar.cs
2  // Demuestra el uso de los métodos static para evaluar caracteres
3  // de la estructura Char
4  using System;
5  using System.Windows.Forms;
6
7  public partial class MetodosStaticCharForm : Form
8  {
9      // constructor predeterminado
10     public MetodosStaticCharForm()
11     {
12         InitializeComponent();
13     } // fin del constructor
14
15     // maneja el evento analizarButton_Click
16     private void analizarButton_Click( object sender, EventArgs e )
17     {
18         // convierte objeto string introducido al tipo char
19         char caracter = Convert.ToChar( entradaTextBox.Text );
20         string salida;
21
22         salida = "es dígito: " +
23             Char.IsDigit( caracter ) + "\r\n";
24         salida += "es letra: " +
25             Char.IsLetter( caracter ) + "\r\n";
26         salida += "es letra o dígito: " +
27             Char.IsLetterOrDigit( caracter ) + "\r\n";
28         salida += "está en minúscula: " +
29             Char.IsLower( caracter ) + "\r\n";
30         salida += "está en mayúscula: " +
31             Char.IsUpper( caracter ) + "\r\n";
32         salida += "a mayúsculas: " +
33             Char.ToUpper( caracter ) + "\r\n";
34         salida += "a minúsculas: " +
35             Char.ToLower( caracter ) + "\r\n";
36         salida += "es signo de puntuación: " +
37             Char.IsPunctuation( caracter ) + "\r\n";
38         salida += "es símbolo: " + Char.IsSymbol( caracter );
39         salidaTextBox.Text = salida;
40     } // fin del método analizarButton_Click
41 } // fin de la clase MetodosStaticCharForm

```

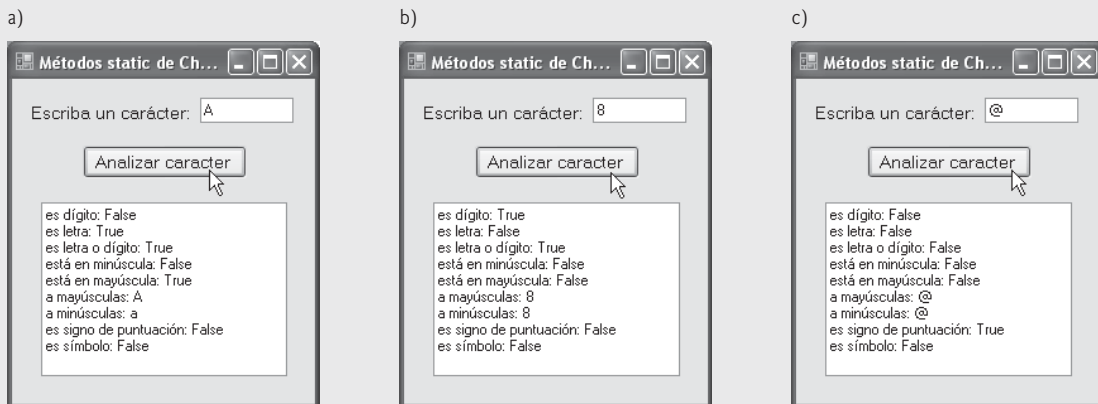


Figura 16.15 | Métodos de Char para evaluación de caracteres y conversión de mayúsculas/minúsculas. (Parte I de 2).

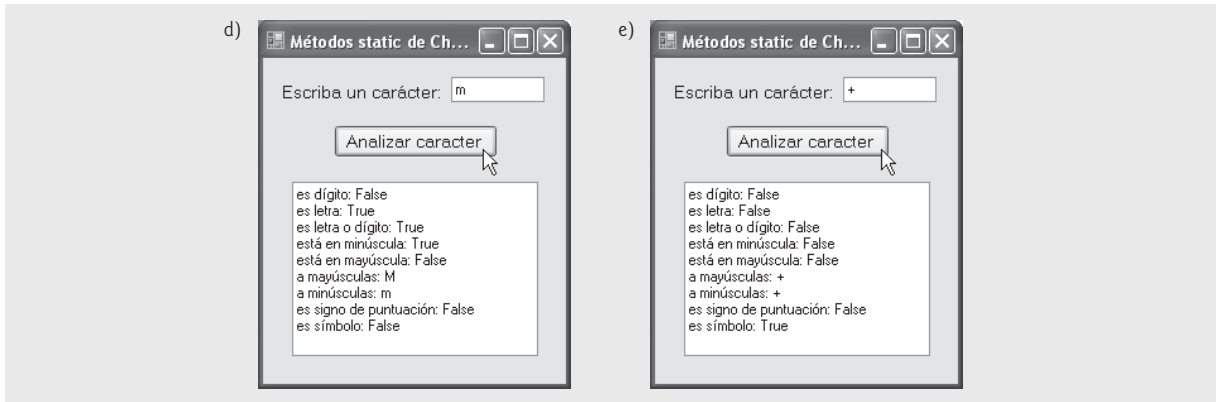


Figura 16.15 | Métodos de Char para evaluación de caracteres y conversión de mayúsculas/minúsculas. (Parte 2 de 2).

La línea 29 utiliza el método `IsLower` de `Char` para determinar si el carácter `character` es una letra minúscula. La línea 31 usa el método `IsUpper` de `Char` para determinar si el carácter `character` es una letra mayúscula. La línea 33 usa el método `ToUpper` de `Char` para convertir el carácter `character` a su equivalente en mayúscula. El método devuelve el carácter convertido si éste tiene un equivalente en mayúscula; en caso contrario, el método devuelve su argumento original. La línea 35 usa el método `ToLower` de `Char` para convertir el carácter `character` en su equivalente en minúscula. El método devuelve el carácter convertido si éste tiene un equivalente en minúscula; en caso contrario, el método devuelve su argumento original.

La línea 37 utiliza el método `IsPunctuation` de `Char` para determinar si el `character` es un signo de puntuación, tal como "!", ":", " o)". La línea 38 usa el método `IsSymbol` de `Char` para determinar si `character` es un símbolo, como "+", "=" o "^".

La estructura tipo `Char` también contiene otros métodos que no mostramos en este ejemplo. Muchos de los métodos `static` son similares; por ejemplo, `IsWhiteSpace` se utiliza para determinar si cierto `character` es un espacio en blanco (es decir, nueva línea, tabulador o espacio). La estructura también contiene varios métodos de instancia `public`; muchos de estos, como los métodos `ToString` y `Equals`, son métodos que hemos visto antes en otras clases. Este grupo incluye el método `CompareTo`, el cual se utiliza para comparar dos valores tipo carácter entre sí.

16.15 Simulación para barajar y repartir cartas

En esta sección utilizaremos la generación de números aleatorios para desarrollar un programa que simule el proceso de barajar y repartir cartas. Estas técnicas pueden formar la base de programas que implementen juegos de cartas específicos. Al final de este capítulo incluimos varios ejercicios que requieren de las herramientas para barajar y repartir cartas.

La clase `Carta` (figura 16.16) contiene dos variables de instancia `string` (`cara` y `palo`), que almacenan referencias al valor de la cara y el nombre del palo de una carta específica. El constructor para la clase recibe dos objetos `string` que utiliza para inicializar a `cara` y a `palo`. El método `ToString` (líneas 16-19) crea un objeto `string` que consiste en la `cara` y el `palo` de la carta, para identificarla cuando se reparta.

Desarrollaremos la aplicación `PaqueteForm` (figura 16.17), que crea un paquete de 52 cartas de juego, mediante el uso de objetos `Carta`. Los usuarios pueden repartir cada carta haciendo clic en el botón **Repartir carta**. Cada carta repartida se muestra en un control `Label`. Los usuarios también pueden barajar el paquete en cualquier momento, haciendo clic en el botón **Barajar cartas**.

El método `PaqueteForm_Load` (líneas 19-31 de la figura 16.17) utiliza una instrucción `for` (líneas 29-30) para llenar el arreglo `paquete` con objetos `Carta`. Observe que cada objeto `Carta` se instancia y se inicializa mediante dos objetos `string`: uno del arreglo `caras` (los objetos `string` del "As" hasta el "Rey") y uno del arreglo `palos` ("Corazones", "Diamantes", "Tréboles" o "Espadas"). El cálculo `i % 13` siempre produce un valor del 0 al 12 (los 13 subíndices del arreglo `caras`), y el cálculo `i / 13` siempre produce un valor del 0 al 3 (los cuatro subíndices en el arreglo `palos`). El arreglo `paquete` inicializado contiene las cartas con las caras desde el As hasta el Rey para cada palo.

```

1  // Fig. 16.16: Carta.cs
2  // Almacena información sobre el palo y la cara en cada carta.
3  using System;
4
5  public class Carta
6  {
7      private string cara;
8      private string palo;
9
10     public Carta( string valorCara, string valorPalo )
11     {
12         cara = valorCara;
13         palo = valorPalo;
14     } // fin del constructor
15
16     public override string ToString()
17     {
18         return cara + " de " + palo;
19     } // fin del método ToString
20 } // fin de la clase Carta

```

Figura 16.16 | La clase Carta.

Cuando el usuario hace clic en el botón **Repartir carta**, el manejador de eventos `repartirButton_Click` (líneas 34-50) invoca el método `RepartirCarta` (definido en las líneas 75-90) para obtener la siguiente carta en el arreglo `paquete`. Si `paquete` no está vacío, el método devuelve una referencia a un objeto `Carta`; en caso contrario, devuelve `null`. Si la referencia no es `null`, las líneas 42-43 muestran la `Carta` en `mostrarLabel` y el número de la carta en `estadoLabel`.

```

1  // Fig. 16.17: PaqueteForm.cs
2  // Simulación para barajar y repartir cartas.
3  using System;
4  using System.Windows.Forms;
5
6  public partial class PaqueteForm : Form
7  {
8      private Carta[] paquete = new Carta[ 52 ]; // paquete de 52 cartas
9      private int cartaActual; // cuenta cual carta se acaba de repartir
10
11     // constructor predeterminado
12     public PaqueteForm()
13     {
14         // Requerido para soporte del Diseñador de Windows Forms
15         InitializeComponent();
16     } // fin del constructor
17
18     // maneja el formulario al momento de cargarlo
19     private void PaqueteForm_Load( object sender, EventArgs e )
20     {
21         string[] caras = { "As", "Dos", "Tres", "Cuatro", "Cinco",
22                             "Seis", "Siete", "Ocho", "Nueve", "Diez",
23                             "Joto", "Quina", "Rey" };
24         string[] palos = { "Corazones", "Diamantes", "Tréboles", "Espadas" };
25
26         cartaActual = -1; // no se han repartido cartas

```

Figura 16.17 | Simulación para barajar y repartir cartas. (Parte 1 de 3).

```

27
28 // inicializa el paquete
29 for ( int i = 0; i < paquete.Length; i++ )
30     paquete[ i ] = new Carta( caras[ i % 13 ], palos[ i / 13 ] );
31 } // fin del método PaqueteForm_Load
32
33 // maneja el clic de repartirButton
34 private void repartirButton_Click( object sender, EventArgs e )
35 {
36     Carta repartida = RepartirCarta();
37
38     // si la carta repartida es null, no quedan cartas
39     // el jugador debe barajar las cartas
40     if ( repartida != null )
41     {
42         mostrarLabel.Text = repartida.ToString();
43         estadoLabel.Text = "Carta #: " + cartaActual;
44     } // fin de if
45     else
46     {
47         mostrarLabel.Text = "NO HAY MÁS CARTAS PARA REPARTIR";
48         estadoLabel.Text = "Barajee las cartas para continuar";
49     } // fin de else
50 } // fin del método repartirButton_Click
51
52 // baraja las cartas
53 private void Barajar()
54 {
55     Random numeroAleatorio = new Random();
56     Carta valorTemporal;
57
58     cartaActual = -1;
59
60     // intercambia cada carta con una carta seleccionada al azar (0-51)
61     for ( int i = 0; i < paquete.Length; i++ )
62     {
63         int j = numeroAleatorio.Next( 52 );
64
65         // intercambia cartas
66         valorTemporal = paquete[ i ];
67         paquete[ i ] = paquete[ j ];
68         paquete[ j ] = valorTemporal;
69     } // fin de for
70
71     repartirButton.Enabled = true; // se barajó el paquete, ahora se pueden repartir
    cartas
72 } // fin del método Barajar
73
74 // reparte una carta si el paquete no está vacío
75 private Carta RepartirCarta()
76 {
77     // si hay una carta entonces la reparte
78     // en caso contrario, indica que hay que barajar las cartas
79     // deshabilitando repartirButton y devolviendo null
80     if ( cartaActual + 1 < paquete.Length )
81     {
82         cartaActual++; // incrementa la cuenta
83         return paquete[ cartaActual ]; // devuelve la nueva carta
84     } // fin de if

```

Figura 16.17 | Simulación para barajar y repartir cartas. (Parte 2 de 3).

```

85     else
86     {
87         repartirButton.Enabled = false; // paquete vacío, no puede repartir cartas
88         return null; // no devuelve una carta
89     } // fin de else
90 } // fin del método RepartirCarta
91
92 // maneja clic de barajarButton
93 private void barajarButton_Click( object sender, EventArgs e )
94 {
95     mostrarLabel.Text = "BARAJANDO...";
96     Barajar();
97     mostrarLabel.Text = "SE BARAJÓ EL PAQUETE";
98 } // fin del método barajarButton_Click
99 } // fin de la clase PaqueteForm

```

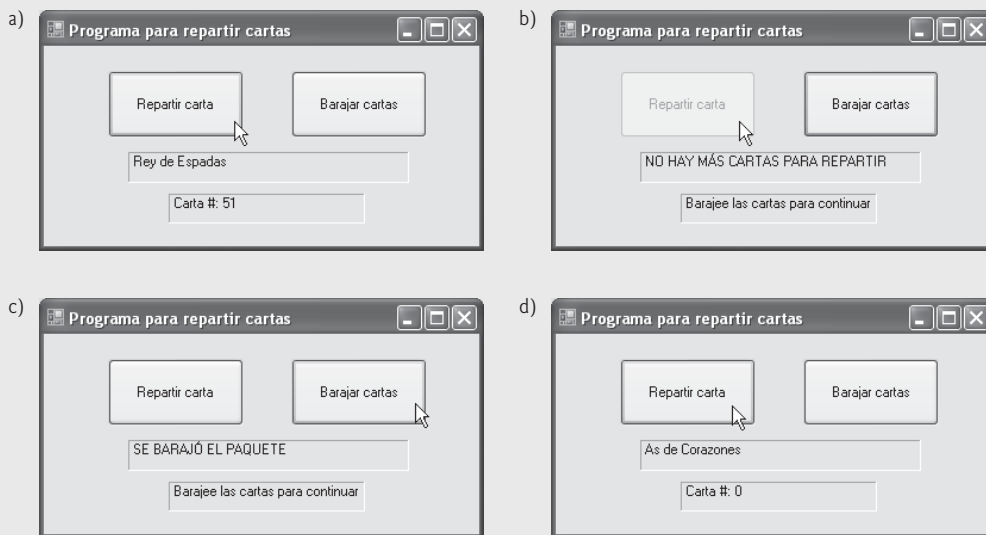


Figura 16.17 | Simulación para barajar y repartir cartas. (Parte 3 de 3).

Si `RepartirCarta` devuelve `null`, se muestra la cadena "NO HAY MÁS CARTAS PARA REPARTIR" en `mostrarLabel`, y se muestra la cadena "Barajee las cartas para continuar" en `estadoLabel`.

Cuando el usuario hace clic en el botón **Barajar cartas**, el manejador de eventos `barajarButton_Click` (líneas 93-98) invoca al método `Barajar` (definido en las líneas 53-72) para barajar las cartas. El método itera a través de las 52 cartas (subíndices 0-51 del arreglo). Para cada carta, el método selecciona al azar un número en el rango de 0 a 51. Después, el objeto `Carta` actual y el objeto `Carta` seleccionado al azar se intercambian en el arreglo. Para barajar las cartas, el método `Barajar` realiza un total de sólo 52 intercambios durante una sola pasada de todo el arreglo. Cuando termina la acción de barajar, `mostrarLabel` muestra la cadena "SE BARAJÓ EL PAQUETE".

16.16 Expresiones regulares y la clase Regex

Las expresiones regulares son objeto `string` un formato especial, que se utilizan para buscar patrones en el texto. Pueden ser útiles durante la validación de información, para asegurar que los datos se encuentren en un formato específico. Por ejemplo, un código postal debe consistir de cinco dígitos, y el apellido debe empezar con letra mayúscula. Los compiladores utilizan las expresiones regulares para validar la sintaxis de los programas. Si el código del programa no concuerda con la expresión regular, el compilador indica que hay un error de sintaxis.

El .NET Framework proporciona varias clases para ayudar a los desarrolladores a reconocer y manipular las expresiones regulares. La clase `Regex` (del espacio de nombres `System.Text.RegularExpressions`) representa

a una expresión regular inmutable. El *método* Match de Regex devuelve un objeto de la *clase* Match, que representa una sola coincidencia de la expresión regular. Regex también cuenta con el método Matches, el cual busca todas las coincidencias de una expresión regular en un objeto string arbitrario, y devuelve un objeto de la clase MatchCollection que contiene todas las coincidencias. Una colección es una estructura de datos, similar a un arreglo y puede usarse con una instrucción foreach para iterar a través de los elementos de la colección. En el capítulo 26, Colecciones, hablaremos sobre las colecciones con más detalle. Para usar la clase Regex, debe agregar una directiva using para el espacio de nombres System.Text.RegularExpressions.

Clases de caracteres de expresiones regulares

La tabla en la figura 16.18 especifica algunas *clases de caracteres* que pueden usarse con las expresiones regulares. No confunda una clase de carácter con la declaración de una clase en C#. Una clase de carácter es tan sólo una secuencia de escape que representa a un grupo de caracteres que podrían aparecer en un objeto string.

Un *carácter de palabra* es cualquier carácter alfanumérico o guión bajo. Un carácter de espacio en blanco es un espacio, un tabulador, un retorno de carro, una nueva línea o un avance de página. Un dígito es cualquier carácter numérico. Las expresiones regulares no se limitan a las clases de caracteres en la figura 16.18. Como verá en nuestro primer ejemplo, las expresiones regulares pueden usar otras notaciones para buscar patrones complejos en objetos string.

16.16.1 Ejemplo de expresión regular

El programa de la figura 16.19 trata de relacionar las fechas de cumpleaños con una expresión regular. Para fines de demostración, la expresión sólo concuerda con los cumpleaños que no ocurren en abril y que pertenecen a personas cuyos nombres empiezan con "J".

Clases de carácter	Concuerda con	Clase de carácter	Concuerda con
\d	Cualquier dígito	\D	Cualquier carácter que no sea dígito
\w	Cualquier carácter de palabra	\W	Cualquier carácter que no sea de palabra
\s	Cualquier espacio en blanco	\S	Cualquier carácter que no sea espacio en blanco

Figura 16.18 | Clases de caracteres.

```
1 // Fig. 16.19: CoincidenciasRegex.cs
2 // Demostración de la clase Regex.
3 using System;
4 using System.Text.RegularExpressions;
5
6 class CoincidenciasRegex
7 {
8     public static void Main()
9     {
10         // crea una expresión regular
11         Regex expresion =
12             new Regex( @"J.*\d[0-35-9]-\d\d-\d\d" );
13
14         string string1 = "Jane cumple años el 05-12-75\n" +
15             "Dave cumple años el 11-04-68\n" +
16             "John cumple años el 04-28-73\n" +
17             "Joe cumple años el 12-17-77";
```

Figura 16.19 | Comprobación de fechas de cumpleaños mediante expresiones regulares. (Parte 1 de 2).

```

18
19      // relaciona expresión regular con objeto string e
20      // imprime todas las coincidencias
21      foreach ( Match miCoincidencia in expresion.Matches( string1 ) )
22          Console.WriteLine( miCoincidencia );
23  } // fin del método Main
24 } // fin de la clase CoincidenciasRegex

```

```

Jane cumple años el 05-12-75
Joe cumple años el 12-17-77

```

Figura 16.19 | Comprobación de fechas de cumpleaños mediante expresiones regulares. (Parte 2 de 2).

Las líneas 11-12 crean un objeto `Regex` y pasan una cadena de patrón de expresión regular al constructor de `Regex`. Observe que antepone un carácter `@` a la cadena. Recuerde que las barras diagonales inversas dentro de las dobles comillas que van después del carácter `@` son caracteres comunes de barra diagonal inversa, no el principio de secuencias de escape. Para definir la expresión regular sin anteponer un `@` a la cadena, habría que escapar cada carácter de barra diagonal inversa, como en:

```
"J.*\\d[0-35-9]-\\d\\d-\\d\\d"
```

lo cual hace que la expresión regular sea más difícil de leer.

El primer carácter en expresión regular, "J", es un carácter literal. Cualquier objeto `string` que coincide con esta expresión regular debe empezar con "J". En una expresión regular, el carácter punto "." coincide con cualquier carácter individual, excepto con el carácter de nueva línea. Cuando el carácter punto va seguido de un asterisco, como en ".*", la expresión regular coincide con cualquier número de caracteres no especificados, excepto nuevas líneas. En general, cuando el operador "*" se aplica a un patrón, el patrón coincide con cero o más ocurrencias. En contraste, si se aplica el operador "+" a un patrón, este coincide con una o más ocurrencias. Por ejemplo, tanto "A*" como "A+" coinciden con "A", pero sólo "A*" coincide con un objeto `string` vacío.

Como se indica en la figura 16.18, "\\d" coincide con cualquier dígito numérico. Para especificar conjuntos de caracteres distintos de los que pertenecen a una clase de carácter predefinida, los caracteres pueden listarse entre corchetes, []. Por ejemplo, el patrón "[aeiou]" coincide con cualquier vocal. Los rangos de caracteres se representan colocando un guión corto (-) entre los caracteres. En el ejemplo, "[0-35-9]" coincide sólo con los dígitos dentro de los rangos especificados por el patrón; es decir, cualquier dígito entre 0 y 3, o entre 5 y 9; por lo tanto, coincide con cualquier dígito excepto 4. También puede especificar que un patrón debe coincidir con cualquier cosa distinta de los caracteres entre corchetes. Para hacerlo, coloque ^ como el primer carácter entre corchetes. Es importante observar que "[^4]" no es igual que "[0-35-9]"; "[^4]" coincide con cualquier carácter que no sea dígito y con los dígitos distintos de 4.

Aunque el carácter "-" indica un rango cuando va encerrado entre corchetes, las instancias del carácter "-" fuera de las expresiones de agrupamiento se tratan como caracteres literales. Por ende, la expresión regular en la línea 12 busca un objeto `string` que empieza con la letra "J", seguido de cualquier número de caracteres, seguido de un número de dos dígitos (de los cuales el segundo no puede ser 4), seguido de un guión bajo, y otro número de dos dígitos, de un guión bajo y de otro número de dos dígitos.

Las líneas 21-22 utilizan una instrucción `foreach` para iterar a través de la colección `MatchCollection` devuelta por el método `Matches` del objeto `expresion`, que recibió a `string1` como argumento. Los elementos en la colección `MatchCollection` son objetos `Match`, por lo que la instrucción `foreach` declara a la variable `miCoincidencia` como de tipo `Match`. Para cada objeto `Match`, la línea 22 imprime en pantalla el texto que coincidió con la expresión regular. Los resultados en la figura 16.19 indican las dos coincidencias que encontramos en `string1`. Observe que ambas se conforman al patrón especificado por la expresión regular.

Cuantificadores

El asterisco (*) en la línea 12 de la figura 16.19 se conoce más formalmente como *cuantificador*. La figura 16.20 lista varios cuantificadores que podemos colocar después de un patrón en una expresión regular, junto con el propósito de cada cuantificador.

Cuantificador	Coincidencias
*	Coincide con cero o más ocurrencias del patrón precedente.
+	Coincide con una o más ocurrencias del patrón precedente.
?	Coincide con cero o una ocurrencias del patrón precedente.
{n}	Coincide con exactamente n ocurrencias del patrón precedente.
{n,}	Coincide con cuando menos n ocurrencias del patrón precedente.
{n,m}	Coincide con entre n y m (inclusive) ocurrencias del patrón precedente.

Figura 16.20 | Cuantificadores usados en expresiones regulares.

Ya hemos hablado sobre cómo funcionan el asterisco (*) y el signo de suma (+) como cuantificadores. El cuantificador de signo de interrogación (?) coincide con cero o una ocurrencia del patrón que cuantifica. Un conjunto de llaves que contenga un número ({n}) coincide exactamente con n ocurrencias del patrón que cuantifica. En el siguiente ejemplo demostramos este cuantificador. Al incluir una coma después del número encerrado entre llaves, coinciden cuando menos n ocurrencias del patrón cuantificado. El conjunto de llaves que contienen dos números ({n, m}) coincide entre n y m ocurrencias (inclusive) del patrón que califica. Todos los cuantificadores son *avariiciosos*; coincidirán con todas las ocurrencias del patrón que sean posibles, hasta que el patrón deje de coincidir. Si un cuantificador va seguido de un signo de interrogación (?), el cuantificador se vuelve *perezoso*, y coincidirá con la menor cantidad posible de ocurrencias, siempre y cuando haya una coincidencia exitosa.

16.16.2 Validación de la entrada del usuario con expresiones regulares

La aplicación para Windows en la figura 16.21 presenta un ejemplo más complejo que utiliza expresiones regulares para validar la información acerca del nombre, la dirección y el número telefónico que introduce un usuario.

Cuando un usuario hace clic en el botón **Aceptar**, el programa verifica para asegurarse que ninguno de los campos esté vacío (líneas 19-22). Si hay uno o más campos vacíos, el programa muestra un mensaje al usuario (líneas 25-26), indicando que todos los campos deben llenarse para que el programa pueda validar la información de entrada. La línea 27 llama al método Focus de `apellidoPaternoTextBox` para colocar el cursor en el control `apellidoPaternoTextBox`. Después, el programa sale del manejador de eventos (línea 28). Si no hay campos vacíos, las líneas 32-105 validan la entrada del usuario. Las líneas 32-40 validan el apellido mediante una llamada al método static Match de la clase Regex, y le pasan como argumentos la cadena a validar, junto con la expresión regular. El método Match devuelve un objeto Match. Este objeto contiene una propiedad llamada Success, la cual indica si el primer argumento del método Match coincidió con el patrón especificado por la expresión regular en el segundo argumento. Si el valor de Success es false (es decir, que no hubo coincidencia), las líneas 36-37 muestran un mensaje de error, la línea 38 establece el foco de vuelta al control `apellidoPaternoTextBox`, para que el usuario pueda volver a escribir la entrada y la línea 39 termina el manejador de eventos. Si hay una

```

1  // Fig. 16.21: Validacion.cs
2  // Valida la información del usuario usando expresiones regulares.
3  using System;
4  using System.Text.RegularExpressions;
5  using System.Windows.Forms;
6
7  public partial class ValidacionForm : Form
8  {
9      // constructor predeterminado
10     public ValidacionForm()
11     {
12         InitializeComponent();
13     } // fin del constructor

```

Figura 16.21 | Validación de la información del usuario, usando expresiones regulares. (Parte I de 4).


```

14
15 // maneja el evento clic de aceptarButton
16 private void aceptarButton_Click( object sender, EventArgs e )
17 {
18     // asegura que no haya controles TextBox vacíos
19     if ( apellidoPaternoTextBox.Text == "" || primerNombreTextBox.Text == "" ||
20         direccionTextBox.Text == "" || ciudadTextBox.Text == "" ||
21         estadoTextBox.Text == "" || codigoPostalTextBox.Text == "" ||
22         telefonoTextBox.Text == "" )
23     {
24         // muestra cuadro desplegable
25         MessageBox.Show( "Por favor llene todos los campos", "Error",
26             MessageBoxButtons.OK, MessageBoxIcon.Error );
27         apellidoPaternoTextBox.Focus(); // establece el foco en apellidoPaternoTextBox
28         return;
29     } // fin de if
30
31     // si el formato del apellido paterno es inválido, muestra un mensaje
32     if ( !Regex.Match( apellidoPaternoTextBox.Text,
33         "[A-Z][a-zA-Z]*$" ).Success )
34     {
35         // el apellido paterno es incorrecto
36         MessageBox.Show( "Apellido paterno inválido", "Mensaje",
37             MessageBoxButtons.OK, MessageBoxIcon.Error );
38         apellidoPaternoTextBox.Focus();
39         return;
40     } // fin de if
41
42     // si el formato del primer nombre es inválido, muestra un mensaje
43     if ( !Regex.Match( primerNombreTextBox.Text,
44         "[A-Z][a-zA-Z]*$" ).Success )
45     {
46         // el primer nombre es incorrecto
47         MessageBox.Show( "Primer nombre inválido", "Mensaje",
48             MessageBoxButtons.OK, MessageBoxIcon.Error );
49         primerNombreTextBox.Focus();
50         return;
51     } // fin de if
52
53     // si el formato de la dirección es inválido, muestra un mensaje
54     if ( !Regex.Match( direccionTextBox.Text,
55         @"^[0-9]+\s+([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$" ).Success )
56     {
57         // la dirección es incorrecta
58         MessageBox.Show( "Dirección inválida", "Mensaje",
59             MessageBoxButtons.OK, MessageBoxIcon.Error );
60         direccionTextBox.Focus();
61         return;
62     } // fin de if
63
64     // si el formato de la ciudad es inválido, muestra un mensaje
65     if ( !Regex.Match( ciudadTextBox.Text,
66         @"^[a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+$" ).Success )
67     {
68         // la ciudad es incorrecta
69         MessageBox.Show( "Ciudad inválida", "Mensaje",
70             MessageBoxButtons.OK, MessageBoxIcon.Error );
71         ciudadTextBox.Focus();
72         return;

```

Figura 16.21 | Validación de la información del usuario, usando expresiones regulares. (Parte 2 de 4).

```

73     } // fin de if
74
75     // si el formato del estado es inválido, muestra un mensaje
76     if ( !Regex.Match( estadoTextBox.Text,
77         @"^[a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+$" ).Success )
78     {
79         // el estado es incorrecto
80         MessageBox.Show( "Estado inválido", "Mensaje",
81             MessageBoxButtons.OK, MessageBoxIcon.Error );
82         estadoTextBox.Focus();
83         return;
84     } // fin de if
85
86     // si el formato del código postal es inválido, muestra un mensaje
87     if ( !Regex.Match( codigoPostalTextBox.Text, @"^\d{5}$" ).Success )
88     {
89         // el código postal es incorrecto
90         MessageBox.Show( "Código postal inválido", "Mensaje",
91             MessageBoxButtons.OK, MessageBoxIcon.Error );
92         codigoPostalTextBox.Focus();
93         return;
94     } // fin de if
95
96     // si el formato del número telefónico es inválido, muestra un mensaje
97     if ( !Regex.Match( telefonoTextBox.Text,
98         @"^[1-9]\d{2}-[1-9]\d{2}-\d{4}$" ).Success )
99     {
100         // el número telefónico es incorrecto
101         MessageBox.Show( "Número telefónico inválido", "Mensaje",
102             MessageBoxButtons.OK, MessageBoxIcon.Error );
103         telefonoTextBox.Focus();
104         return;
105     } // fin de if
106
107     // la información es válida, notifica al usuario y sale de la aplicación
108     this.Hide(); // oculta la ventana principal mientras se muestra el MessageBox
109     MessageBox.Show( "¡Gracias!", "Información correcta",
110         MessageBoxButtons.OK, MessageBoxIcon.Information );
111     Application.Exit();
112 } // fin del método aceptarButton_Click
113 } // fin de la clase ValidacionForm

```

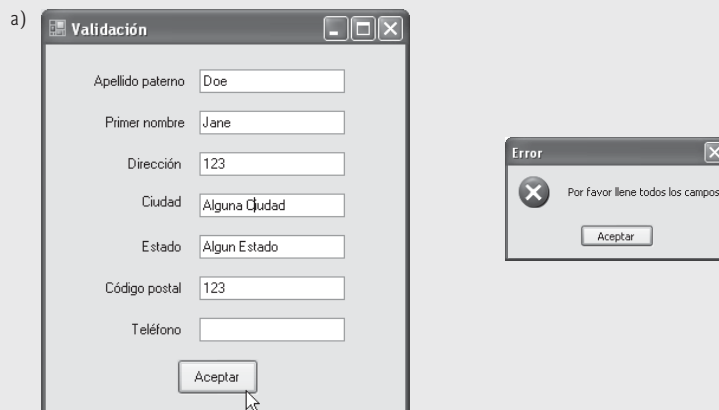


Figura 16.21 | Validación de la información del usuario, usando expresiones regulares. (Parte 3 de 4).

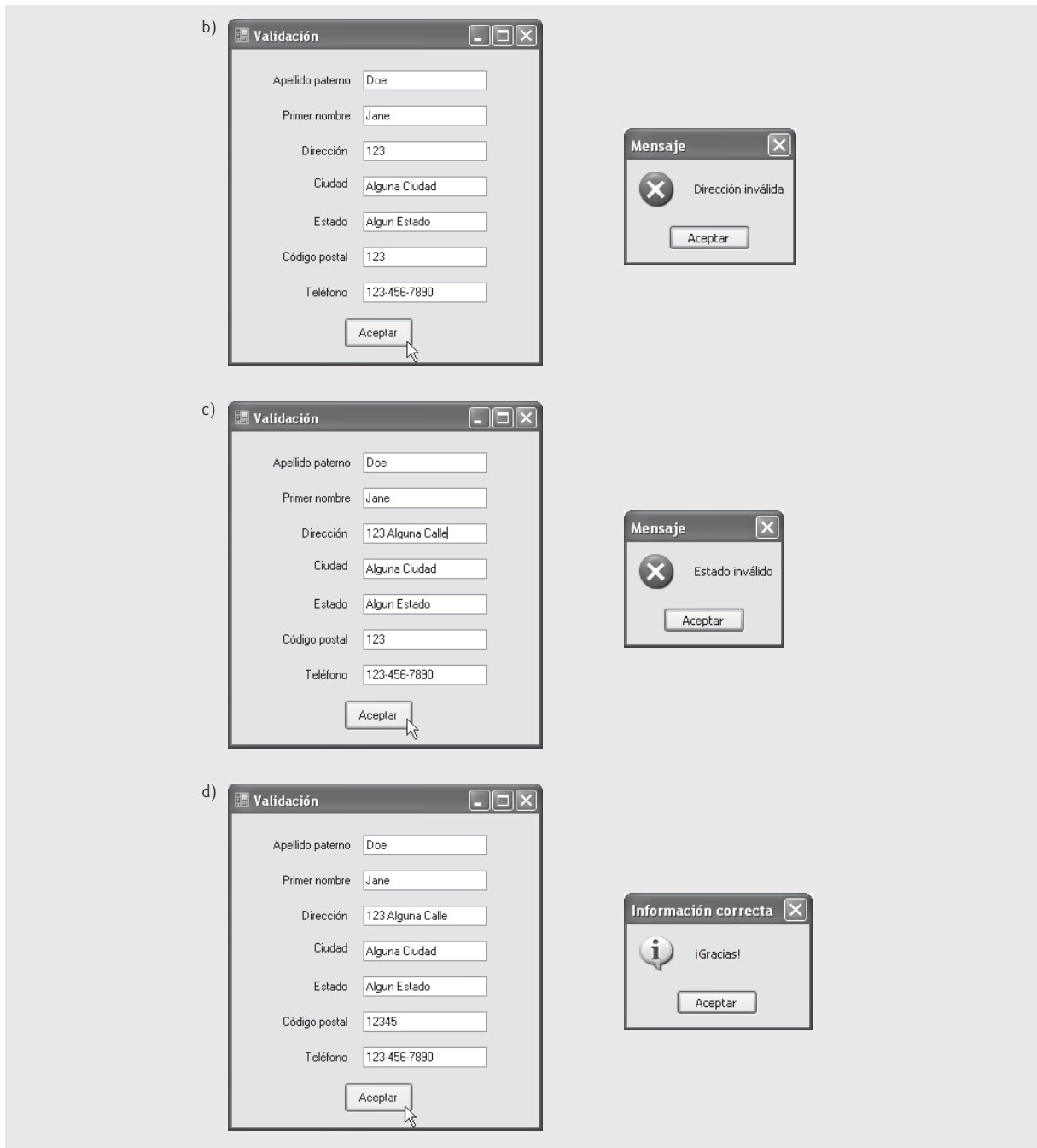


Figura 16.21 | Validación de la información del usuario, usando expresiones regulares. (Parte 4 de 4).

coincidencia, el manejador de eventos procede a validar el primer nombre. Este proceso continúa hasta que el manejador de eventos valida la entrada del usuario en todos los controles `TextBox`, o hasta que falla una validación. Si todos los campos contienen información válida, el programa muestra un cuadro de diálogo de mensaje indicando esto, y sale cuando el usuario cierra el cuadro de diálogo.

En el ejemplo anterior, buscamos sus cadenas en un objeto `string` que coincidieron con una expresión regular. En este ejemplo, queremos asegurar que todo el objeto `string` en cada control `TextBox` se conforme a

una expresión regular específica. Por ejemplo, queremos aceptar "Smith" como apellido, pero no "9@Smith#". En una expresión regular que comience con un carácter "^" y termine con un carácter "\$", los caracteres "^" y "\$" representan el principio y el final de un objeto `string`, respectivamente. Estos caracteres obligan a que una expresión regular devuelva una coincidencia sólo si todo el objeto `string` que se está procesando coincide con la expresión regular.

La expresión regular en la línea 33 utiliza la anotación de un rango entre corchetes para buscar coincidencias con una primera letra, seguida por letras en mayúsculas o en minúsculas; `a-z` coincide con cualquier letra en minúscula, y `A-Z` coincide con cualquier letra en mayúsculas. El cuantificador `*` indica que el segundo rango de caracteres puede ocurrir cero o más veces en el objeto `string`. En consecuencia, esta expresión coincide con cualquier objeto `string` que consiste de una letra mayúscula, seguida de cero o más letras adicionales.

La notación `\s` coincide con un carácter individual de espacio en blanco (líneas 55, 66 y 77). La expresión `\d{5}`, utilizada en el campo **Código postal**, coincide con cualquier combinación de cinco dígitos (línea 87). Observe que sin los caracteres "^" y "\$", la expresión regular coincidiría con cualquier combinación de cinco dígitos consecutivos en el objeto `string`. Al incluir los caracteres "^" y "\$", aseguramos que sólo se permitan códigos postales de cinco dígitos.

El carácter "|" (líneas 55, 66 y 77) coincide con la expresión a su izquierda o con la expresión a su derecha. Por ejemplo, `Ho!a (John | Jane)` coincide con `Ho!a John` y con `Ho!a Jane`. En la líneas 55, usamos el carácter "|" para indicar que la dirección puede contener una palabra de uno o más caracteres, o una palabra de uno o más caracteres, seguida por un espacio y otra palabra de uno o más caracteres. Observe el uso de los paréntesis para agrupar partes de la expresión regular. Los cuantificadores pueden aplicarse a los patrones encerrados entre paréntesis para crear expresiones regulares más complejas.

Los campos **Apellido paterno** y **Primer nombre** aceptan objetos `string` de cualquier longitud que comiencen con una letra mayúscula. La expresión regular para el campo **Dirección** (línea 55) coincide con un número de por lo menos un dígito, seguido de un espacio y después por una o más letras, o por una o más letras seguidas de un espacio, y otra serie de una o más letras. Por lo tanto, "10 Broadway" y "10 Main Street" son direcciones válidas. Según su formación actual, la expresión regular en la línea 55 no coincide con una dirección que no empiece con un número, o que tenga más de dos palabras. Las expresiones regulares para los campos **Ciudad** (línea 66) y **Estado** (línea 77) coinciden con cualquier palabra de por lo menos un carácter o, de manera alternativa, con dos palabras cualesquiera de por lo menos un carácter, si van separadas por un solo espacio. Esto significa que tanto `Waltham` como `West Newton` son coincidencias aceptables. De nuevo, estas expresiones regulares no aceptarían nombres que tuvieran más de dos palabras. La expresión regular para el campo **Código postal** (línea 87) asegura que el código postal sea un número de cinco dígitos. La expresión regular para el campo **Teléfono** (línea 98) indica que el número telefónico debe tener la forma `xxx-yyy-yyyy`, en donde las `x`'s representan el código de área y las `y`'s representan el número. La primera `x` y la primera `y` no pueden ser cero, según lo especificado mediante el rango `[1-9]` en cada caso.

16.16.3 Los métodos `Replace` y `Split` de `Regex`

Algunas veces es útil sustituir partes de un objeto `string` con otro, o dividir un objeto `string` de acuerdo con una expresión regular. Para este fin, la clase `Regex` cuenta con versiones `static` y de instancia de los métodos `Replace` y `Split`, que demostramos en la figura 16.22.

```

1 // Fig. 16.22: SustitucionRegex.cs
2 // Uso de los métodos Replace y Split de Regex.
3 using System;
4 using System.Text.RegularExpressions;
5
6 class SustitucionRegex
7 {
8     public static void Main()
9     {
10         string stringPrueba1 =

```

Figura 16.22 | Los métodos `Replace` y `Split` de `Regex`. (Parte 1 de 2).

```

11      "Esta oración termina con 5 estrellas *****";
12      string salida = "";
13      string stringPrueba2 = "1, 2, 3, 4, 5, 6, 7, 8";
14      Regex pruebaRegex1 = new Regex( @"\d" );
15      string[] resultado;
16
17      Console.WriteLine( "Cadena original: " +
18          stringPrueba1 );
19      stringPrueba1 = Regex.Replace( stringPrueba1, @"\*", "^" );
20      Console.WriteLine( "^ se sustituye por *: " + stringPrueba1 );
21      stringPrueba1 = Regex.Replace( stringPrueba1, "estrellas",
22          "circunflejos" );
23      Console.WriteLine( "\"circunflejos\" se sustituyeron por \"estrellas\": " +
24          stringPrueba1 );
25      Console.WriteLine( "Todas las palabras se sustituyeron por \"palabra\": " +
26          Regex.Replace( stringPrueba1, @"\w+", "palabra" ) );
27      Console.WriteLine( "\nCadena original: " + stringPrueba2 );
28      Console.WriteLine( "Sustituye los primeros 3 dígitos por \"dígito\": " +
29          pruebaRegex1.Replace( stringPrueba2, "dígito", 3 ) );
30      Console.Write( "se dividió la cadena en las comas [" );
31
32      resultado = Regex.Split( stringPrueba2, @",\s" );
33
34      foreach ( string resultString in resultado )
35          salida += "\"" + resultString + "\", ";
36
37      // Elimina ", " al final de la cadena de salida
38      Console.WriteLine( salida.Substring( 0, salida.Length - 2 ) + "]" );
39  } // fin del método Main
40 } // fin de la clase SustitucionRegex

```

```

Cadena original: Esta oración termina con 5 estrellas *****
^ se sustituye por *: Esta oración termina con 5 estrellas ^^^^^
"circunflejos" se sustituyeron por "estrellas": Esta oración termina con 5 circunflejos
^^^^^
Todas las palabras se sustituyeron por "palabra": palabra palabra palabra palabra palabra
palabra ^^^^^

Cadena original: 1, 2, 3, 4, 5, 6, 7, 8
Sustituye los primeros 3 dígitos por "dígito": dígito, dígito, dígito, 4, 5, 6, 7, 8
se dividió la cadena en las comas ["1", "2", "3", "4", "5", "6", "7", "8"]

```

Figura 16.22 | Los métodos Replace y Split de Regex. (Parte 2 de 2).

El método `Replace` sustituye texto en un objeto `string` con nuevo texto, en cualquier parte en donde el objeto `string` original coincide con una expresión regular. Utilizamos dos versiones de este método en la figura 16.22. La primera versión (línea 19) es `static` y recibe tres parámetros: el objeto `string` a modificar, el objeto `string` que contiene la expresión regular con la que debe coincidir y el objeto `string` de sustitución. Aquí, `Replace` sustituye todas las instancias de `"*"` en `stringPrueba1` con `"^"`. Observe que la expresión regular (`@\"\\*\"`) va antes del carácter `*` con una barra diagonal inversa, `\\`. Por lo general, `*` es un cuantificador que indica que una expresión regular debe coincidir con cualquier número de ocurrencias del patrón anterior. No obstante, en la línea 19 queremos encontrar todas las ocurrencias del carácter literal `*`; para ello, debemos escapar el carácter `*` con el carácter `\\`. Al escapar un carácter de una expresión regular con una `\\`, indicamos al motor de coincidencias de expresiones regulares que busque el carácter `*`, en vez de usarlo como cuantificador.

La segunda versión del método `Replace` (línea 29) es un método de instancia que utiliza la expresión regular que se pasa al constructor para `pruebaRegex1` (línea 14), para realizar la operación de sustitución. La línea 14 instancia `pruebaRegex1` con el argumento `@\"\\d\"`. La llamada al método de instancia `Replace` en la línea 29 recibe

tres argumentos: un objeto `string` a modificar, un objeto `string` que contiene el texto de sustitución y un `int` que especifica el número de sustituciones a realizar. En este caso, la línea 29 sustituye las tres primeras instancias de un dígito ("`\d`") en `stringPrueba2` con el texto "dígito".

El método `Split` divide un objeto `string` en varias subcadenas. El objeto `string` original se descompone en los delimitadores que coinciden con una expresión regular especificada. El método `String` devuelve un arreglo que contiene las subcadenas. En la línea 32, utilizamos la versión `static` del método `Split` para separar un objeto `string` de enteros separados por comas. El primer argumento es el objeto `string` a dividir; el segundo es la expresión regular que representa el delimitador. La expresión regular `@",\s"` separa las subcadenas en cada coma. Al haber coincidencias con cualquier carácter de espacio en blanco (`\s*` en la expresión regular), eliminamos los espacios adicionales de las subcadenas resultantes.

16.17 Conclusión

En este capítulo, aprendió acerca de las herramientas de procesamiento de cadenas y caracteres de la FCL. Vimos las generalidades acerca de los fundamentos de los caracteres y las cadenas. Vio cómo determinar la longitud de las cadenas, copiarlas, acceder a los caracteres individuales, buscar en ellas, obtener subcadenas a partir de cadenas más grandes, compararlas, concatenarlas, sustituir caracteres en cadenas y convertirlas a letras mayúsculas o minúsculas.

Mostramos cómo utilizar la clase `StringBuilder` para crear cadenas en forma dinámica. Aprendió a determinar y especificar el tamaño de un objeto `StringBuilder`, y cómo adjuntar, insertar, eliminar y sustituir caracteres en un objeto `StringBuilder`. Después, presentamos los métodos para evaluar caracteres de tipo `Char`, que permiten a un programa determinar cuándo un carácter es un dígito, una letra, una letra en minúscula, mayúscula, un signo de puntuación o un símbolo distinto de un signo de puntuación, y los métodos para convertir un carácter a mayúscula o minúscula.

Por último, hablamos sobre las clases `Regex` y `Match` del espacio de nombres `System.Text.RegularExpressions` y los símbolos que se utilizan para formar expresiones regulares. Aprendió a buscar patrones en una cadena y a relacionar cadenas completas con patrones mediante los métodos `Match` y `Matches` de `Regex`, cómo sustituir caracteres en una cadena mediante el método `Replace` de `Regex` y cómo dividir cadenas en ciertos delimitadores mediante el método `Split` de `Regex`. En el siguiente capítulo, aprenderá a agregar gráficos y otras herramientas multimedia a sus aplicaciones para Windows.

17

Gráficos y multimedia

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Comprender los contextos de gráficos y los objetos de gráficos.
- Manipular los colores y las fuentes.
- Comprender y utilizar métodos de `Graphics` de GDI para dibujar líneas, rectángulos, objetos `string` e imágenes.
- Utilizar la clase `Image` para manipular y visualizar imágenes.
- Dibujar figuras complejas a partir de figuras simples, mediante la clase `GraphicsPath`.
- Utilizar el Reproductor de Windows Media para reproducir audio o video en una aplicación en C#.
- Utilizar el Agente de Microsoft para agregar personajes animados interactivos a una aplicación en C#.

Una imagen vale más que mil palabras.

—Proverbio chino

Hay que tratar a la naturaleza en términos del cilindro, la esfera, el cono, todo en perspectiva.

—Paul Cezanne

Nada se vuelve real hasta que se experimenta; incluso hasta un proverbio no lo será para usted sino hasta que su vida lo haya ilustrado.

—John Keats

Una imagen me muestra de un vistazo lo que requiere docenas de páginas de un libro para explicar.

—Ivan Sergeyevich

- 17.1 Introducción
- 17.2 Las clases de dibujo y el sistema de coordenadas
- 17.3 Contextos de gráficos y objetos Graphics
- 17.4 Control de los colores
- 17.5 Control de las fuentes
- 17.6 Dibujo de líneas, rectángulos y óvalos
- 17.7 Dibujo de arcos
- 17.8 Dibujo de polígonos y polilíneas
- 17.9 Herramientas de gráficos avanzadas
- 17.10 Introducción a multimedia
- 17.11 Carga, visualización y escalado de imágenes
- 17.12 Animación de una serie de imágenes
- 17.13 Reproductor de Windows Media
- 17.14 Microsoft Agent
- 17.15 Conclusión

17.1 Introducción

En este capítulo veremos las herramientas de C# para dibujar figuras bidimensionales, y para controlar colores y fuentes. C# soporta gráficos que permiten a los programadores mejorar sus aplicaciones Windows en forma visual. La FCL contiene muchas herramientas sofisticadas de dibujo como parte del espacio de nombres `System.Drawing` y los demás espacios de nombres que conforman la **GDI+** de recursos de .NET. GDI+ es una interfaz de programación de aplicaciones (API) que proporciona clases para crear gráficos vectoriales de dos dimensiones (una manera de describir los gráficos, para poder manipularlos con facilidad mediante técnicas de alto rendimiento), manipular fuentes e insertar imágenes.

Comenzaremos con una introducción a las herramientas de dibujo del .NET Framework. Después presentaremos herramientas de dibujo más poderosas, como las que se utilizan para modificar los estilos de las líneas utilizadas para dibujar figuras y controlar los colores y patrones de las figuras rellenas.

Más adelante en este capítulo, exploraremos técnicas para manipular imágenes y crear animaciones uniformes. También hablaremos sobre la clase `Image`, la cual puede almacenar y manipular imágenes de varios formatos. Explicaremos cómo combinar las herramientas de representación de gráficos que cubriremos en las primeras secciones del capítulo, con las herramientas para manipulación de imágenes. El capítulo termina con varios ejemplos de multimedia en los que usted creará una animación, usará el control Windows Media Player y Microsoft Agent: una tecnología para agregar personajes animados interactivos a las aplicaciones o páginas Web.

17.2 Las clases de dibujo y el sistema de coordenadas

La figura 17.1 ilustra una porción del espacio de nombres `System.Drawing`, incluyendo varias clases y estructuras de gráficos que veremos en este capítulo. Los espacios de nombres `System.Drawing` y `System.Drawing.Drawing2D` contienen los componentes más utilizados de la GDI+.

La clase `Graphics` contiene los métodos utilizados para dibujar objetos `string`, líneas, rectángulos y otras figuras en un objeto `Control`. Los métodos de dibujo de la clase `Graphics` por lo general requieren de un objeto **Pen** o **Brush** para representar una figura específica. El objeto `Pen` dibuja los contornos de las figuras; el objeto `Brush` dibuja objetos sólidos.

La estructura `Color` contiene numerosas propiedades `static`, las cuales establecen los colores de varios componentes gráficos, además de métodos que permiten a los usuarios crear nuevos colores. La clase `Font` contiene propiedades que definen fuentes únicas. La clase `FontFamily` contiene métodos para obtener información sobre las fuentes.

Para empezar a dibujar en C#, primero debemos comprender el *sistema de coordenadas* de la GDI+ (figura 17.2), un esquema para identificar cada uno de los puntos en la pantalla. De manera predeterminada, la esquina

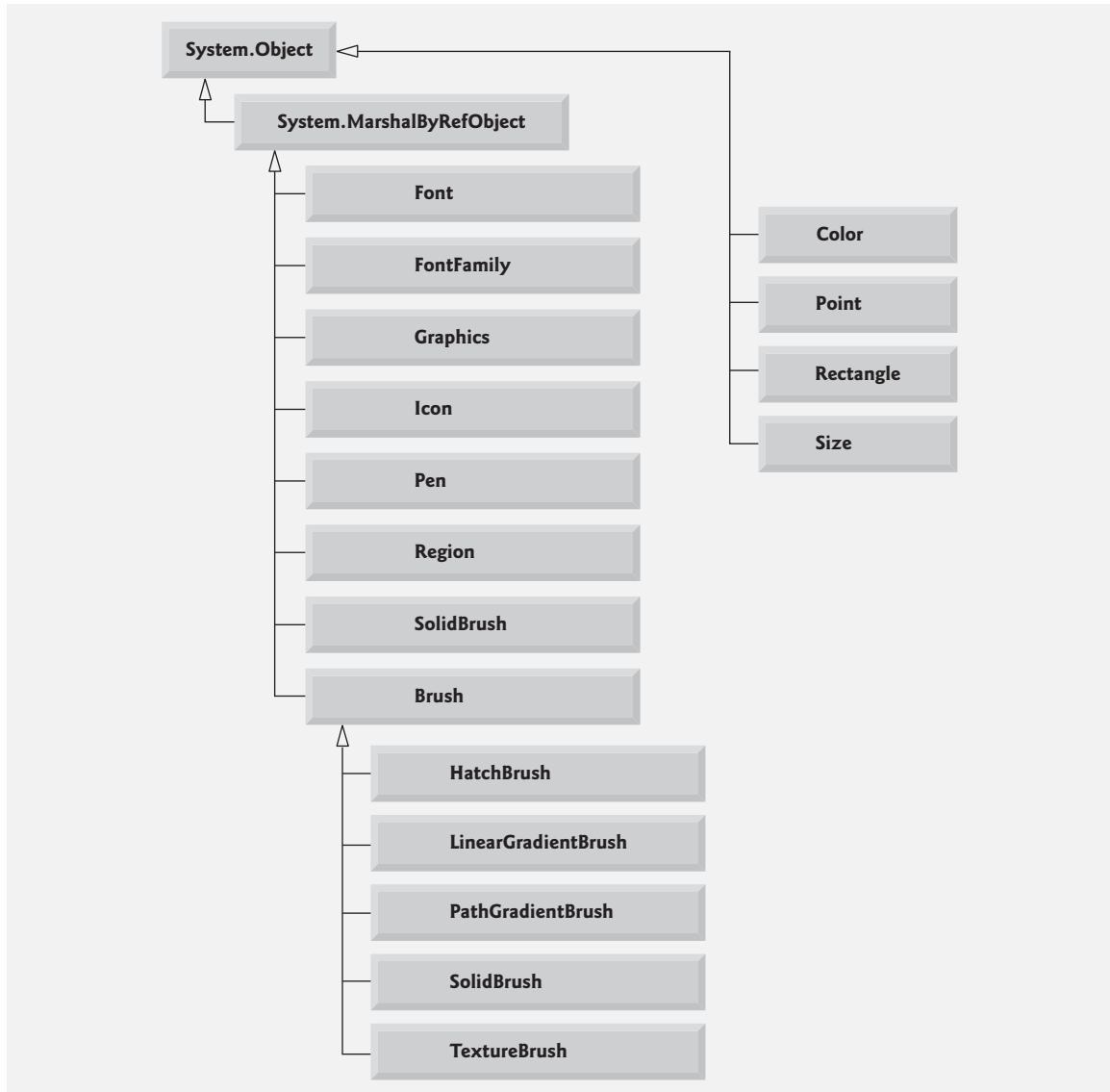


Figura 17.1 | Clases y estructuras del espacio de nombres System.Drawing.

superior izquierda de un componente de la GUI (como un objeto Panel o Form) tiene las coordenadas (0, 0). Un par de coordenadas tiene tanto una *coordenada x* (la *coordenada horizontal*) como una *coordenada y* (la *coordenada vertical*). La coordenada *x* es la distancia horizontal (a la derecha) a partir de la esquina superior izquierda. La coordenada *y* es la distancia vertical (hacia abajo) a partir de la esquina superior izquierda. El *eje x* define a cada una de las coordenadas horizontales, y el *eje y* define a cada una de las coordenadas verticales. Los programadores posicionan texto y figuras en la pantalla especificando sus coordenadas (*x, y*). Las unidades de las coordenadas se miden en píxeles (“elementos de imagen”), que son las unidades más pequeñas de *resolución* en un monitor de visualización.



Tip de portabilidad 17.1

Los distintos monitores de visualización tienen distintas resoluciones, por lo que la densidad de los píxeles en dichos monitores es variable. Esto puede hacer que los tamaños de los gráficos aparezcan de manera distinta en varios monitores diferentes.

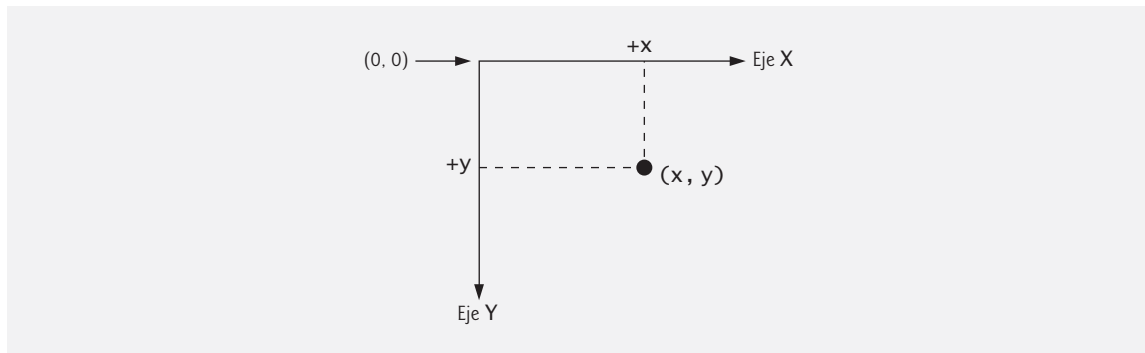


Figura 17.2 | Sistema de coordenadas de la GDI+. Las unidades se miden en píxeles.

El espacio de nombres `System.Drawing` cuenta con varias estructuras que representan tamaños y ubicaciones en el sistema de coordenadas. La estructura ***Point*** representa las coordenadas *x-y* de un punto en un plano bidimensional. La estructura ***Rectangle*** define la anchura y altura de carga de una figura rectangular. La estructura ***Size*** representa la anchura y la altura de una figura.

17.3 Contextos de gráficos y objetos Graphics

Un *contexto de gráficos* en C# representa una superficie de dibujo que permite dibujar en la pantalla. Un objeto ***Graphics*** administra un contexto gráfico; para ello controla la manera en que se dibuja la información. Los objetos ***Graphics*** contiene métodos para dibujar, manipular fuentes, manipular colores y demás acciones relacionadas con los gráficos. Cada clase derivada de `System.Windows.Forms.Form` hereda un método virtual llamado ***OnPaint***, en el cual se llevan a cabo la mayoría de las operaciones con gráficos. Los argumentos para el método ***OnPaint*** incluyen un objeto ***PaintEventArgs***, del cual podemos obtener un objeto ***Graphics*** para el formulario (Form). Debemos obtener el objeto ***Graphics*** en cada llamada al método, debido a que las propiedades del contexto de gráficos que representa el objeto de gráficos pueden cambiar. El método ***OnPaint*** activa el evento ***Paint*** de ***Control***.

Al dibujar en un formulario, puede redefinir el método ***OnPaint*** para extraer un objeto ***Graphics*** del argumento ***PaintEventArgs*** o para crear un nuevo objeto ***Graphics*** asociado con la superficie apropiada. Más adelante en este capítulo demostraremos estas técnicas de dibujo en C#.

Para redefinir el método ***OnPaint*** heredado, utilice el siguiente encabezado para el método:

```
protected override void OnPaint ( PaintEventArgs e )
```

A continuación, extraiga el objeto ***Graphics*** entrante del argumento ***PaintEventArgs***, como en:

```
Graphics objetoGraficos = e.Graphics;
```

Ahora, la variable ***objetoGraficos*** puede usarse para dibujar figuras y objetos ***string*** en el formulario.

Una llamada al método ***OnPaint*** genera el evento ***Paint***. En vez de redefinir el método ***OnPaint***, los programadores pueden agregar un manejador de eventos para el evento ***Paint***. Visual Studio .NET genera el manejador de eventos ***Paint*** de esta forma:

```
protected void MiManejadorEventos_Paint(  
    object sender, PaintEventArgs e )
```

No es muy común que los programadores llamen al método ***OnPaint*** directamente, ya que el dibujo de gráficos es un *proceso controlado por eventos*. Un evento (como cubrir, descubrir o cambiar de tamaño una ventana) llama al método ***OnPaint*** de ese formulario. De manera similar, cuando se muestra algún control (como ***TextBox*** o ***Label***), se hace una llamada al método ***OnPaint*** de ese control.

Podemos forzar una llamada al método ***OnPaint*** mediante una llamada al método ***Invalidate*** de ***Control***. Este método actualiza un control y vuelve a pintar de manera implícita todos sus componentes gráficos. La clase ***Control*** tiene varios métodos ***Invalidate*** sobrecargados que permiten a los programadores actualizar las partes de un control.



Tip de rendimiento 17.1

Llamar al método `Invalidate` para actualizar el `Control` puede ser ineficiente, si sólo una pequeña parte de un `Control` visita actualizarse. Al llamar a `Invalidate` con un parámetro `Rectangle` sólo se actualiza el área designada por el rectángulo. Esto mejora el rendimiento del programa.

Los controles, tales como `Label` y `Button`, no tiene sus propios contextos de gráficos, pero usted puede crearlos. Para dibujar en un control, primero cree un objeto de gráficos mediante una invocación al método `CreateGraphics` del control, como en:

```
Graphics objetoGraficos = nombreControl.CreateGraphics();
```

Ahora puede utilizar los métodos que se proporcionan en la clase `Graphics` para dibujar en el control.

17.4 Control de los colores

Los colores pueden mejorar la apariencia de un programa y ayudar a transmitir el significado. Por ejemplo, una luz roja del semáforo indica alto, la luz amarilla indica precaución y la luz verde indica que se puede pasar. La estructura `Color` define los métodos y constantes que se utilizan para manipular los colores.

Cualquier color puede crearse a partir de una combinación de los componentes alfa, rojo, verde y azul (a los cuales se les llama **valores ARGB**). Los cuatro componentes ARGB son valores tipo `byte` que representan valores enteros en el rango de 0 a 255. El valor alfa determina la opacidad del color. Por ejemplo, el valor alfa cero representa un color transparente, y el valor 255 representa un color opaco. Los valores alfa entre 0 y 255 producen un efecto ponderado de mezcla del **valor RGB** del color con el de cualquier color de fondo, lo que ocasiona un efecto semitransparente. El primer número en el valor RGB define la cantidad de rojo en el color, el segundo define la cantidad de verde y el tercero define la cantidad de azul. Entre mayor sea el valor, mayor será la cantidad de ese color específico. C# permite a los programadores elegir de entre casi 17 millones de colores. Si cierta computadora no puede mostrar todos estos colores, mostrará el color más cercano al que se le especificó. La figura 17.3 muestra un resumen de algunas constantes predefinidas de `Color` (todas son `public` y `static`), y la figura 17.4 describe varios métodos y propiedades de `Color`. Para obtener una lista completa de constantes, métodos y propiedades predefinidas de `Color`, consulte la documentación en línea para la estructura `Color` (msdn2.microsoft.com/en-us/library/system.drawing.color en inglés; msdn2.microsoft.com/es-es/library/system.drawing.color (VS. 80).aspx en español).

La tabla en la figura 17.4 describe dos llamadas al método `FromArgb`. Una de ellas recibe tres argumentos `int`, y la otra recibe cuatro argumentos `int` (los valores de todos los argumentos deben estar entre 0 y 255, inclusive). Ambas reciben argumentos `int` que especifican la cantidad de rojo, verde y azul. La versión sobrecargada también permite al usuario especificar el componente alfa; la versión con tres argumentos utiliza el valor predeterminado de 255 para el componente alfa (opaco). Ambos métodos devuelven un objeto `Color`. Las propiedades **A**, **R**, **G** y **B** de `Color` devuelven `bytes` que representan valores `int` de 0 a 255, y corresponden a las cantidades de alfa, rojo, verde y azul, respectivamente.

Constantes en la estructura <code>Color</code>	Valor RGB	Constantes en la estructura <code>Color</code>	Valor RGB
Orange	255, 200, 0	White	255, 255, 255
Pink	255, 175, 175	Gray	128, 128, 128
Cyan	0, 255, 255	DarkGray	64, 64, 64
Magenta	255, 0, 255	Red	255, 0, 0
Yellow	255, 255, 0	Green	0, 255, 0
Black	0, 0, 0	Blue	0, 0, 255

Figura 17.3 | Constantes `static` de la estructura `Color` y sus valores RGB.

Métodos y propiedades de la estructura Color	Descripción
<i>Métodos comunes</i>	
FromArgb	Un método static que crea un color con base en los valores rojo, verde y azul que se expresan como valores int de 0 a 255. La versión sobrecargada permite especificar valores de alfa, rojo, verde y azul.
FromName	Un método static que crea un color a partir de un nombre que recibe como string.
<i>Propiedades comunes</i>	
A	Un byte entre 0 y 255, que representa al componente alfa.
R	Un byte entre 0 y 255, que representa al componente rojo.
G	Un byte entre 0 y 255, que representa al componente verde.
B	Un byte entre 0 y 255, que representa al componente azul.

Figura 17.4 | Métodos y propiedades de la estructura Color.

Los programadores dibujan figuras y objetos string con objetos Brush y Pen. Una pluma (Pen) funciona de manera similar a una pluma ordinaria; se utiliza para dibujar líneas. La mayoría de los métodos de dibujo requieren un objeto Pen. Los constructores sobrecargados de Pen permiten a los programadores especificar los colores y los grosores de las líneas que desean dibujar. El espacio de nombres System.Drawing también cuenta con una clase llamada Pens, que contiene objetos Pen predefinidos.

Todas las clases que se derivan de la clase abstracta Brush definen objetos que colorean el interior de las figuras gráficas. Por ejemplo, el constructor de SolidBrush recibe un objeto Color: el color a dibujar. En la mayoría de los métodos Fill, los objetos Brush llenan un espacio con un color, patrón o imagen. La figura 17.5 muestra un resumen de los diversos objetos Brush y sus funciones.

Manipulación de los colores

La figura 17.6 demuestra varios de los métodos y propiedades descritas en la figura 17.4. Además, muestra dos rectángulos traslapados, que le permiten experimentar con los valores del color, los nombres de los colores y los valores del componente alfa (para la transparencia).

Clase	Descripción
HatchBrush	Llena una región con un patrón. Este patrón se define mediante un miembro de la enumeración HatchStyle, un color de primer plano (con el cual se dibuja el patrón) y un color de fondo.
LinearGradientBrush	Llena una región con una mezcla gradual que va de un color a otro. Los degradados lineales se definen a lo largo de una línea. Pueden especificarse mediante los dos colores, el ángulo del degradado y ya sea el grosor de un rectángulo o dos puntos.
SolidBrush	Llena una región con un color especificado mediante un objeto Color.
TextureBrush	Llena una región mediante la repetición de un objeto Image especificado a lo largo de la superficie.

Figura 17.5 | Clases que se derivan de la clase Brush.

```

1  // Fig 17.6: MostrarColoresForm.cs
2  // Demostración del valor y el componente alfa de un color.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  public partial class MostrarColoresForm : Form
8  {
9      // color para rectángulo posterior
10     private Color colorPosterior = Color.Wheat;
11
12     // color para rectángulo frontal
13     private Color colorFrontal = Color.FromArgb( 100, 0, 0, 255 );
14
15     // constructor predeterminado
16     public MostrarColoresForm()
17     {
18         InitializeComponent();
19     } // fin del constructor
20
21     // redefine el método OnPaint de Form
22     protected override void OnPaint( PaintEventArgs e )
23     {
24         Graphics objetoGraficos = e.Graphics; // obtiene objeto de gráficos
25
26         // crea brocha de texto
27         SolidBrush brochaTexto = new SolidBrush( Color.Black );
28
29         // crea brocha sólida
30         SolidBrush brocha = new SolidBrush( Color.White );
31
32         // dibuja fondo blanco
33         objetoGraficos.FillRectangle( brocha, 4, 4, 275, 180 );
34
35         // muestra el nombre de colorPosterior
36         objetoGraficos.DrawString( colorPosterior.Name, this.Font,
37             brochaTexto, 40, 5 );
38
39         // establece el color de brocha y muestra rectángulo posterior
40         brocha.Color = colorPosterior;
41         objetoGraficos.FillRectangle( brocha, 45, 20, 150, 120 );
42
43         // muestra valores Argb del color frontal
44         objetoGraficos.DrawString( "Alfa: " + colorFrontal.A +
45             " Rojo: " + colorFrontal.R + " Verde: " + colorFrontal.G +
46             " Azul: " + colorFrontal.B, this.Font, brochaTexto, 55, 165 );
47
48         // establece el color de brocha y muestra rectángulo frontal
49         brocha.Color = colorFrontal;
50         objetoGraficos.FillRectangle( brocha, 65, 35, 170, 130 );
51     } // fin del método OnPaint
52
53     // maneja el evento Click de nombreColorButton
54     private void nombreColorButton_Click( object sender, EventArgs e )
55     {
56         // establece colorPosterior al color especificado en el cuadro de texto
57         colorPosterior = Color.FromName( nombreColorTextBox.Text );
58
59         Invalidate(); // actualiza el formulario

```

Figura 17.6 | Demostración del valor y el componente alfa de un color. (Parte 1 de 2).

```

60 } // fin del método nombreColorButton_Click
61
62 // maneja el evento Click de valorColorButton
63 private void valorColorButton_Click( object sender, EventArgs e )
64 {
65     // obtiene nuevo color frontal de los cuadros de texto
66     colorFrontal = Color.FromArgb(
67         Convert.ToInt32( alfaTextBox.Text ),
68         Convert.ToInt32( rojoTextBox.Text ),
69         Convert.ToInt32( verdeTextBox.Text ),
70         Convert.ToInt32( azulTextBox.Text ) );
71
72     Invalidate(); // actualiza el formulario
73 } // fin del método valorColorButton_Click
74 } // fin de la clase MostrarColoresForm

```

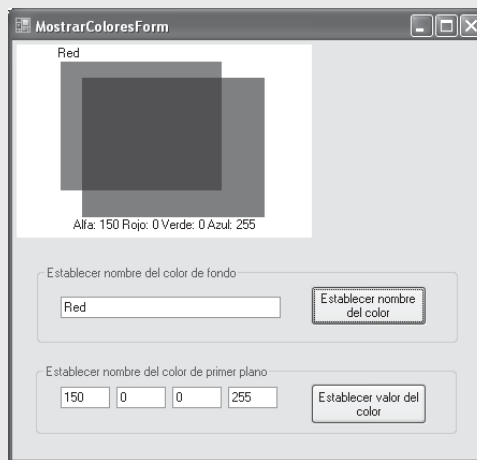


Figura 17.6 | Demostración del valor y el componente alfa de un color. (Parte 2 de 2).

Cuando la aplicación comienza a ejecutarse, se muestra su formulario. Esto produce una llamada al método `OnPaint` de `MostrarColoresForm`, para pintar el contenido del formulario. La línea 24 obtiene una referencia al objeto `Graphics` de `PaintEventArgs` e y lo asigna a `objetoGraficos`. Las líneas 27 y 30 crean un objeto `SolidBrush` en blanco y negro para dibujar figuras sólidas en el formulario. La clase **`SolidBrush`** se deriva de la clase base abstracta `Brush`, por lo que puede pasarse un objeto `SolidBrush` a cualquier método que espera un parámetro `Brush`.

La línea 33 utiliza el método `FillRectangle` de `Graphics` para dibujar un rectángulo sólido color blanco, usando el objeto `SolidBrush` creado en la línea 30. `FillRectangle` recibe como parámetros un objeto `Brush`, las coordenadas x - y de la esquina superior izquierda del rectángulo, y la anchura y altura del rectángulo. Las líneas 36-37 muestran la propiedad `Name` de `colorPosterior` mediante el método **`DrawString`** de `Graphics`. Existen varios métodos `DrawString` sobrecargados; la versión que se demuestra en las líneas 36-37 recibe como argumentos el objeto `string` a mostrar, el objeto `Font` a mostrar, el objeto `Brush` a usar para dibujar y las coordenadas x - y de la ubicación para el primer carácter del objeto `string`.

Las líneas 40-41 asignan el valor de `colorPosterior` a la propiedad `Color` de brocha y muestran un rectángulo. Las líneas 44-46 extraen y muestran los valores ARGB de `colorFrontal`, y dibujan una cadena que contiene esos valores. Las líneas 49-50 asignan el valor de `colorFrontal` a la propiedad `Color` de brocha, y después dibujan un rectángulo relleno con el `colorFrontal`, que traslapa el rectángulo que se dibujó en la línea 41.

El manejador de eventos `nombreColorButton_Click` de `Button` (líneas 54-60) utiliza el método **`FromName`** de la clase `Color` para crear un nuevo objeto `Color` a partir del `nombreColor` que introduce el usuario en un control `TextBox`. Este `Color` se asigna a `colorPosterior` (línea 57). Después, la línea 59 invoca al

método `Invalidate` de `Form` para indicar que el formulario debe volver a pintarse, lo cual produce una llamada a `OnPaint` para actualizar el formulario en la pantalla.

El manejador de eventos `valorColorButton_Click` de `Button` (líneas 63-73) utiliza el método `FromArgb` de `Color` para construir un nuevo objeto `Color`, a partir de los valores ARGB que especifica el usuario mediante los controles `TextBox`, y después asigna el `Color` recién creado a `colorFrontal`. La línea 72 invoca al método `Invalidate` de `Form` para indicar que se debe volver a pintar el formulario, lo cual produce una llamada a `OnPaint` para actualizar el formulario en la pantalla.

Si el usuario asigna un valor alfa entre 0 y 255 para el `colorFrontal`, los efectos de la mezcla con el componente alfa son aparentes. En los resultados de ejemplo, el rectángulo posterior color rojo se mezcla con el rectángulo frontal color azul para crear un color morado, en donde los dos rectángulos se traslapan. Observe que no puede cambiar las características de un objeto `Color` existente. Para usar un color distinto, cree un nuevo objeto `Color`.

Uso del cuadro de diálogo `ColorDialog` para seleccionar colores de una paleta de colores

El componente predefinido **`ColorDialog`** de la GUI es un cuadro de diálogo que permite a los usuarios seleccionar un color de una paleta de colores disponibles, o crear colores personalizados. La figura 17.7 demuestra el cuadro de diálogo `ColorDialog`. Cuando un usuario selecciona un color y oprime **Aceptar**, la aplicación extrae la selección del usuario a través de la propiedad **`Color`** de `ColorDialog`.

```

1  // Fig. 17.7: MostrarComplejoColores.cs
2  // El cuadro de diálogo ColorDialog se usa para cambiar el color de fondo y del texto.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // permite a los usuarios cambiar los colores usando un ColorDialog
8  public partial class MostrarComplejoColoresForm : Form
9  {
10     // crea objeto ColorDialog
11     private static ColorDialog selectorColor = new ColorDialog();
12
13     // constructor predeterminado
14     public MostrarComplejoColoresForm()
15     {
16         InitializeComponent();
17     } // fin del constructor
18
19     // cambia el color del texto
20     private void colorTextoButton_Click( object sender, EventArgs e )
21     {
22         // obtiene el color elegido
23         DialogResult resultado = selectorColor.ShowDialog();
24
25         if ( resultado == DialogResult.Cancel )
26             return;
27
28         // asigna el color de primer plano al resultado del cuadro de diálogo
29         colorFondoButton.ForeColor = selectorColor.Color;
30         colorTextoButton.ForeColor = selectorColor.Color;
31     } // fin del método colorTextoButton_Click
32
33     // cambia el color de fondo
34     private void colorFondoButton_Click( object sender, EventArgs e )
35     {
36         // muestra ColorDialog y obtiene el resultado

```

Figura 17.7 | El cuadro de diálogo `ColorDialog` se utiliza para cambiar el color de fondo y del texto. (Parte 1 de 2).


```

37     selectorColor.FullOpen = true;
38     DialogResult resultado = selectorColor.ShowDialog();
39
40     if ( resultado == DialogResult.Cancel )
41         return;
42
43     // establece el color de fondo
44     this.BackColor = selectorColor.Color;
45 } // fin del método colorFondoButton_Click
46 } // fin de la clase MostrarComplejoColoresForm

```

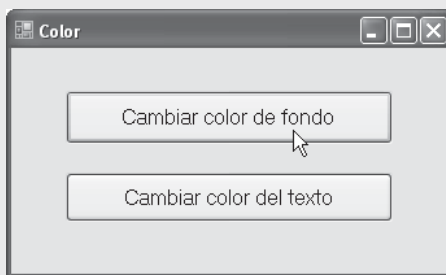
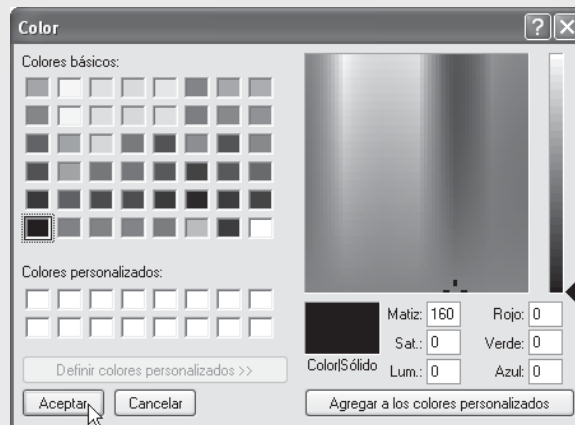


Figura 17.7 | El cuadro de diálogo `ColorDialog` se utiliza para cambiar el color de fondo y del texto. (Parte 2 de 2.)

La GUI para esta aplicación contiene dos controles `Button`. El control `colorFondoButton` permite al usuario cambiar el color de fondo del formulario. El control `colorTextoButton` permite al usuario cambiar los colores de texto de los botones. La línea 11 crea un cuadro de diálogo `private static ColorDialog` llamado `selectorColor`, que se utiliza en los manejadores de eventos para ambos controles `Button`.

Las líneas 20-31 definen el manejador de eventos `colorTextoButtonClick`, que invoca al método `ShowDialog` de `selectorColor` (línea 23) para mostrar el cuadro de diálogo. La propiedad `Color` del cuadro de diálogo almacena la selección del usuario. Las líneas 29-30 establecen el color de texto de ambos botones al color seleccionado.

Las líneas 34-45 definen el manejador de eventos `colorFondoButtonClick`, el cual modifica el color de fondo del formulario, enviando su propiedad `BackColor` a la propiedad `Color` del cuadro de diálogo (línea 44). El método establece la propiedad **`FullOpen`** de `ColorDialog` a `true` (línea 37), de manera que el cuadro de diálogo muestre todos los colores disponibles, como se muestra en la captura de pantalla de la figura 17.7. Cuando `FullOpen` es `false`, en el cuadro de diálogo sólo aparecen las muestras de colores.

Los usuarios no están restringidos a las 48 muestras de colores de `ColorDialog`. Para crear un color personalizado, los usuarios pueden hacer clic en cualquier parte del rectángulo grande de `ColorDialog`, el cual muestra varias sombras de colores. Ajuste el control deslizable, el matiz y las demás características para refinar el color.

Cuando termine, haga clic en el botón **Agregar a los colores personalizados**, para añadir el color personalizado a uno de los cuadros en la sección **Colores personalizados** del cuadro de diálogo. Al hacer clic en **Aceptar** se establece la propiedad `Color` del cuadro de diálogo `ColorDialog` a ese color.

17.5 Control de las fuentes

En esta sección hablaremos de los métodos y las constantes relacionadas con el control de las fuentes. Las propiedades de los objetos `Font` no pueden modificarse. Si necesita un objeto `Font` distinto, debe crear uno nuevo. Hay muchas versiones sobrecargadas del constructor de `Font` para inicializar objetos `Font`. La figura 17.8 muestra un resumen de algunas propiedades de la clase `Font`.

Observe que la propiedad **Size** devuelve el tamaño de la fuente, según su medida en unidades de diseño, mientras que **SizeInPoints** devuelve el tamaño de la fuente según su medida en puntos (la medida más común). Las *unidades de diseño* permiten especificar el tamaño de la fuente en una de varias unidades de medición, como pulgadas o milímetros. Algunas versiones del constructor de `Font` aceptan un argumento **GraphicsUnit**. Ésta enumeración nos permite especificar la unidad de medición que describe el tamaño de la fuente. Los miembros de la enumeración `GraphicsUnit` son **Point** (1/72 pulgada), **Display** (1/75 pulgada), **Document** 1/300 pulgada), **Millimeter**, **Inch** y **Pixel**. Si se proporciona este argumento, la propiedad `Size` contiene el tamaño de la fuente según lo que mide en la unidad de diseño especificada, y la propiedad `SizeInPoints` contiene el tamaño correspondiente de la fuente en puntos. Por ejemplo, si creamos un objeto `Font` que tenga un tamaño de 1 y especificamos la unidad de medición como `GraphicsUnit.Inch`, la propiedad `Size` será 1 y la propiedad `SizeInPoints` será 72, ya que hay 72 puntos en una pulgada. La medida predeterminada para el tamaño de fuente es `GraphicsUnit.Point` (así, las propiedades `Size` y `SizeInPoints` serán iguales).

La clase `Font` tiene varios constructores. La mayoría de ellos requieren un nombre de fuente, que es un objeto `string` que representa una fuente soportada por el sistema. Algunas de las fuentes comunes son *Microsoft Sans-Serif* y *Serif*. La mayoría de los constructores de `Font` también requieren como argumentos el *tamaño de la fuente* y el *estilo de la fuente*. El estilo de la fuente se especifica con una constante de la enumeración **FontStyle** (**Bold**, **Italic**, **Regular**, **Strikeout** y **Underline**, o una combinación de éstos). Puede combinar cuatro estilos con el operador `|`, como en `FontStyle.Italic | FontStyle.Bold`, lo cual hace que la fuente esté en cursiva y en negrita. El método **DrawString** de `Graphics` establece la fuente de dibujo actual (la fuente en la que se mostrará el texto) a su argumento `Font`.



Error común de programación 17.1

Especificar una fuente que no está disponible en un sistema es un error lógico. Si esto ocurre, C# la sustituirá por la fuente predeterminada del sistema.

Propiedad	Descripción
Bold	Devuelve <code>true</code> si la fuente está en negrita.
FontFamily	Devuelve la familia de donde proviene el objeto <code>Font</code> ; es una estructura de agrupamiento para organizar fuentes y definir sus propiedades similares.
Height	Devuelve la altura de la fuente.
Italic	Devuelve <code>true</code> si la fuente está en cursiva.
Name	Devuelve el nombre de la fuente como un objeto <code>string</code> .
Size	Devuelve un valor <code>float</code> que indica el tamaño actual de la fuente, medido en unidades de diseño (cualquier unidad especificada de medición para la fuente).
SizeInPoints	Devuelve un valor <code>float</code> que indica el tamaño actual de la fuente en puntos.
Strikeout	Devuelve <code>true</code> si la fuente está en formato tachado.
Underline	Devuelve <code>true</code> si la fuente está subrayada.

Figura 17.8 | Propiedades de sólo lectura de la clase `Font`.

Dibujo de cadenas en distintas fuentes

El programa en la figura 17.9 muestra texto en distintas fuentes y varios tamaños. Utiliza el constructor de `Font` para inicializar los objetos `Font` (líneas 24, 28, 32 y 36). Cada llamada al constructor `Font` pasa un nombre de fuente (por ejemplo, Arial, Times New Roman, Courier New o Tahoma) como objeto `string`, un tamaño de fuente (`float`) y un objeto `FontStyle` (estilo). El método `DrawString` de `Graphics` establece la fuente y dibuja el texto en la ubicación especificada. Observe que la línea 20 crea un objeto `SolidBrush` (brocha) color azul oscuro (`DarkBlue`). Todos los objetos `string` que se dibujen con esa brocha aparecerán en color `DarkBlue`.

```

1  // Fig. 17.9 UsoDeFuentesForm.cs
2  // Objetos Font y FontStyle.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // demuestra los constructores y propiedades de las fuentes
8  public partial class UsoDeFuentesForm : Form
9  {
10     // constructor predeterminado
11     public UsoDeFuentesForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // demuestra varias configuraciones de fuentes y estilos
17     protected override void OnPaint( PaintEventArgs paintEvent )
18     {
19         Graphics objetoGraficos = paintEvent.Graphics;
20         SolidBrush brocha = new SolidBrush( Color.DarkBlue );
21
22         // arial, 12 pt negrita
23         FontStyle estilo = FontStyle.Bold;
24         Font arial = new Font( "Arial" , 12, estilo );
25
26         // times new roman, 12 pt normal
27         estilo = FontStyle.Regular;
28         Font timesNewRoman = new Font( "Times New Roman", 12, estilo );
29
30         // courier new, 16 pt negrita y cursiva
31         estilo = FontStyle.Bold | FontStyle.Italic;
32         Font courierNew = new Font( "Courier New", 16, estilo );
33
34         // tahoma, 18 pt subrayado
35         estilo = FontStyle.Strikeout;
36         Font tahoma = new Font( "Tahoma", 18, estilo );
37
38         objetoGraficos.DrawString( arial.Name +
39             " 12 puntos negrita.", arial, brocha, 10, 10 );
40
41         objetoGraficos.DrawString( timesNewRoman.Name +
42             " 12 puntos normal.", timesNewRoman, brocha, 10, 30 );
43
44         objetoGraficos.DrawString( courierNew.Name +
45             " 16 puntos negrita y cursiva.", courierNew,
46             brocha, 10, 54 );
47
48         objetoGraficos.DrawString( tahoma.Name +
49             " 18 puntos tachado.", tahoma, brocha, 10, 75 );

```

Figura 17.9 | Objetos `Font` y `FontStyle`. (Parte I de 2).

```
50     } // fin del método OnPaint
51 } // fin de la clase UsoDeFuentesForm
```

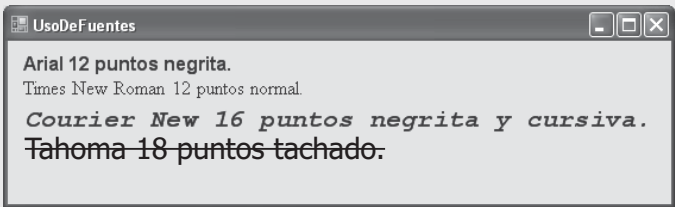


Figura 17.9 | Objetos Font y FontStyle. (Parte 2 de 2).

Métrica de las fuentes

Podemos determinar información precisa acerca de la *métrica* (o propiedades) de una fuente, como la *altura*, el *descendente* (la distancia entre la base de la línea y el punto inferior de la fuente), el *ascendente* (la distancia que se elevan los caracteres por encima de la base de la línea) y el *interlineado* (la diferencia entre el descendente de una línea de texto y el ascendente de la línea de texto que está arriba). La figura 17.10 ilustra estos elementos de la métrica de las fuentes.

La clase FontFamily define las características comunes para un grupo de fuentes relacionadas. Además, proporciona varios métodos que se utilizan para determinar la métrica de las fuentes que se comparte por los miembros de una familia específica. En la figura 17.11 se muestra un resumen de estos métodos.

El programa en la figura 17.12 muestra la métrica de dos fuentes. La línea 23 crea el objeto Font llamado arial y lo establece con una fuente Arial de 12 puntos. La línea 24 usa la propiedad FontFamily de Font para obtener el objeto FontFamily del objeto arial. Las líneas 27-28 muestran en pantalla la representación string de la fuente. Después, las líneas 30-44 utilizan los métodos de la clase FontFamily para obtener el ascendente, el descendente, la altura y el interlineado de la fuente, y para dibujar objetos string que contienen esa información. Las líneas 47-68 repiten este proceso para la fuente sansSerif, un objeto Font derivado de la familia de fuentes MS Sans Serif.

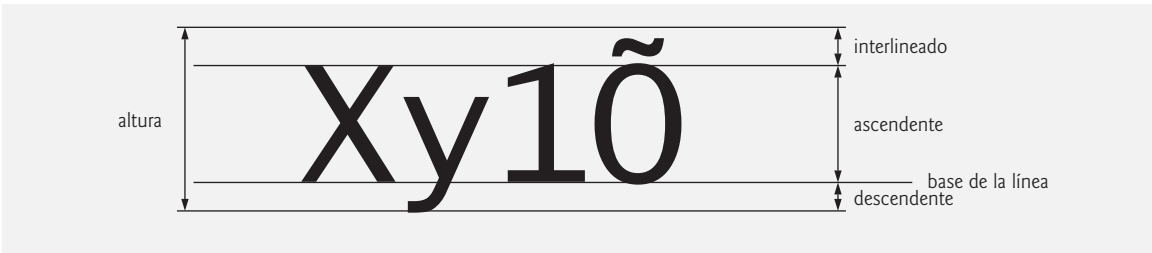


Figura 17.10 | Ilustración de la métrica de las fuentes.

Método	Descripción
GetCellAscent	Devuelve un int que representa el ascendente de una fuente, medido en unidades de diseño.
GetCellDescent	Devuelve un int que representa el descendente de una fuente, medido en unidades de diseño.
GetEmHeight	Devuelve un int que representa la altura de una fuente, medida en unidades de diseño.
GetLineSpacing	Devuelve un int que representa la distancia entre dos líneas consecutivas de texto, medida en unidades de diseño.

Figura 17.11 | Métodos de FontFamily que devuelven información sobre la métrica de las fuentes.

```

1  // Fig 17.12: UsoMetricaFuentesForm.cs
2  // Visualización de la información sobre la métrica de las fuentes
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // muestra información de la fuente
8  public partial class UsoMetricaFuentesForm : Form
9  {
10     // constructor predeterminado
11     public UsoMetricaFuentesForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // muestra información sobre la fuente
17     protected override void OnPaint( PaintEventArgs paintEvent )
18     {
19         Graphics objetoGraficos = paintEvent.Graphics;
20         SolidBrush brocha = new SolidBrush( Color.DarkBlue );
21
22         // métrica de la fuente Arial
23         Font arial = new Font( "Arial", 12 );
24         FontFamily familia = arial.FontFamily;
25
26         // muestra la métrica de la fuente Arial
27         objetoGraficos.DrawString( "Fuente actual: " +
28             arial, arial, brocha, 10, 10 );
29
30         objetoGraficos.DrawString( "Ascendente: " +
31             familia.GetCellAscent( FontStyle.Regular ), arial,
32             brocha, 10, 30 );
33
34         objetoGraficos.DrawString( "Descendente: " +
35             familia.GetCellDescent( FontStyle.Regular ), arial,
36             brocha, 10, 50 );
37
38         objetoGraficos.DrawString( "Altura: " +
39             familia.GetEmHeight( FontStyle.Regular ), arial,
40             brocha, 10, 70 );
41
42         objetoGraficos.DrawString( "Interlineado: " +
43             familia.GetLineSpacing( FontStyle.Regular ), arial,
44             brocha, 10, 90 );
45
46         // muestra la métrica de la fuente Sans Serif
47         Font sanSerif = new Font( "Microsoft Sans Serif",
48             14, FontStyle.Italic );
49         familia = sanSerif.FontFamily;
50
51         objetoGraficos.DrawString( "Fuente actual: " +
52             sanSerif, sanSerif, brocha, 10, 130 );
53
54         objetoGraficos.DrawString( "Ascendente: " +
55             familia.GetCellAscent( FontStyle.Regular ), sanSerif,
56             brocha, 10, 150 );
57
58         objetoGraficos.DrawString( "Descendente: " +
59             familia.GetCellDescent( FontStyle.Regular ), sanSerif,

```

Figura 17.12 | La clase `FontFamily` se utiliza para obtener información sobre la métrica de las fuentes. (Parte 1 de 2).

```

60         brocha, 10, 170 );
61
62         objetoGraficos.DrawString( "Altura: " +
63             familia.GetEmHeight( FontStyle.Regular ), sanSerif,
64             brocha, 10, 190 );
65
66         objetoGraficos.DrawString( "Interlineado: " +
67             familia.GetLineSpacing( FontStyle.Regular ), sanSerif,
68             brocha, 10, 210 );
69     } // fin del método OnPaint
70 } // fin de la clase UsoMetricaFuentesForm

```

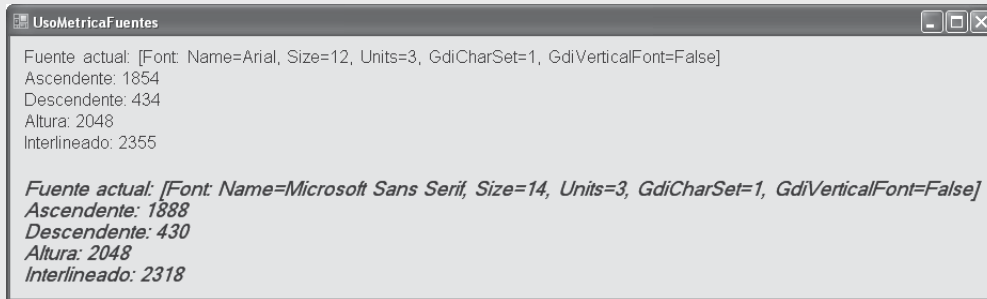


Figura 17.12 | La clase `FontFamily` se utiliza para obtener información sobre la métrica de las fuentes. (Parte 2 de 2).

17.6 Dibujo de líneas, rectángulos y óvalos

En esta sección presentamos los métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. Cada uno de estos métodos de dibujo tiene varias versiones sobrecargadas. Los métodos que dibujan figuras sin relleno por lo general requieren como argumentos un objeto `Pen` y cuatro valores `int`; por otro lado, los que dibujan figuras sólidas, por lo general, requieren como argumentos un objeto `Brush` y cuatro valores `int`. Los primeros dos argumentos `int` representan las coordenadas de la esquina superior izquierda de la figura (o de su área circundante), y los últimos dos argumentos `int` indican la anchura y la altura de la figura (o del área circundante). La figura 17.13 muestra un resumen de varios de los métodos de `Graphics` y sus parámetros. [Nota: muchos de estos métodos están sobrecargados; consulte la documentación para un listado completo (msdn2.microsoft.com/en-us/library/system.drawing.graphics en inglés; [msdn2.microsoft.com/es-es/library/system.drawing.graphics\(VS.80\).aspx](http://msdn2.microsoft.com/es-es/library/system.drawing.graphics(VS.80).aspx) en español).]

La aplicación en la figura 17.14 dibuja líneas, rectángulos y elipses. En esta aplicación también mostramos los métodos que dibujan figuras rellenas y sin relleno.

Los métodos de dibujo de `Graphics` y sus descripciones

`DrawLine( Pen p, int x1, int y1, int x2, int y2 )`

Dibuja una línea de (x1, y1) a (x2, y2). El objeto `Pen` determina el color, estilo y anchura de la línea.

`DrawRectangle( Pen p, int x, int y, int anchura, int altura )`

Dibuja un rectángulo con la anchura y altura especificadas. La esquina superior izquierda del rectángulo está en el punto (x, y). El objeto `Pen` determina el color, estilo y anchura de los bordes del rectángulo.

`FillRectangle( Brush b, int x, int y, int anchura, int altura )`

Dibuja un rectángulo sólido con la anchura y altura especificadas. La esquina superior izquierda del rectángulo está en el punto (x, y). El objeto `Brush` determina el patrón de relleno dentro del rectángulo.

Figura 17.13 | Métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. (Parte 1 de 2).

Los métodos de dibujo de Graphics y sus descripciones

DrawEllipse(Pen p, *int* x, *int* y, *int* anchura, *int* altura)

Dibuja una elipse dentro de un rectángulo delimitador, con la anchura y altura especificadas. La esquina superior izquierda del rectángulo delimitador está en (x, y). El objeto Pen determina el color, estilo y anchura del borde de la elipse.

FillEllipse(Brush b, *int* x, *int* y, *int* anchura, *int* altura)

Dibuja una elipse rellena dentro de un rectángulo delimitador, con la anchura y altura especificadas. La esquina superior izquierda del rectángulo delimitador se encuentra en (x, y). El objeto Brush determina el patrón de relleno dentro de la elipse.

Figura 17.13 | Métodos de Graphics para dibujar líneas, rectángulos y óvalos. (Parte 2 de 2).

```

1  // Fig. 17.14: LineasRectangulosOvalosForm.cs
2  // Demostración de líneas, rectángulos y óvalos.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // dibuja las figuras en el formulario
8  public partial class LineasRectangulosOvalosForm : Form
9  {
10     // constructor predeterminado
11     public LineasRectangulosOvalosForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // redefine el método OnPaint del formulario
17     protected override void OnPaint( PaintEventArgs paintEvent )
18     {
19         // obtiene objeto de gráficos
20         Graphics g = paintEvent.Graphics;
21         SolidBrush brocha = new SolidBrush( Color.Blue );
22         Pen pluma = new Pen( Color.AliceBlue );
23
24         // crea rectángulo relleno
25         g.FillRectangle( brocha, 90, 30, 150, 90 );
26
27         // dibuja líneas para conectar los rectángulos
28         g.DrawLine( pluma, 90, 30, 110, 40 );
29         g.DrawLine( pluma, 90, 120, 110, 130 );
30         g.DrawLine( pluma, 240, 30, 260, 40 );
31         g.DrawLine( pluma, 240, 120, 260, 130 );
32
33         // dibuja rectángulo superior
34         g.DrawRectangle( pluma, 110, 40, 150, 90 );
35
36         // establece color de brocha a rojo
37         brocha.Color = Color.Red;
38
39         // dibuja elipse base
40         g.FillEllipse( brocha, 280, 75, 100, 50 );
41
42         // dibuja líneas conectoras

```

Figura 17.14 | Demostración de los métodos para dibujar líneas, rectángulos y elipses. (Parte 1 de 2).

```

43     g.DrawLine( pluma, 380, 55, 380, 100 );
44     g.DrawLine( pluma, 280, 55, 280, 100 );
45
46     // dibuja el contorno de la elipse
47     g.DrawEllipse( pluma, 280, 30, 100, 50 );
48 } // fin del método OnPaint
49 } // fin de la clase LíneasRectángulosÓvalosForm

```

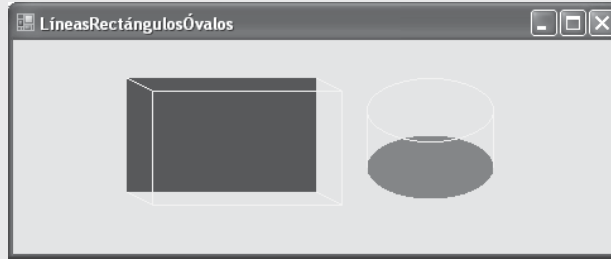


Figura 17.14 | Demostración de los métodos para dibujar líneas, rectángulos y elipses. (Parte 2 de 2).

Los métodos **FillRectangle** y **DrawRectangle** (líneas 25 y 34) dibujan rectángulos en la pantalla. Para cada método, el primer argumento especifica el objeto de dibujo a utilizar. El método **FillRectangle** utiliza un objeto **Brush** (en este caso, una instancia de **SolidBrush**, una clase que se deriva de **Brush**), mientras que el método **DrawRectangle** utiliza un objeto **Pen**. Los siguientes dos argumentos especifican las coordenadas de la esquina superior izquierda del *rectángulo delimitador*, que representa el área en la que se dibujará el rectángulo. Los argumentos cuarto y quinto especifican la anchura y la altura del rectángulo. El método **DrawLine** (líneas 28-31) recibe un objeto **Pen** y dos pares de valores **int**, que especifican el inicio y el final de una línea. Después, el método dibuja una línea, usando el objeto **Pen**.

Los métodos **FillEllipse** y **DrawEllipse** (líneas 40 y 47) proporcionan cada uno versiones sobrecargadas que reciben cinco argumentos. En ambos métodos, el primer argumento especifica el objeto de dibujo a utilizar. Los siguientes dos argumentos especifican las coordenadas de la esquina superior izquierda del rectángulo delimitador que representa el área en la que se dibujará la elipse. Los últimos dos argumentos especifican la anchura y la altura del rectángulo, respectivamente. La figura 17.15 ilustra una elipse delimitada por un rectángulo. La elipse toca el punto medio de cada uno de los cuatro lados del rectángulo delimitador. Este rectángulo delimitador no se muestra en la pantalla.

17.7 Dibujo de arcos

Los arcos son porciones de elipses y se miden en grados; empiezan en un *ángulo inicial* y continúan hasta un número especificado de grados, conocidos como *ángulo del arco*. Se dice que un arco *extiende* (recorre) su ángulo,

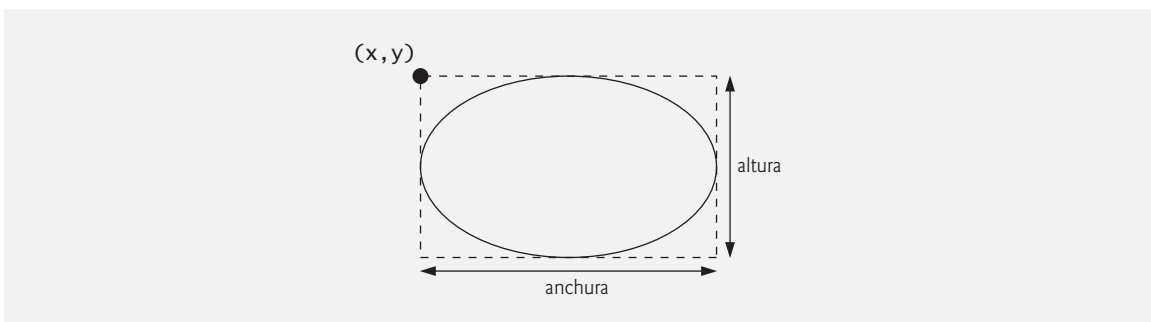


Figura 17.15 | Elipse delimitada por un rectángulo.

empezando desde su ángulo inicial. Los arcos que se extienden en dirección a favor de las manecillas del reloj se miden en grados positivos, mientras que los arcos que se extienden en dirección en contra de las manecillas del reloj se miden en grados negativos. La figura 17.16 ilustra dos arcos. Observe que el arco a la izquierda de la figura se extiende hacia arriba, desde cero grados hasta un valor aproximado de -110 grados. De manera similar, el arco a la derecha de la figura se extiende hacia abajo, desde cero grados hasta un valor aproximado de 110 grados.

Observe los cuadros punteados alrededor de los arcos en la figura 17.16. Cada arco se dibuja como parte de un óvalo (el resto del cual no puede verse). Cuando dibujamos un óvalo, especificamos sus medidas en forma de un rectángulo delimitador que encierra al óvalo. Los cuadros en la figura 17.16 corresponden a esos rectángulos delimitadores. En la figura 17.17 se muestra un resumen de los métodos de **Graphics** que se utilizan para dibujar arcos: **DrawArc**, **DrawPie** y **FillPie**.

El programa en la figura 17.18 dibuja seis imágenes (tres arcos y tres rebanadas de pastel rellenas) para demostrar el uso de los métodos para arcos que se listan en la figura 17.17. Para ilustrar los rectángulos delimitadores que determinan los tamaños y las ubicaciones de los arcos, éstos se muestran dentro de rectángulos rojos que tienen los mismos argumentos para las coordenadas x - y , la anchura y la altura que los definidos por los rectángulos delimitadores para los arcos.

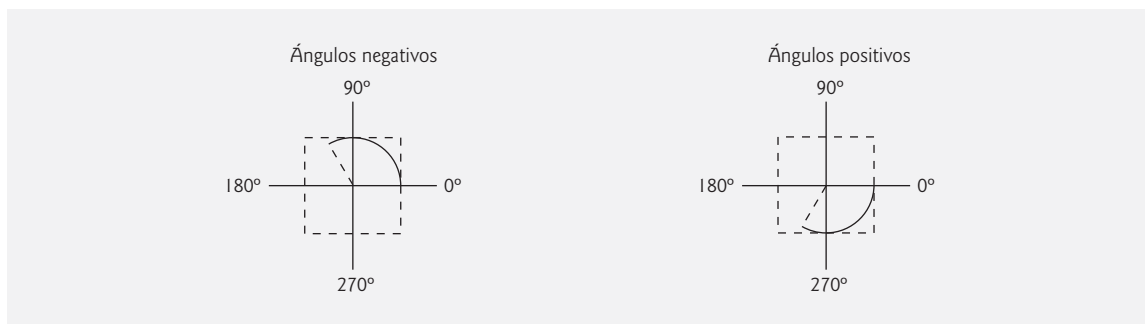


Figura 17.16 | Ángulos positivos y negativos de un arco.

Métodos de **Graphics** y sus descripciones

Nota: Muchos de estos métodos están sobrecargados; consulte la documentación para un listado completo.

DrawArc(Pen p, **int** x, **int** y, **int** anchura, **int** altura, **int** anguloInicial, **int** anguloExtension)

Dibuja un arco que empieza desde el ángulo **anguloInicial** (en grados) y se extiende hasta **anguloExtension** grados. La elipse se define mediante un rectángulo delimitador cuyas medidas se determinan con los valores de **altura**, **anchura** y la esquina superior derecha (x, y). El objeto **Pen** determina el color, la anchura del borde y el estilo del arco.

DrawPie(Pen p, **int** x, **int** y, **int** anchura, **int** altura, **int** anguloInicial, **int** anguloExtension)

Dibuja una sección de pastel de una elipse, que empieza desde el ángulo **anguloInicial** (en grados) y se extiende hasta **anguloExtension** grados. La elipse se define mediante un rectángulo delimitador cuyas medidas se determinan con los valores de **altura**, **anchura** y la esquina superior derecha (x, y). El objeto **Pen** determina el color, la anchura del borde y el estilo del arco.

FillPie(Brush b, **int** x, **int** y, **int** anchura, **int** altura, **int** anguloInicial, **int** anguloExtension)

Funciona de manera similar a **DrawPie**, sólo que dibuja un arco sólido (es decir, un sector). El objeto **Brush** determina el patrón de relleno para el arco sólido.

Figura 17.17 | Métodos de **Graphics** para dibujar arcos.

```

1  // Fig. 17.18: DibujarArcosForm.cs
2  // Dibujar varios arcos en un formulario.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // dibuja varios arcos
8  public partial class DibujarArcosForm : Form
9  {
10     // constructor predeterminado
11     public DibujarArcosForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // dibuja arcos
17     private void DibujarArcosForm_Paint( object sender, PaintEventArgs e )
18     {
19         // obtiene el objeto de gráficos
20         Graphics objetoGraficos = e.Graphics;
21         Rectangle rectangulo1 = new Rectangle( 15, 35, 80, 80 );
22         SolidBrush brocha1 = new SolidBrush( Color.Firebrick );
23         Pen pluma1 = new Pen( brocha1, 1 );
24         SolidBrush brocha2 = new SolidBrush( Color.DarkBlue );
25         Pen pluma2 = new Pen( brocha2, 1 );
26
27         // inicia en 0 y se extiende 360 grados
28         objetoGraficos.DrawRectangle( pluma1, rectangulo1 );
29         objetoGraficos.DrawArc( pluma2, rectangulo1, 0, 360 );
30
31         // inicia en 0 y se extiende 110 grados
32         rectangulo1.Location = new Point( 100, 35 );
33         objetoGraficos.DrawRectangle( pluma1, rectangulo1 );
34         objetoGraficos.DrawArc( pluma2, rectangulo1, 0, 110 );
35
36         // inicia en 0 y se extiende -270 grados
37         rectangulo1.Location = new Point( 185, 35 );
38         objetoGraficos.DrawRectangle( pluma1, rectangulo1 );
39         objetoGraficos.DrawArc( pluma2, rectangulo1, 0, -270 );
40
41         // inicia en 0 y se extiende 360 grados
42         rectangulo1.Location = new Point( 15, 120 );
43         rectangulo1.Size = new Size( 80, 40 );
44         objetoGraficos.DrawRectangle( pluma1, rectangulo1 );
45         objetoGraficos.FillPie( brocha2, rectangulo1, 0, 360 );
46
47         // inicia en 270 y se extiende -90 grados
48         rectangulo1.Location = new Point( 100, 120 );
49         objetoGraficos.DrawRectangle( pluma1, rectangulo1 );
50         objetoGraficos.FillPie( brocha2, rectangulo1, 270, -90 );
51
52         // inicia en 0 y se extiende -270 grados
53         rectangulo1.Location = new Point( 185, 120 );
54         objetoGraficos.DrawRectangle( pluma1, rectangulo1 );
55         objetoGraficos.FillPie( brocha2, rectangulo1, 0, -270 );
56     } // fin del método DibujarArcosForm_Paint
57 } // fin de la clase DibujarArcosForm

```

Figura 17.18 | Dibujo de varios arcos en un formulario. (Parte I de 2).

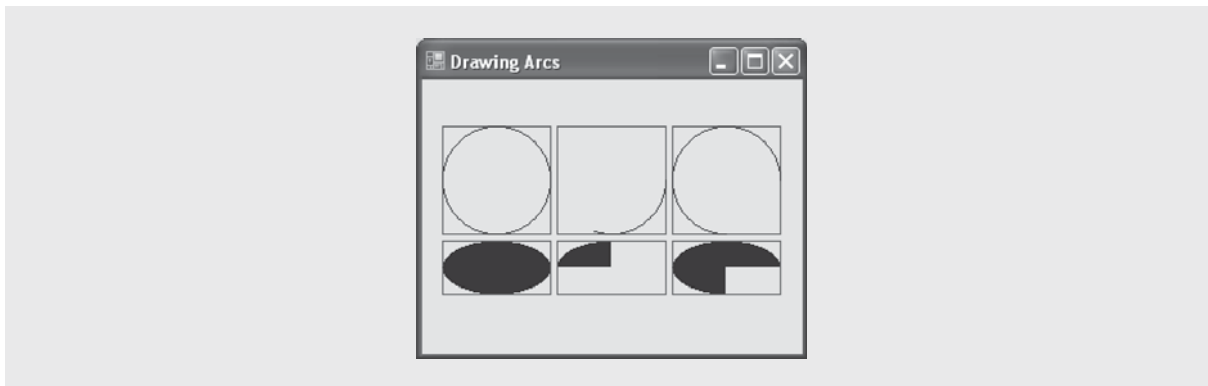


Figura 17.18 | Dibujo de varios arcos en un formulario. (Parte 2 de 2).

Las líneas 20-25 crean los objetos que necesitamos para dibujar varios arcos: un objeto `Graphics`, un objeto `Rectangle`, objetos `SolidBrush` y `Pen`. Después, las líneas 28-29 dibujan un rectángulo y un arco dentro de ese rectángulo. El arco se extiende 360 grados, formando un círculo. La línea 32 cambia la ubicación del rectángulo estableciendo su propiedad `Location` a un nuevo punto. El constructor `Punto` toma argumentos de las coordenadas x y y del nuevo `Punto`. La propiedad `Location` define la esquina superior izquierda del rectángulo. Después de dibujar el rectángulo, el programa dibuja un arco que empieza en 0 grados y se extiende hasta 110 grados. Como los ángulos se incrementan en dirección a favor de las manecillas del reloj, el arco se extiende hacia abajo.

Las líneas 37-39 realizan funciones similares, sólo que el arco especificado se extiende -270 grados. La propiedad `Size` de un objeto `Rectangle` determina la altura y la anchura del arco. La línea 43 extiende la propiedad `Size` a un nuevo objeto `Size`, el cual cambia el tamaño del rectángulo.

El resto de este programa es similar a las porciones antes descritas, sólo que se utiliza un objeto `SolidBrush` con el método `FillPie`. Los arcos resultantes, que están rellenos, pueden verse en la mitad inferior de los resultados de ejemplo (figura 17.18).

17.8 Dibujo de polígonos y polilíneas

Los polígonos son figuras de varios lados. Hay varios métodos de `Graphics` que se utilizan para dibujar polígonos: **`DrawLines`** dibuja una serie de líneas conectadas, **`DrawPolygon`** dibuja un polígono cerrado y **`FillPolygon`** dibuja un polígono sólido. En la figura 17.19 se describen estos métodos. El programa en la figura 17.20 permite a los usuarios dibujar polígonos y líneas conectadas mediante los métodos que se listan en la figura 17.19.

Para permitir al usuario especificar un número variable de puntos, la línea 18 declara el objeto **`ArrayList`** puntos como contenedor para nuestros objetos `Punto`. Un objeto `ArrayList` es similar a un arreglo, sólo que

Método	Descripción
<code>DrawLines</code>	Dibuja una serie de líneas conectadas. Las coordenadas de cada punto se especifican en un arreglo de objetos <code>Point</code> . Si el último punto es distinto del primer punto, la figura no se cierra.
<code>DrawPolygon</code>	Dibuja un polígono. Las coordenadas de cada punto se especifican en un arreglo de objetos <code>Point</code> . Si el último punto es distinto del primero, esos dos puntos se conectan para cerrar el polígono.
<code>FillPolygon</code>	Dibuja un polígono sólido. Las coordenadas de cada punto se especifican en un arreglo de objetos <code>Point</code> . Si el último punto es distinto del primero, esos dos puntos se conectan para cerrar el polígono.

Figura 17.19 | Métodos de `Graphics` para dibujar polígonos.

```

1  // Fig. 17.20: DibujarPoligonos.cs
2  // Demostración de los polígonos.
3  using System;
4  using System.Collections;
5  using System.Drawing;
6  using System.Windows.Forms;
7
8  // demostración de los polígonos
9  public partial class PoligonoForm : Form
10 {
11     // constructor predeterminado
12     public PoligonoForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     // contiene la lista de los vértices del polígono
18     private ArrayList puntos = new ArrayList();
19
20     // inicializa pluma y brocha predeterminadas
21     Pen pluma = new Pen( Color.DarkBlue );
22     SolidBrush brocha = new SolidBrush( Color.DarkBlue );
23
24     // manejador del evento MouseDown del panel de dibujo
25     private void dibujarPanel_MouseDown( object sender, MouseEventArgs e )
26     {
27         // agrega la posición del ratón a la lista de vértices
28         puntos.Add( new Point( e.X, e.Y ) );
29         dibujarPanel.Invalidate(); // actualiza el panel
30     } // fin del método dibujarPanel_MouseDown
31
32     // manejador del evento Paint del panel de dibujo
33     private void dibujarPanel_Paint( object sender, PaintEventArgs e )
34     {
35         // obtiene el objeto de gráficos para el panel
36         Graphics objetoGraficos = e.Graphics;
37
38         // si listaarreglo tiene 2 o más puntos, muestra la figura
39         if ( puntos.Count > 1 )
40         {
41             // obtiene el arreglo para usarlo en las funciones de dibujo
42             Point[] arregloPuntos =
43                 ( Point[] ) puntos.ToArray( puntos[ 0 ].GetType() );
44
45             if ( lineasOption.Checked )
46                 objetoGraficos.DrawLine( pluma, arregloPuntos );
47             else if ( poligonoOption.Checked )
48                 objetoGraficos.DrawPolygon( pluma, arregloPuntos );
49             else if ( poligonoRellenoOption.Checked )
50                 objetoGraficos.FillPolygon( brocha, arregloPuntos );
51         } // fin de if
52     } // fin del método dibujarPanel_Paint
53
54     // maneja el evento Click de borrarButton
55     private void borrarButton_Click( object sender, EventArgs e )
56     {
57         puntos.Clear(); // elimina los puntos
58         dibujarPanel.Invalidate(); // actualiza el panel
59     } // fin del método borrarButton_Click

```

Figura 17.20 | Demostración de cómo dibujar polígonos. (Parte I de 3).

```

60
61 // maneja el evento CheckedException del control RadioButton del polígono
62 private void poligonoOption_CheckedChanged(
63     object sender, System.EventArgs e )
64 {
65     dibujarPanel.Invalidate(); // actualiza el panel
66 } // fin del método poligonoOption_CheckedChanged
67
68 // maneja evento CheckedException del control RadioButton de la línea
69 private void lineaOption_CheckedChanged(
70     object sender, System.EventArgs e )
71 {
72     dibujarPanel.Invalidate(); // actualiza el panel
73 } // fin del método lineaOption_CheckedChanged
74
75 // maneja el evento CheckedException del control RadioButton del polígono relleno
76 private void poligonoRellenoOption_CheckedChanged(
77     object sender, System.EventArgs e )
78 {
79     dibujarPanel.Invalidate(); // actualiza el panel
80 } // fin del método poligonoRellenoOption_CheckedChanged
81
82 // maneja el evento Click de colorButton
83 private void colorButton_Click( object sender, EventArgs e )
84 {
85     // crea nuevo cuadro de diálogo de colores
86     ColorDialog cuadroDialogoColores = new ColorDialog();
87
88     // muestra el cuadro de diálogo y obtiene el resultado
89     DialogResult resultado = cuadroDialogoColores.ShowDialog();
90
91     // regresa si el usuario cancela
92     if ( resultado == DialogResult.Cancel )
93         return;
94
95     pluma.Color = cuadroDialogoColores.Color; // establece la pluma al color
96     brocha.Color = cuadroDialogoColores.Color; // establece la brocha
97     dibujarPanel.Invalidate(); // actualiza el panel;
98 } // fin del método colorButton_Click
99 } // fin de la clase PoligonoForm

```

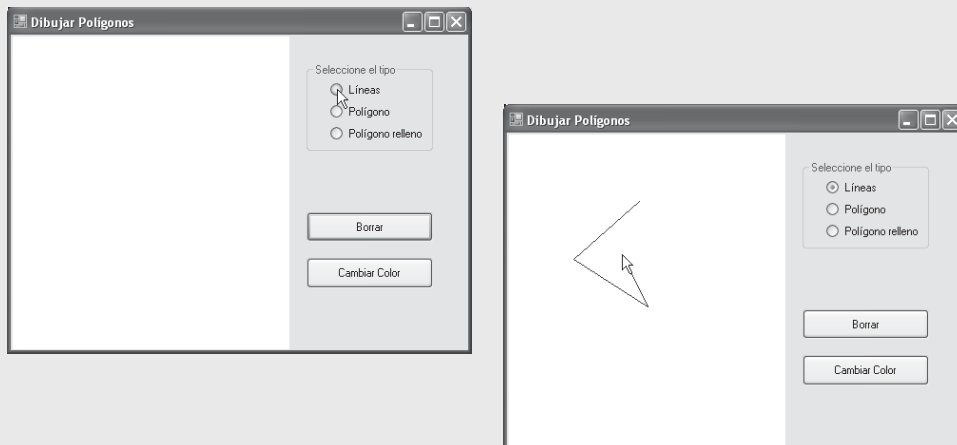


Figura 17.20 | Demostración de cómo dibujar polígonos. (Parte 2 de 3).

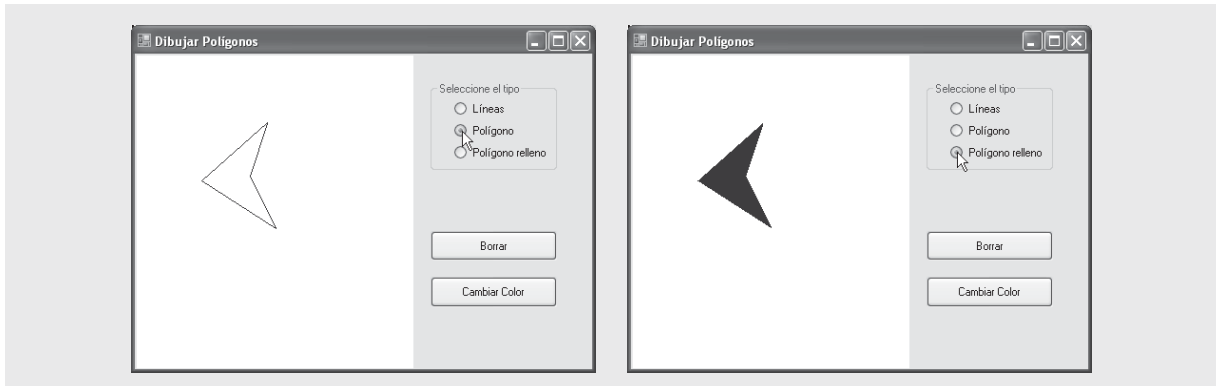


Figura 17.20 | Demostración de cómo dibujar polígonos. (Parte 3 de 3).

puede crecer en forma dinámica para alojar más elementos. Las líneas 21-22 declaran los objetos `Pen` y `Brush` que se utilizan para dar color a nuestras figuras. El manejador del evento `MouseDown` (líneas 25-30) para `dibujarPanel` almacena las ubicaciones (en las que se hizo clic con el ratón) en el objeto `puntos`, mediante el método `Add` de `ArrayList` (línea 28). Después, el manejador de eventos llama al método `Invalidate` de `dibujarPanel` (línea 29) para asegurar que el panel se actualice, de manera que incluya el nuevo punto. El método `dibujarPanel_Paint` (líneas 33-52) maneja el evento `Paint` del objeto `Panel`. Obtiene el objeto `Graphics` del `Panel` (línea 36) y, si el objeto `ArrayList puntos` contiene dos o más objetos `Point` (línea 39), muestra el polígono con el método que seleccionó el usuario mediante los botones de opción de la GUI (líneas 45-50). En las líneas 42-43 extraemos un arreglo del objeto `ArrayList`, mediante el método `ToArray`. Este método puede recibir un solo argumento para determinar el tipo del arreglo devuelto; obtenemos el tipo del primer elemento en el objeto `ArrayList`, llamando al método `GetType` del elemento.

El método `borrarButton_Click` (líneas 55-59) maneja el evento `Click` del botón `Borrar` mediante una llamada al método `Clear` de `ArrayList` (la lista anterior se borra) y la actualización de la pantalla. Las líneas 62-80 definen los manejadores de eventos para el evento `CheckedChanged` de cada botón de opción. Cada método invalida a `dibujarPanel` para asegurar que se vuelva a pintar el panel y se refleje el tipo de figura seleccionado. El manejador de eventos `colorButton_Click` (83-98) permite al usuario seleccionar un nuevo color de dibujo mediante un cuadro de diálogo `ColorDialog`, usando las técnicas demostradas en la figura 17.7.

17.9 Herramientas de gráficos avanzadas

C# ofrece muchas herramientas de gráficos adicionales. Por ejemplo, la jerarquía `Brush` también incluye a *HatchBrush*, *LinearGradientBrush*, *PathGradientBrush* y *TextureBrush*.

Degradados, estilos de líneas y patrones de relleno

El programa de la figura 17.21 muestra diversas características de gráficos, como las líneas punteadas, las líneas gruesas y la habilidad de rellenar figuras con varios patrones. Éstas representan sólo unas cuantas de las herramientas adicionales del espacio de nombres `System.Drawing`.

Las líneas 18-88 definen el manejador de eventos `Paint` de `DibujarFigurasForm`. Las líneas 25-27 crean un objeto de la clase `LinearGradientBrush` llamado `linealBrush`. Un objeto `LinearGradientBrush` (espacio de nombres `System.Drawing.Drawing2D`) permite a los usuarios dibujar con un color degradado. El objeto `LinearGradientBrush` que se utiliza en este ejemplo recibe cuatro argumentos: un objeto `Rectangle`, dos objetos `Color` y un miembro de la enumeración `LinearGradientMode`. En C#, todos los degradados lineales se definen a lo largo de una línea que determina los puntos finales del degradado. Esta línea puede especificarse ya sea por los puntos inicial y final, o por la diagonal de un rectángulo. El primer argumento, `Rectangle areaDibujo1`, representa los puntos finales del degradado lineal; la esquina superior izquierda es el punto inicial y la esquina inferior derecha es el punto final. Los argumentos segundo y tercero especifican los colores que utilizará el degradado. En este caso, el color de la elipse cambiará en forma gradual, de `Color.Blue` a `Color.Yellow`. El último argumento, un tipo de la enumeración `LinearGradientMode`, especifica la dirección del degradado lineal. En nuestro caso, usamos *Linear-*

```

1  // Fig. 17.21: DibujarFigurasForm.cs
2  // Dibujar varias figuras en un formulario.
3  using System;
4  using System.Drawing;
5  using System.Drawing.Drawing2D;
6  using System.Windows.Forms;
7
8  // dibuja figuras con distintas brochas
9  public partial class DibujarFigurasForm : Form
10 {
11     // constructor predeterminado
12     public DibujarFigurasForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     // dibuja varias figuras en el formulario
18     private void DibujarFigurasForm_Paint( object sender, PaintEventArgs e )
19     {
20         // referencias al objeto que vamos a utilizar
21         Graphics objetoGraficos = e.Graphics;
22
23         // brocha de elipse, rectángulo y degradado
24         Rectangle areaDibujo1 = new Rectangle( 5, 35, 30, 100 );
25         LinearGradientBrush linealBrush =
26             new LinearGradientBrush( areaDibujo1, Color.Blue,
27                                     Color.Yellow, LinearGradientMode.ForwardDiagonal );
28
29         // dibuja elipse rellena con un degradado azul-amarillo
30         objetoGraficos.FillEllipse( linealBrush, 5, 30, 65, 100 );
31
32         // pluma y ubicación para el rectángulo de contorno rojo
33         Pen rojoGruesaPen = new Pen( Color.Red, 10 );
34         Rectangle areaDibujo2 = new Rectangle( 80, 30, 65, 100 );
35
36         // dibuja contorno de rectángulo grueso en rojo
37         objetoGraficos.DrawRectangle( rojoGruesaPen, areaDibujo2 );
38
39         // textura de mapa de bits
40         Bitmap texturaBitmap = new Bitmap( 10, 10 );
41
42         // obtiene gráficos de mapa de bits
43         Graphics objetoGraficos2 =
44             Graphics.FromImage( texturaBitmap );
45
46         // brocha y pluma que se utilizan en el programa
47         SolidBrush colorSolidoBrush =
48             new SolidBrush( Color.Red );
49         Pen deColorPen = new Pen( colorSolidoBrush );
50
51         // rellena texturaBitmap con amarillo
52         colorSolidoBrush.Color = Color.Yellow;
53         objetoGraficos2.FillRectangle( colorSolidoBrush, 0, 0, 10, 10 );
54
55         // dibuja un pequeño rectángulo negro en texturaBitmap
56         deColorPen.Color = Color.Black;
57         objetoGraficos2.DrawRectangle( deColorPen, 1, 1, 6, 6 );
58
59         // dibuja un pequeño rectángulo azul en texturaBitmap

```

Figura 17.21 | Varias figuras dibujadas en un formulario. (Parte I de 2).

```

60 colorSolidoBrush.Color = Color.Blue;
61 objetoGraficos2.FillRectangle( colorSolidoBrush, 1, 1, 3, 3 );
62
63 // dibuja un pequeño cuadrado rojo en
64 colorSolidoBrush.Color = Color.Red;
65 objetoGraficos2.FillRectangle( colorSolidoBrush, 4, 4, 3, 3 );
66
67 // crea brocha texturizadaBrush y
68 // muestra rectángulo texturizado
69 TextureBrush texturizadaBrush =
70     new TextureBrush( texturaBitmap );
71 objetoGraficos.FillRectangle( texturizadaBrush, 155, 30, 75, 100 );
72
73 // dibuja arco en forma de pastel, en color blanco
74 deColorPen.Color = Color.White;
75 deColorPen.Width = 6;
76 objetoGraficos.DrawPie( deColorPen, 240, 30, 75, 100, 0, 270 );
77
78 // dibuja líneas en verde y amarillo
79 deColorPen.Color = Color.Green;
80 deColorPen.Width = 5;
81 objetoGraficos.DrawLine( deColorPen, 395, 30, 320, 150 );
82
83 // dibuja una línea amarilla redondeada y punteada
84 deColorPen.Color = Color.Yellow;
85 deColorPen.DashCap = DashCap.Round;
86 deColorPen.DashStyle = DashStyle.Dash;
87 objetoGraficos.DrawLine( deColorPen, 320, 30, 395, 150 );
88 } // fin del método DibujarFigurasForm_Paint
89 } // fin de la clase DibujarFigurasForm

```

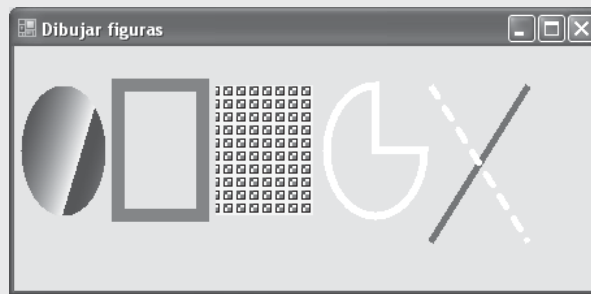


Figura 17.21 | Varias figuras dibujadas en un formulario. (Parte 2 de 2).

GradientMode.ForwardDiagonal, que crea un degradado desde la esquina superior izquierda, hasta la esquina inferior derecha. Después usamos el método `FillEllipse` de `Graphics` en la línea 30 para dibujar una elipse con `linealBrush`; el color cambia en forma gradual de azul a amarillo, como se describió antes.

En la línea 33 creamos el objeto `rojoGruesaPen`. Pasamos los argumentos `Color.Red` e `int 10` al constructor de `rojoGruesaPen`, indicando que queremos que `rojoGruesaPen` dibuje líneas rojas con un grosor de 10 píxeles.

La línea 40 crea una nueva imagen **Bitmap**, que al principio está vacía. La clase `Bitmap` puede producir imágenes a color y en escala de grises; este objeto `Bitmap` específico tiene una anchura de 10 píxeles y una altura de 10 píxeles. El método **FromImage** (líneas 43-44) es un miembro `static` de la clase `Graphics` y extrae el objeto `Graphics` asociado con un objeto `Image`, el cual puede usarse para dibujar en una imagen. Las líneas 52-65 dibujan en el objeto `Bitmap` un patrón que consiste en rectángulos y líneas color negro, azul, rojo y amarillo. Un objeto `TextureBrush` es una brocha que rellena el interior de una figura con una imagen, en vez de hacerlo con un color sólido. En la línea 71, el objeto `TextureBrush` llamado `texturaBrush` rellena un rectángulo con nuestro objeto `Bitmap`. El constructor de `TextureBrush` que se utiliza en las líneas 69-70 recibe como argumento una imagen que define su textura.

A continuación dibujamos un arco en forma de pastel, con una línea gruesa de color blanco. Las líneas 74-75 establecen el objeto `deColorPen` al color blanco (`White`) y modifican su anchura para que sea de seis píxeles. Después dibujamos el pastel en el formulario, especificando el objeto `Pen`, las coordenadas x - y , la anchura y la altura del rectángulo delimitador, y los ángulos inicial y de extensión.

Las líneas 79-81 dibujan una línea color verde con cinco píxeles de anchura. Por último, las líneas 85-86 usan las enumeraciones ***DashCap*** y ***DashStyle*** (espacio de nombres `System.Drawing.Drawing2D`) para especificar las opciones de una línea punteada. La línea 85 establece la propiedad ***DashCap*** de `deColorPen` (no debe confundirse con la enumeración `DashCap`) a un miembro de la enumeración `DashCap`. Esta enumeración especifica los estilos para el inicio y el final de una línea punteada. En este caso, queremos que ambos extremos de la línea punteada estén redondeados, por lo que usamos ***DashCap.Round***. La línea 86 establece la propiedad ***DashStyle*** de `deColorPen` (no debe confundirse con la enumeración `DashStyle`) a ***DashStyle.Dash***, indicando que queremos que nuestra línea consista solamente de guiones cortos.

Rutas generales

Nuestro siguiente ejemplo demuestra el uso de una *ruta general*. Una ruta general es una figura que se construye a partir de líneas rectas y rutas complejas. Un objeto de la clase ***GraphicsPath*** (espacio de nombres `System.Drawing.Drawing2D`) representa a una ruta general. La clase `GraphicsPath` proporciona la funcionalidad que permite la creación de figuras complejas a partir de objetos de gráficos primitivos, basados en vectores. Un objeto `GraphicsPath` consiste de figuras definidas por figuras simples. El punto inicial de cada objeto de gráficos vectoriales (como una línea o arco) que se agrega a la ruta, se conecta mediante una línea recta al punto final del objeto anterior. Cuando se llama, el método ***CloseFigure*** adjunta el punto final del objeto de gráficos vectoriales final al primer punto inicial para la figura actual mediante una línea recta, y después empieza una nueva figura. El método ***StartFigure*** empieza una nueva figura dentro de la ruta, sin cerrar la figura anterior.

El programa de la figura 17.22 dibuja rutas generales en la forma de estrellas de cinco puntas. Las líneas 26-29 definen dos arreglos `int` que representan las coordenadas x y y de los puntos en la estrella, y la línea 32 define el objeto `GraphicsPath` llamado `estrella`. Después, un ciclo (líneas 35-37) crea líneas para conectar los puntos de la estrella y agrega estas líneas a `estrella`. Utilizamos el método ***AddLine*** de `GraphicsPath` para adjuntar una línea a la figura. Los argumentos de ***AddLine*** especifican las coordenadas para los puntos finales de la línea; cada nueva llamada a ***AddLine*** agrega una línea del punto anterior al punto actual. La línea 40 utiliza el método ***CloseFigure*** de `GraphicsPath` para completar la figura.

La línea 43 establece el origen del objeto `Graphics`. Los argumentos para el método ***TranslateTransform*** indican que el origen debe traducirse a las coordenadas (150, 150). El ciclo en las líneas 46-55 dibuja la estrella 18 veces, girándola alrededor del origen. La línea 48 utiliza el método ***RotateTransform*** de `Graphics` para avanzar

```

1  // Fig. 17.22: DibujarEstrellasForm.cs
2  // Uso de rutas para dibujar estrellas en el formulario.
3  using System;
4  using System.Drawing;
5  using System.Drawing.Drawing2D;
6  using System.Windows.Forms;
7
8  // dibuja estrellas de colores al azar
9  public partial class DibujarEstrellasForm : Form
10 {
11     // constructor predeterminado
12     public DibujarEstrellasForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     // crea la ruta y dibuja estrellas a lo largo de ésta
18     private void DibujarEstrellasForm_Paint(object sender, PaintEventArgs e)
19     {

```

Figura 17.22 | Rutas utilizadas para dibujar estrellas en un formulario. (Parte I de 2).

```

20 Graphics objetoGraficos = e.Graphics;
21 Random aleatorio = new Random();
22 SolidBrush brocha =
23     new SolidBrush( Color.DarkMagenta );
24
25 // puntos x y y de la ruta
26 int[] xPoints =
27     { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
28 int[] yPoints =
29     { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
30
31 // crea ruta de gráficos para la estrella;
32 GraphicsPath estrella = new GraphicsPath();
33
34 // crea una estrella a partir de una serie de puntos
35 for ( int i = 0; i <= 8; i += 2 )
36     estrella.AddLine( xPoints[ i ], yPoints[ i ],
37                     xPoints[ i + 1 ], yPoints[ i + 1 ] );
38
39 // cierra la figura
40 estrella.CloseFigure();
41
42 // traduce el origen a (150, 150)
43 objetoGraficos.TranslateTransform( 150, 150 );
44
45 // gira el origen y dibuja estrellas en colores al azar
46 for ( int i = 1; i <= 18; i++ )
47 {
48     objetoGraficos.RotateTransform( 20 );
49
50     brocha.Color = Color.FromArgb(
51         aleatorio.Next( 200, 255 ), aleatorio.Next( 255 ),
52         aleatorio.Next( 255 ), aleatorio.Next( 255 ) );
53
54     objetoGraficos.FillPath( brocha, estrella );
55 } // fin de for
56 } // fin del método DibujarEstrellasForm_Paint
57 } // fin de la clase DibujarEstrellasForm

```



Figura 17.22 | Rutas utilizadas para dibujar estrellas en un formulario. (Parte 2 de 2).

a la siguiente posición en el formulario; el argumento especifica el ángulo de giro en grados. Después, el método `FillPath` de `Graphics` (línea 54) dibuja una versión rellena de la estrella, con el objeto `Brush` que se creó en las líneas 50-52. La aplicación determina el color del objeto `SolidBrush` al azar, usando el método `Next` de la clase `Random`.

17.10 Introducción a multimedia

C# ofrece muchas formas convenientes de incluir imágenes y animaciones en los programas. Las personas que entraron al campo de la computación décadas atrás usaban las computadoras principalmente para realizar cálculos aritméticos. A medida que la disciplina evoluciona, nos damos cuenta de la importancia de las capacidades de manipulación de datos de las computadoras. Cada vez vemos más aplicaciones tridimensionales nuevas e interesantes. La programación con multimedia es un campo interesante e innovador, pero presenta muchos retos.

Las aplicaciones multimedia demandan un extraordinario poder de cómputo. Los procesadores ultra rápidos de hoy hacen que las aplicaciones basadas en multimedia sean algo común. Con la creciente explosión del mercado para multimedia, los usuarios están comprando los procesadores más rápidos, memorias más extensas y los anchos de banda de comunicaciones más amplios, necesarios para soportar las aplicaciones multimedia. Esto beneficia a las industrias de la computación y de las comunicaciones, las cuales proveen el hardware, el software y los servicios que impulsan a la revolución multimedia.

En el resto de las secciones de este capítulo, presentaremos las características y herramientas básicas de procesamiento de imágenes y demás contenido multimedia. La sección 17.11 habla acerca de cómo cargar, mostrar y escalar imágenes; la sección 17.12 muestra la animación de imágenes; la sección 17.13 presenta las capacidades de video del control `Windows Media Player`; y la sección 17.14 explora la tecnología `Microsoft Agent`.

17.11 Carga, visualización y escalado de imágenes

Las herramientas multimedia de C# incluyen gráficos, imágenes, animaciones y video. Las secciones anteriores demostraron las herramientas de gráficos vectoriales de C#; esta sección presenta la manipulación de imágenes. La aplicación de la figura 17.23 carga un objeto `Image` (espacio de nombres `System.Drawing`) y después permite al usuario escalar ese objeto `Image` a una altura y anchura especificadas.

```

1  // Fig. 17.23: MostrarLogoForm.cs
2  // Mostrar y cambiar el tamaño de una imagen
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // muestra una imagen y permite al usuario cambiar su tamaño
8  public partial class MostrarLogoForm : Form
9  {
10     private Image imagen = Image.FromFile( @"imagenes\Logo.gif" );
11     private Graphics objetoGraficos;
12
13     public MostrarLogoForm()
14     {
15         InitializeComponent();
16         objetoGraficos = this.CreateGraphics();
17     }
18
19     // maneja el evento Click de setButton
20     private void establecerButton_Click( object sender, EventArgs e )
21     {
22         // obtiene la entrada del usuario
23         int anchura = Convert.ToInt32( anchuraTextBox.Text );
24         int altura = Convert.ToInt32( alturaTextBox.Text );

```

Figura 17.23 | Cambiar el tamaño de las imágenes. (Parte 1 de 2).

```

25
26 // si las medidas especificadas son demasiado grandes
27 // muestra mensaje de problema
28 if ( anchura > 375 || altura > 225 )
29 {
30     MessageBox.Show( " Altura o Anchura demasiado grandes" );
31     return;
32 } // fin de if
33
34 // borra el formulario y después dibuja la imagen
35 objetoGraficos.Clear( this.BackColor );
36 objetoGraficos.DrawImage( imagen, 5, 5, anchura, altura );
37 } // fin del método establecerButton_Click
38 } // fin de la clase MostrarLogoForm

```



Figura 17.23 | Cambiar el tamaño de las imágenes. (Parte 2 de 2).

La línea 10 declara a la variable `Image` llamada `imagen` y utiliza el método `static FromFile` de `Image` para cargar una imagen a partir de un archivo en el disco. La línea 16 utiliza el método `CreateGraphics` de `Form` para crear un objeto `Graphics` y dibujarlo en el formulario. El método `CreateGraphics` se hereda de la clase `Control`. Al hacer clic en el botón **Establecer**, las líneas 28-32 validan la anchura y la altura para asegurar que no sean demasiado grandes. Si los parámetros son válidos, la línea 35 llama al método `Clear` de `Graphics` para pintar todo el formulario en el color de fondo actual. La línea 36 llama al método `DrawImage` de `Graphics`, y le pasa como argumentos la imagen a dibujar, la coordenada x de la esquina superior izquierda de la imagen, la coordenada y de la misma esquina, la anchura y la altura de la imagen. Si la anchura y la altura no corresponden a las medidas originales, la imagen se escala para ajustarla a los nuevos valores de anchura y altura.

17.12 Animación de una serie de imágenes

El siguiente ejemplo anima una serie de imágenes almacenadas en un arreglo. La aplicación utiliza la misma técnica para cargar y mostrar objetos `Image` que en la figura 17.23.

La animación en la figura 17.24 utiliza un control `PictureBox`, el cual contiene las imágenes que animaremos. Utilizamos un control `Timer` para recorrer en forma cíclica todas las imágenes y mostrar una imagen cada 50 milisegundos. La variable `cuenta` lleva la cuenta del número de la imagen actual y se incrementa en uno cada vez que mostramos una nueva imagen. El arreglo incluye 30 imágenes (numeradas del 0 al 29); cuando la aplicación llega a la imagen 29, regresa a la imagen 0. Las 30 imágenes se encuentran en la carpeta `imagenes`, dentro de los directorios `bin/Debug` y `bin/Release` del proyecto.

```

1  // Fig. 17.24: AnimadorLogo.cs
2  // Programa que anima una serie de imágenes.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  // anima una serie de 30 imágenes
8  public partial class AnimadorLogoForm : Form
9  {
10     private Image[] imagenes = new Image[ 30 ];
11     private int cuenta = -1;
12
13     // constructor de LogoAnimator
14     public AnimadorLogoForm()
15     {
16         InitializeComponent();
17
18         for ( int i = 0; i < 30; i++ )
19             imagenes[ i ] = Image.FromFile( @"imagenes\deitel" + i + ".gif" );
20
21         logoPictureBox.Image = imagenes[ 0 ]; // muestra la primera imagen
22
23         // establece el control PictureBox para que sea del mismo tamaño que el objeto Image
24         logoPictureBox.Size = logoPictureBox.Image.Size;
25     } // fin del constructor de LogoAnimator
26
27     // manejador de eventos para el evento Tick del temporizador
28     private void timer_Tick( object sender, EventArgs e )
29     {
30         cuenta = ( cuenta + 1 ) % 30; // incrementa el contador
31         logoPictureBox.Image = imagenes[ cuenta ]; // muestra la siguiente imagen
32     } // fin del método timer_Tick
33 } // fin de la clase AnimadorLogoForm

```



Figura 17.24 | Animación de una serie de imágenes.

Las líneas 18-19 cargan cada una de las 30 imágenes y las colocan en un arreglo de objetos Image. La línea 21 coloca la primera imagen en el control PictureBox. La línea 24 modifica el tamaño del control PictureBox, de manera que sea igual al tamaño del objeto Image que está mostrando. El manejador de eventos para el evento Tick de timer (líneas 28-32) responde a cada evento mostrando la siguiente imagen del arreglo.



Tip de rendimiento 17.2

Es más eficiente cargar los cuadros de una animación como una imagen, que cargar cada imagen por separado. (Puede usarse un programa de dibujo como Adobe Photoshop® o Jasc® PaintShop Pro™ para combinar los cuadros de la animación en una sola imagen.) Si las imágenes se cargan por separado de un sitio Web, cada imagen que se cargue requiere una conexión separada al sitio en el que están almacenadas las imágenes; este proceso puede ocasionar un bajo rendimiento.

Ejemplo de ajedrez

El siguiente ejemplo de ajedrez muestra las técnicas para la *detección de colisiones* bidimensional, seleccionando cuadros individuales de una imagen con varios cuadros, y la *invalidación regional*, actualizando sólo las partes de la pantalla que han cambiado, para incrementar el rendimiento. La detección de colisiones bidimensional permite a un programa detectar si dos figuras se traslapan o si un punto está contenido dentro de una figura. En el siguiente ejemplo, mostraremos la forma más simple de detección de colisiones, la cual determina si un punto (la ubicación del clic del ratón) está contenido dentro de un rectángulo (una imagen de una pieza de ajedrez).

La clase PiezaAjedrez (figura 17.25) representa las piezas de ajedrez individuales. Las líneas 10-18 definen una enumeración pública de constantes que identifican a cada tipo de pieza de ajedrez. Las constantes también sirven para identificar la ubicación de cada pieza en el archivo de imágenes de piezas de ajedrez. El objeto Rectangle llamado rectanguloDestino (líneas 24-25) identifica la ubicación de la imagen en el tablero de ajedrez. Las propiedades x y y del rectángulo se asignan en el constructor de PiezaAjedrez, y todas las imágenes de las piezas de ajedrez tienen una anchura y una altura de 75 píxeles.

El constructor de PiezaAjedrez (líneas 28-39) recibe el tipo de pieza de ajedrez, las coordenadas *x-y* de su ubicación y el objeto Bitmap que contiene todas las imágenes de las piezas de ajedrez. En vez de cargar la imagen de la pieza de ajedrez dentro de la clase, permitimos que la clase que hace la llamada pase la imagen. Esto incrementa la flexibilidad de la clase, al permitir al usuario cambiar las imágenes. Las líneas 36-38 extraen una subimagen que sólo contiene los datos del mapa de bits de la pieza actual. Nuestras imágenes de las piezas de ajedrez se definen de una manera específica: una imagen contiene seis imágenes de piezas de ajedrez, cada una definida dentro de un bloque de 75 píxeles, lo que produce un tamaño total de la imagen de 450 por 75. Para obtener una sola imagen, usamos el método Clone de Bitmap, el cual nos permite especificar la ubicación de una imagen rectangular y el formato de píxeles deseado. La ubicación es un bloque de 75 por 75 píxeles, en donde la coordenada x de su esquina superior izquierda es igual a $75 * \text{tipo}$ y la coordenada y correspondiente es igual a 0. Para el formato de los píxeles, especificamos la constante DontCare, con lo cual el formato queda sin cambios.

```

1  // Fig. 17.25 : PiezaAjedrez.cs
2  // Clase que representa los atributos de una pieza de ajedrez.
3  using System;
4  using System.Drawing;
5
6  // representa a una pieza de ajedrez
7  class PiezaAjedrez
8  {
9      // define las constantes del tipo de pieza de ajedrez
10     public enum Tipos
11     {
12         REY,
13         REINA,
14         ALFIL,
15         CABALLO,

```

Figura 17.25 | Clase que representa los atributos de una pieza de ajedrez. (Parte 1 de 2).

```

16     TORRE,
17     PEON
18 } // fin de la enumeración Tipos
19
20 private int tipoActual; // el tipo de este objeto
21 private Bitmap imagenPieza; // la imagen de este objeto
22
23 // ubicación de visualización predeterminada
24 private Rectangle rectanguloDestino =
25     new Rectangle( 0, 0, 75, 75 );
26
27 // construye la pieza
28 public PiezaAjedrez( int tipo, int ubicacionX,
29     int ubicacionY, Bitmap imagenOrigen )
30 {
31     tipoActual = tipo; // establece el tipo actual
32     rectanguloDestino.X = ubicacionX; // establece la ubicación actual x
33     rectanguloDestino.Y = ubicacionY; // establece la ubicación actual y
34
35     // obtiene imagenPieza de una sección de imagenOrigen
36     imagenPieza = imagenOrigen.Clone(
37         new Rectangle( tipo * 75, 0, 75, 75 ),
38         System.Drawing.Imaging.PixelFormat.DontCare );
39 } // fin del método PiezaAjedrez
40
41 // dibuja la pieza de ajedrez
42 public void Dibujar( Graphics objetoGraficos )
43 {
44     objetoGraficos.DrawImage( imagenPieza, rectanguloDestino );
45 } // fin del método Dibujar
46
47 // obtiene el rectángulo de la ubicación de esta pieza
48 public Rectangle ObtenerLimites()
49 {
50     return rectanguloDestino;
51 } // fin del método ObtenerLimites
52
53 // establece la ubicación de esta pieza
54 public void EstablecerUbicacion( int ubicacionX, int ubicacionY )
55 {
56     rectanguloDestino.X = ubicacionX;
57     rectanguloDestino.Y = ubicacionY;
58 } // fin del método EstablecerUbicacion
59 } // fin de la clase PiezaAjedrez

```

Figura 17.25 | Clase que representa los atributos de una pieza de ajedrez. (Parte 2 de 2).

El método Dibujar (líneas 42-45) hace que PiezaAjedrez dibuje un objeto imagenPieza en el objeto rectanguloDestino, usando el objeto Graphics que se pasa como argumento para Dibujar. El método ObtenerLimites (líneas 48-51) devuelve el objeto rectanguloDestino para usarlo en la detección de colisiones, y el método EstablecerUbicacion (líneas 54-58) permite que la clase que hace la llamada especifique una nueva ubicación para la pieza.

La clase JuegoAjedrezForm (figura 17.26) define el código del juego y de los gráficos para nuestro juego de ajedrez. Las líneas 11-15 definen las variables de instancia requeridas por el programa. El objeto ArrayList llamado mosaicoAjedrez (línea 11) almacena las imágenes de los mosaicos del tablero. El objeto ArrayList llamado piezasAjedrez (línea 12) almacena todos los objetos PiezaAjedrez activos, y el objeto int indiceSelecto (línea 13) identifica el índice en los objetos piezaAjedrez para la pieza actual seleccionada. El tablero (línea 14) es un arreglo int bidimensional de 8 por 8, que corresponde a los cuadros de un tablero de ajedrez. Cada elemento

del tablero es un entero del 0 al 3, que corresponde a un índice en `mosaicoAjedrez` y se utiliza para especificar la imagen del cuadro del tablero de ajedrez. El valor `const TAMANIOMOSAICO` (línea 15) define el tamaño de cada mosaico en píxeles.

```

1  // Fig. 17.26: JuegoAjedrezForm.cs
2  // Código de los gráficos del juego de ajedrez.
3  using System;
4  using System.Collections;
5  using System.Drawing;
6  using System.Windows.Forms;
7
8  // permite jugar ajedrez a 2 jugadores
9  public partial class JuegoAjedrezForm : Form
10 {
11     private ArrayList mosaicoAjedrez = new ArrayList(); // para las imágenes de los
        mosaicos
12     private ArrayList piezasAjedrez = new ArrayList(); // para las piezas de ajedrez
13     private int indiceSelecto = -1; // índice para la pieza seleccionada
14     private int[ , ] tablero = new int[ 8, 8 ]; // arreglo del tablero
15     private const int TAMANIOMOSAICO = 75; // tamaño del mosaico de ajedrez en píxeles
16
17     // constructor predeterminado
18     public JuegoAjedrezForm()
19     {
20         // Requerido para el soporte del diseñador de Windows Forms
21         InitializeComponent();
22     } // fin del constructor
23
24     // carga los mapas de bits de los mosaicos y reinicia el juego
25     private void JuegoAjedrezForm_Load( object sender, EventArgs e )
26     {
27         // carga los mosaicos del tablero de ajedrez
28         mosaicoAjedrez.Add( Bitmap.FromFile( @"imagenes\mosaicoClaro1.png" ) );
29         mosaicoAjedrez.Add( Bitmap.FromFile( @"imagenes\mosaicoClaro2.png" ) );
30         mosaicoAjedrez.Add( Bitmap.FromFile( @"imagenes\mosaicoOscuro1.png" ) );
31         mosaicoAjedrez.Add( Bitmap.FromFile( @"imagenes\mosaicoOscuro2.png" ) );
32
33         ReiniciarTablero(); // inicializa el tablero
34         Invalidate(); // actualiza el formulario
35     } // fin del método JuegoAjedrezForm_Load
36
37     // inicializa las piezas para empezar y reconstruir el tablero
38     private void ReiniciarTablero()
39     {
40         int actual = -1;
41         PiezaAjedrez pieza;
42         Random aleatorio = new Random();
43         bool claro = false;
44         int tipo;
45
46         piezasAjedrez.Clear(); // asegura que la lista de arreglos esté vacía
47
48         // carga imagen de piezas blancas
49         Bitmap piezasBlancas =
50             ( Bitmap ) Image.FromFile( @"imagenes\piezasBlancas.png" );
51
52         // carga imagen de piezas negras

```

Figura 17.26 | Código del juego de ajedrez. (Parte I de 6).


```

53 Bitmap piezasnegras =
54     ( Bitmap ) Image.FromFile( @"imagenes\piezasNegras.png" );
55
56 // establece que se van a dibujar primero las piezas blancas
57 Bitmap seleccion = piezasBlancas;
58
59 // recorre las filas del tablero en el ciclo exterior
60 for ( int fila = 0; fila <= tablero.GetUpperBound( 0 ); fila++ )
61 {
62     // si está en las filas inferiores, establece a imagenes de piezas negras
63     if ( fila > 5 )
64         seleccion = piezasnegras;
65
66     // recorre las columnas del tablero en el ciclo interno
67     for ( int columna = 0;
68         columna <= tablero.GetUpperBound( 1 ); columna++ )
69     {
70         // si es la primera o última fila, organiza las piezas
71         if ( fila == 0 || fila == 7 )
72         {
73             switch ( columna )
74             {
75                 case 0:
76                     // establece la pieza actual a una torre
77                     actual = ( int ) PiezaAjedrez.Tipos.TORRE;
78                     break;
79                 case 1:
80                     // establece pieza actual a caballo
81                     actual = ( int ) PiezaAjedrez.Tipos.CABALLO;
82                     break;
83                 case 2:
84                     // establece pieza actual a alfil
85                     actual = ( int ) PiezaAjedrez.Tipos.ALFIL;
86                     break;
87                 case 3: // establece pieza actual a rey
88                     actual = ( int ) PiezaAjedrez.Tipos.REY;
89                     break;
90                 case 4: // establece pieza actual a reina
91                     actual = ( int ) PiezaAjedrez.Tipos.REINA;
92                     break;
93             } // fin de switch
94
95             // crea la pieza actual en la posición inicial
96             pieza = new PiezaAjedrez( actual,
97                 columna * TAMANIOMOSAICO, fila * TAMANIOMOSAICO, seleccion );
98
99             piezasAjedrez.Add( pieza ); // agrega la pieza a la lista de arreglos
100         } // fin de if
101
102         // si es la segunda o séptima fila, organiza los peones
103         if ( fila == 1 || fila == 6 )
104         {
105             pieza = new PiezaAjedrez(
106                 ( int ) PiezaAjedrez.Tipos.PEON,
107                 columna * TAMANIOMOSAICO, fila * TAMANIOMOSAICO, seleccion );
108             piezasAjedrez.Add( pieza ); // agrega la pieza a la lista de arreglos
109         } // fin de if
110

```

Figura 17.26 | Código del juego de ajedrez. (Parte 2 de 6).

```

111         tipo = aleatorio.Next( 0, 2 ); // determina el tipo de la pieza de ajedrez
112
113         if ( claro ) // establece mosaico claro
114         {
115             tablero[ fila, columna ] = tipo;
116             claro = false;
117         }
118         else // establece mosaico oscuro
119         {
120             tablero[ fila, columna ] = tipo + 2;
121             claro = true;
122         }
123     } // fin del ciclo for para las columnas
124
125     claro = !claro; // considera cambio de color para mosaico de nueva fila
126 } // fin del ciclo for para las filas
127 } // fin del método ReiniciarTablero
128
129 // muestra el tablero en el evento OnPaint del formulario
130 private void JuegoAjedrezForm_Paint( object sender, PaintEventArgs e )
131 {
132     Graphics objetoGraficos = e.Graphics; // obtiene el objeto de gráficos
133     objetoGraficos.TranslateTransform( 0, 24 ); // ajusta el origen
134
135     for ( int fila = 0; fila <= tablero.GetUpperBound( 0 ); fila++ )
136     {
137         for ( int columna = 0;
138             columna <= tablero.GetUpperBound( 1 ); columna++ )
139         {
140             // dibuja la imagen especificada en el arreglo del tablero
141             objetoGraficos.DrawImage(
142                 ( Image ) mosaicoAjedrez[ tablero[ fila, columna ] ],
143                 new Point( TAMANIOMOSAICO * columna, ( TAMANIOMOSAICO * fila ) ) );
144         } // fin del ciclo for para las columnas
145     } // fin del ciclo for para las filas
146 } // fin del método JuegoAjedrezForm_Paint
147
148 // devuelve índice de pieza que interseca el punto
149 // excluye un valor en forma opcional
150 private int ComprobarLimites( Point punto, int excluir )
151 {
152     Rectangle rectangulo; // rectangulo delimitador actual
153
154     for ( int i = 0; i < piezasAjedrez.Count; i++ )
155     {
156         // obtiene el rectángulo de la pieza
157         rectangulo = ObtenerPieza( i ).ObtenerLimites();
158
159         // comprueba si el rectangulo contiene el punto
160         if ( rectangulo.Contains( punto ) && i != excluir )
161             return i;
162     } // fin de for
163
164     return -1;
165 } // fin del método ComprobarLimites
166
167 // maneja el evento Paint de piezaBox
168 private void piezaBox_Paint(

```

Figura 17.26 | Código del juego de ajedrez. (Parte 3 de 6).

```

169     object sender, System.Windows.Forms.PaintEventArgs e )
170 {
171     // dibuja todas las piezas
172     for ( int i = 0; i < piezasAjedrez.Count; i++ )
173         ObtenerPieza( i ).Dibujar( e.Graphics );
174 } // fin del método piezaBox_Paint
175
176 // maneja eventoMouseDown de piezaBox
177 private void piezaBox_MouseDown(
178     object sender, System.Windows.Forms.MouseEventArgs e )
179 {
180     // determina la pieza seleccionada
181     indiceSelecto = ComprobarLimites( new Point( e.X, e.Y ), -1 );
182 } // fin del método piezaBox_MouseDown
183
184 // si se seleccionó la pieza moverla
185 private void piezaBox_MouseMove(
186     object sender, System.Windows.Forms.MouseEventArgs e )
187 {
188     if ( indiceSelecto > -1 )
189     {
190         Rectangle region = new Rectangle(
191             e.X - TAMANIOMOSAICO * 2, e.Y - TAMANIOMOSAICO * 2,
192             TAMANIOMOSAICO * 4, TAMANIOMOSAICO * 4 );
193
194         // establece el centro de la pieza en base al ratón
195         ObtenerPieza( indiceSelecto ).EstablecerUbicacion(
196             e.X - TAMANIOMOSAICO / 2, e.Y - TAMANIOMOSAICO / 2 );
197
198         piezaBox.Invalidate( region ); // actualiza la region
199     } // fin de if
200 } // fin del método piezaBox_MouseMove
201
202 // si se suelta el botón del ratón, se deselectiona la pieza y se deja en su lugar
203 private void piezaBox_MouseUp( object sender, MouseEventArgs e )
204 {
205     int eliminar = -1;
206
207     // si se seleccionó la pieza de ajedrez
208     if ( indiceSelecto > -1 )
209     {
210         Point actual = new Point( e.X, e.Y );
211         Point nuevoPunto = new Point(
212             actual.X - ( actual.X % TAMANIOMOSAICO ),
213             actual.Y - ( actual.Y % TAMANIOMOSAICO ) );
214
215         // comprueba los límites con el punto, excluye la pieza seleccionada
216         eliminar = ComprobarLimites( actual, indiceSelecto );
217
218         // inserta pieza en el centro del cuadro más cercano
219         ObtenerPieza( indiceSelecto ).EstablecerUbicacion( nuevoPunto.X, nuevoPunto.Y );
220         indiceSelecto = -1; // deselectiona la pieza
221
222         // elimina la pieza que se tomó
223         if ( eliminar > -1 )
224             piezasAjedrez.RemoveAt( eliminar );
225     } // fin de if
226

```

Figura 17.26 | Código del juego de ajedrez. (Parte 4 de 6).

```

227     piezaBox.Invalidate(); // asegura que se elimine el artefacto
228 } // fin del método piezaBox_MouseUp
229
230 // función auxiliar para convertir
231 // el objeto ArrayList a PiezaAjedrez
232 private PiezaAjedrez ObtenerPieza( int i )
233 {
234     return ( PiezaAjedrez ) piezasAjedrez[ i ];
235 } // fin del método ObtenerPieza
236
237 // maneja clic en el menú de opción NuevoJuego
238 private void nuevoJuegoItem_Click(
239     object sender, System.EventArgs e )
240 {
241     ReiniciarTablero(); // reinicializa el tablero
242     Invalidate(); // actualiza el formulario
243 } // fin del método nuevoJuegoItem_Click
244 } // fin de la clase JuegoAjedrezForm

```

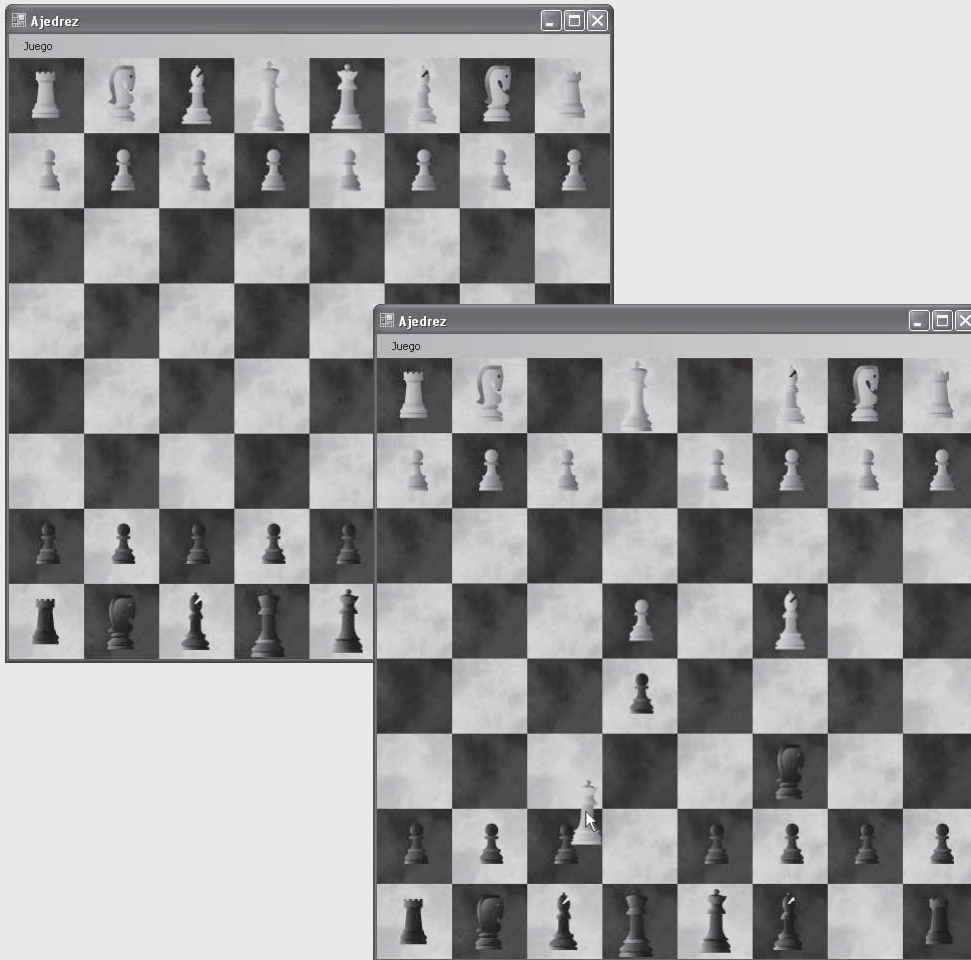


Figura 17.26 | Código del juego de ajedrez. (Parte 5 de 6).

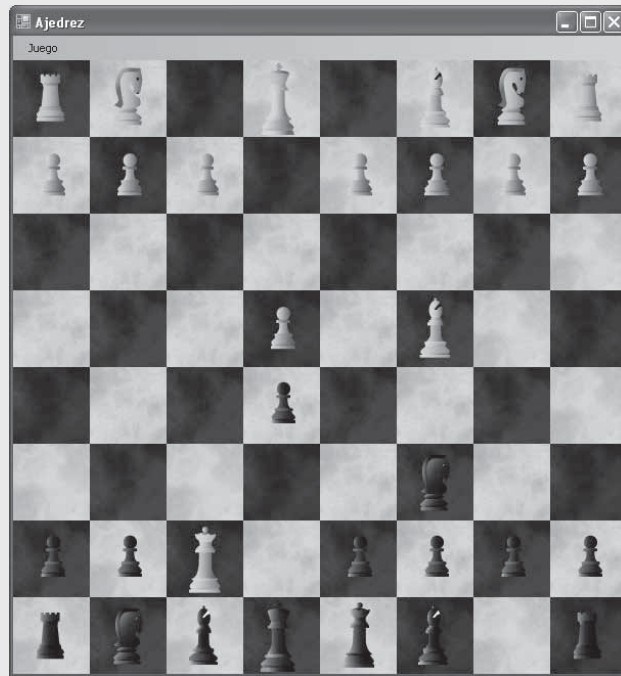


Figura 17.26 | Código del juego de ajedrez. (Parte 6 de 6).

La GUI del juego de ajedrez consta del formulario `JuegoAjedrezForm`, el área en la que dibujamos los mosaicos; el control `Panel piezaBox`, el área en la que dibujamos las piezas (observe que el color de fondo de `piezaBox` se establece a “transparente”); y un menú que permite al usuario iniciar un nuevo juego. Aunque las piezas y los mosaicos se pudieron haber dibujado en el mismo formulario, esto reduciría el rendimiento, ya que estaríamos obligados a actualizar el tablero y todas las piezas cada vez que actualizáramos el control.

El manejador de eventos `JuegoAjedrezForm_Load` (líneas 25-35) carga cuatro imágenes de mosaico en `mosaicoAjedrez`: dos mosaicos claros y dos mosaicos oscuros para tener variedad. Después llama al método `ReiniciarTablero` para actualizar el formulario y comenzar el juego. El método `ReiniciarTablero` (líneas 38-127) borra objetos `piezaAjedrez`, carga imágenes para los conjuntos de piezas de ajedrez negras y blancas, y crea el objeto `Bitmap` llamado `seleccion` para definir el conjunto del objeto `Bitmap` actual seleccionado. Las líneas 60-126 iteran a través de las 64 posiciones del tablero, estableciendo el color del mosaico y la pieza para cada mosaico. Las líneas 63-64 hacen que la imagen actual seleccionada cambie a las `piezasNegras` después de la quinta fila. Si el contador de filas se encuentra en la primera o en la última fila, las líneas 71-100 agregan una nueva pieza a `piezasAjedrez`. El tipo de la pieza se basa en la columna actual que inicializaremos. Las piezas en el ajedrez se posicionan en el siguiente orden, de izquierda a derecha: torre, caballo, alfil, reina, rey, alfil, caballo y torre. Las líneas 103-109 agregan un nuevo peón en la ubicación actual, si la fila actual es la segunda o la séptima.

Para definir un tablero de ajedrez, es necesario alternar los mosaicos claros y oscuros a lo largo de una fila, en un patrón en donde el color que empieza cada fila es igual al color del último mosaico de la fila anterior. Las líneas 113-122 asignan el color actual del mosaico del tablero a un elemento en el arreglo `tablero`. Con base en el valor alternante de la variable `bool` llamada `claro`, y los resultados de la operación al azar en la línea 111, asignaremos un `int` al tablero para determinar el color de ese mosaico: 0 y 1 representan mosaicos claros; 2 y 3 representan mosaicos oscuros. La línea 125 invierte el valor de `claro` al final de cada fila para mantener el efecto escalonado de un tablero de ajedrez.

El método `JuegoAjedrezForm_Paint` (líneas 130-146) maneja el evento `Paint` del formulario y dibuja los mosaicos de acuerdo con sus valores en el arreglo del tablero. Como la altura predeterminada de un objeto `MenuStrip` es de 24 píxeles, utilizamos el método `TranslateTransform` de la clase `Graphics` para desplazar el origen

del formulario 24 píxeles hacia abajo (línea 133). Este desplazamiento evita que la fila superior de mosaicos quede oculta detrás del objeto `MenuStrip`. El método `piezaBox_Paint` (líneas 168-174), que maneja el evento `Paint` para el objeto `Panel` `piezaBox`, itera a través de cada elemento del objeto `ArrayList` `piezaAjedrez` y llama a su método `Dibujar`.

El manejador del evento `MouseDown` de `piezaBox` (líneas 177-182) llama a `ComprobarLimites` (líneas 150-165) con la ubicación del ratón, para determinar si el usuario seleccionó una pieza.

El manejador del evento `MouseMove` de `piezaBox` (líneas 185-200) mueve la pieza seleccionada con el ratón. Las líneas 190-192 definen una región del control `Panel` que abarca dos mosaicos en cada dirección del puntero. Como dijimos antes, `Invalidate` es lento. Esto significa que el manejador del evento `MouseMove` de `piezaBox` podría llamarse varias veces antes de que termine el método `Invalidate`. Si un usuario que trabaje en una computadora lenta mueve el ratón con rapidez, la aplicación podría dejar *artefactos* atrás. Un artefacto es una anomalía visual inesperada en un programa gráfico. Al hacer que el programa actualice un rectángulo de dos cuadros, lo cual debería ser suficiente en la mayoría de los casos, logramos una considerable mejora en el rendimiento en relación con la actualización de todo un componente completo durante cada evento `MouseMove`. Las líneas 195-196 establecen la ubicación de la pieza seleccionada a la posición del cursor del ratón, ajustando la ubicación para centrar la imagen en el ratón. La línea 198 invalida la región definida en las líneas 190-192 para que se actualice.

Las líneas 203-228 definen el manejador del evento `MouseUp` de `piezaBox`. Si se seleccionó una pieza, las líneas 208-225 determinan el índice en `piezasAjedrez` de cualquier colisión de piezas, eliminan la pieza que provocó la colisión, ajustan (alinean) la pieza actual a una ubicación válida y deselectan la pieza. Revisamos si hay colisiones de piezas para permitir que la pieza de ajedrez “tome” otras piezas. La línea 216 comprueba si alguna pieza (excluyendo a la pieza actual seleccionada) se encuentra debajo de la ubicación actual del ratón. Si se detecta una colisión, el índice de la pieza devuelta se asigna a `eliminar`. Las líneas 211-213 determinan el mosaico de ajedrez válido más cercano y “ajustan” la pieza seleccionada a esa ubicación. Si `eliminar` contiene un valor positivo, la línea 224 elimina del objeto `ArrayList` `piezasAjedrez` el objeto en ese índice. Por último, todo el control `Panel` se invalida en la línea 227 para mostrar la nueva ubicación de la pieza y eliminar cualquier artefacto creado durante el movimiento.

El método `ComprobarLimites` (líneas 150-165) es un método auxiliar para detección de colisiones; itera a través de los objetos `ArrayList` `piezasAjedrez` y devuelve el índice del rectángulo de cualquier pieza que contenga el punto que se pasó al método (en este ejemplo, la ubicación del ratón). `ComprobarLimites` utiliza el método `Contains` de `Rectangle` para determinar si un punto está dentro del objeto `Rectangle`. El método `ComprobarLimites` puede excluir de manera opcional un índice de pieza individual (en este ejemplo, para ignorar el índice seleccionado en el manejador del evento `MouseUp` de `piezaBox`).

Las líneas 232-235 definen la función auxiliar `ObtenerPieza`, la cual simplifica la conversión de los objetos `object` en el objeto `ArrayList` `piezasAjedrez` para que sean de tipo `PiezaAjedrez`. El método `nuevoJuego_Item_Click` (líneas 238-243) maneja el evento `Click` del elemento de menú `NuevoJuego`, llama a `ReiniciarTablero` para reinicializar el juego e invalida el formulario por completo.

17.13 Reproductor de Windows Media

El control *Windows Media Player* permite que una aplicación reproduzca video y sonido en muchos *formatos multimedia*. Éstos incluyen el audio y video *MPEG* (*grupo de expertos en imágenes en movimiento*), el video *AVI* (*intercalación de audio y video*), el audio *WAV* (*formato de archivos de onda de Windows*) y el audio *MIDI* (*interfaz digital de instrumentos musicales*). Los usuarios pueden encontrar audio y video en Internet, o pueden crear sus propios archivos, usando los paquetes disponibles de sonido y gráficos.

La aplicación en la figura 17.27 demuestra el control *Windows Media Player*. Para usar este control, debe agregarlo al **Cuadro de herramientas**. Primero seleccione **Herramientas > Elegir elementos del cuadro de herramientas...** para que aparezca el cuadro de diálogo **Elegir elementos del cuadro de herramientas**. Haga clic en la ficha

```
1 // Fig. 17.27: PruebaReproductorMedios.cs
2 // El control Windows Media Player se utiliza para reproducir archivos de medios.
3 using System;
4 using System.Windows.Forms;
5
```

Figura 17.27 | Demostración del control *Windows Media Player*. (Parte 1 de 2).

```

6  public partial class ReproductorMedios : Form
7  {
8      // constructor predeterminado
9      public ReproductorMedios()
10     {
11         InitializeComponent();
12     } // fin del constructor
13
14     // abre nuevo archivo de medios en el Reproductor de Windows Media
15     private void abrirItem_Click( object sender, EventArgs e )
16     {
17         abrirArchivoMediosDialog.ShowDialog();
18
19         // carga y reproduce el clip de medios
20         player.URL = abrirArchivoMediosDialog.FileName;
21     } // fin del método abrirItem_Click
22
23     // sale del programa cuando se hace clic en el elemento de menú Salir
24     private void salirItem_Click( object sender, EventArgs e )
25     {
26         Application.Exit();
27     } // fin del método salirItem_Click
28 } // fin de la clase ReproductorMedios

```

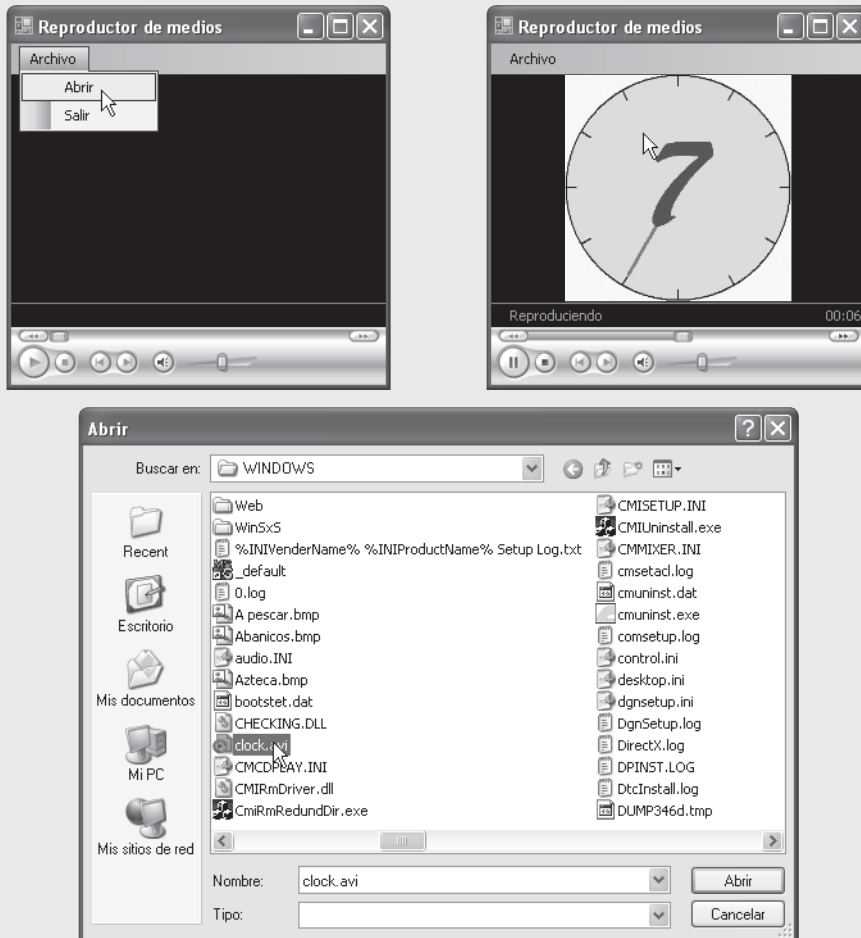


Figura 17.27 | Demostración del control Windows Media Player. (Parte 2 de 2).

Componentes COM y después desplácese hacia abajo y seleccione la opción **Windows Media Player**. Haga clic en el botón **Aceptar** para cerrar el cuadro de diálogo. Ahora el control Windows Media Player aparecerá en la parte inferior del **Cuadro de herramientas**.

El control Windows Media Player cuenta con varios botones que permiten al usuario reproducir el archivo actual, pausar, detener, reproducir el archivo anterior, rebobinar, adelantar y reproducir el siguiente archivo. El control también incluye un control de volumen y barras de rastreo para seleccionar una posición específica en el archivo de medios.

Nuestra aplicación proporciona un menú **Archivo** que contiene los elementos de menú **Abrir** y **Salir**. Cuando un usuario selecciona **Abrir** del menú **Archivo**, se ejecuta el manejador de eventos `abrirItem_Click` (líneas 15-21). A continuación aparece un cuadro de diálogo `OpenFileDialog` (línea 17) para permitir al usuario seleccionar un archivo. Después el programa establece la propiedad URL del reproductor (el objeto del control Windows Media Player de tipo **AxMediaPlayer**) con el nombre del archivo seleccionado por el usuario. La propiedad URL especifica el archivo que el Reproductor de Windows Media utiliza en ese momento.

El manejador de eventos `salirItem_Click` (línea 24-27) se ejecuta cuando el usuario selecciona **Salir** del menú **Archivo**. Este manejador de eventos sólo hace una llamada a `Application.Exit` para terminar la aplicación. En el directorio que contiene el ejemplo, incluimos archivos de audio y video de ejemplo.

17.14 Microsoft Agent

Microsoft Agent es una tecnología que se utiliza para agregar *personajes animados interactivos* a las aplicaciones Windows o a las páginas Web. Los personajes de Microsoft Agent pueden hablar y responder a la entrada del usuario, a través del reconocimiento y la síntesis de voz. Microsoft emplea su tecnología Agent en aplicaciones tales como Word, Excel y PowerPoint. Los agentes en estos programas ayudan a los usuarios a encontrar respuestas a sus preguntas, y a comprender la forma en que funcionan las aplicaciones.

El control Microsoft Agent proporciona a los programadores el acceso a cuatro personajes predefinidos: *Genie* (un genio), *Merlin* (un mago), *Peedy* (un perico) y *Robby* (un robot). Cada personaje tiene un conjunto único de animaciones que los programadores pueden usar en sus aplicaciones, para ilustrar distintos puntos y funciones. Por ejemplo, el conjunto de animaciones del personaje Peedy incluye distintas animaciones de vuelo, que el programador puede utilizar para desplazar a Peedy en la pantalla. Microsoft proporciona información básica sobre la tecnología Agent en

www.microsoft.com/msagent

La tecnología Microsoft Agent permite a los usuarios interactuar con las aplicaciones y las páginas Web mediante la voz, la forma más natural de comunicación humana. Para comprender la voz, el control utiliza un *motor de reconocimiento de voz*: una aplicación que traduce la entrada de sonido vocal de un micrófono a un lenguaje que la computadora entiende. El control Microsoft Agent también utiliza un *motor de texto a voz*, el cual genera las respuestas habladas de los personajes. Un motor de texto a voz es una aplicación que traduce las palabras escritas en sonido de audio que los usuarios escuchan a través de auriculares o altavoces conectadas a una computadora. Microsoft proporciona motores de reconocimiento de voz y de texto a voz para varios lenguajes en

www.microsoft.com/msagent/downloads/user.asp

Incluso, los programadores pueden crear sus propios personajes animados con la ayuda del *Editor de personajes de Microsoft Agent* (*Microsoft Agent Character Editor*) y la *Herramienta de edición de sonido lingüístico de Microsoft* (*Microsoft Linguistic Sound Editing Tool*). Estos productos se pueden descargar sin costo de

www.microsoft.com/msagent/downloads/developer.asp

Esta sección presenta las herramientas básicas del control Microsoft Agent. Para obtener todos los detalles relacionados con la descarga de este control, visite

www.microsoft.com/msagent/downloads/user.asp

El siguiente ejemplo, Peedy's Pizza Palace, lo desarrolló Microsoft para ilustrar las herramientas del control Microsoft Agent. Peedy's Pizza Palace es una pizzería en línea, en donde los usuarios pueden colocar sus pedidos mediante la entrada de voz. El personaje Peedy interactúa con los usuarios, ayudándoles a seleccionar ingredientes y a calcular el total de sus pedidos. Puede ver este ejemplo en

agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm

Para ejecutar el ejemplo, debe ir a www.microsoft.com/msagent/downloads/user.asp y descargar e instalar el archivo del personaje Peedy, un motor de texto a voz y un motor de reconocimiento de voz.

Al abrir la ventana, Peedy se presenta a sí mismo (figura 17.28) y las palabras que diga aparecen en una burbuja de caricatura, por encima de su cabeza. Observe que las animaciones de Peedy corresponden a las palabras que dice.

Los programadores pueden sincronizar las animaciones con la salida de voz para ilustrar un punto, o para transmitir el humor de un personaje. Por ejemplo, la figura 17.29 ilustra la animación de Peedy *complacido*. El conjunto de animaciones del personaje Peedy incluye 85 animaciones distintas, cada una de las cuales es única para el personaje Peedy.

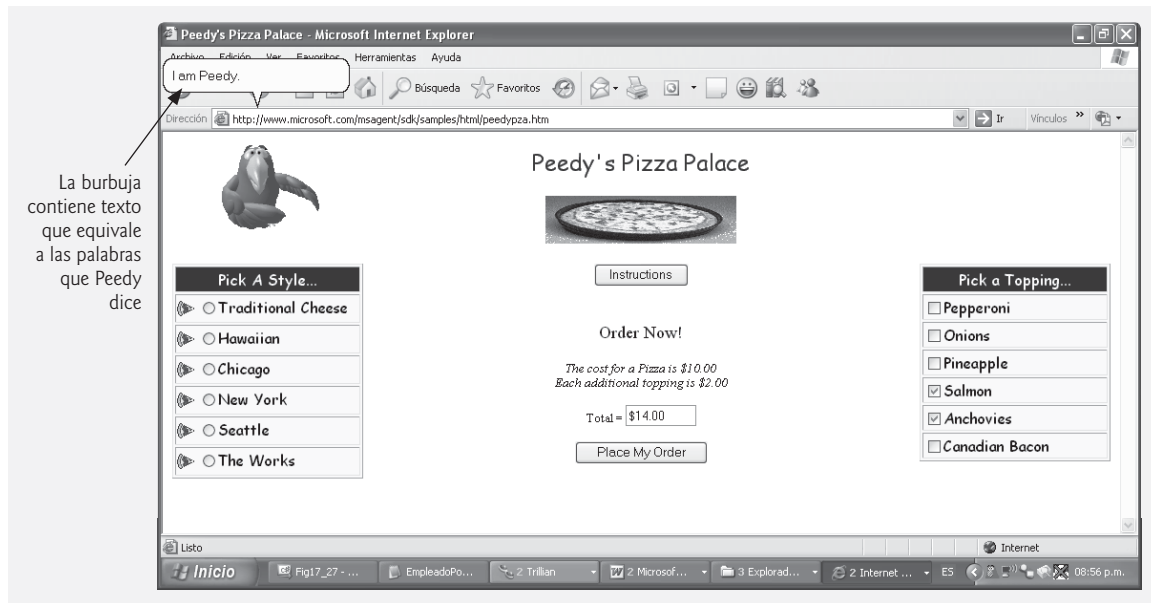


Figura 17.28 | Peedy presentándose a sí mismo cuando se abre la ventana.

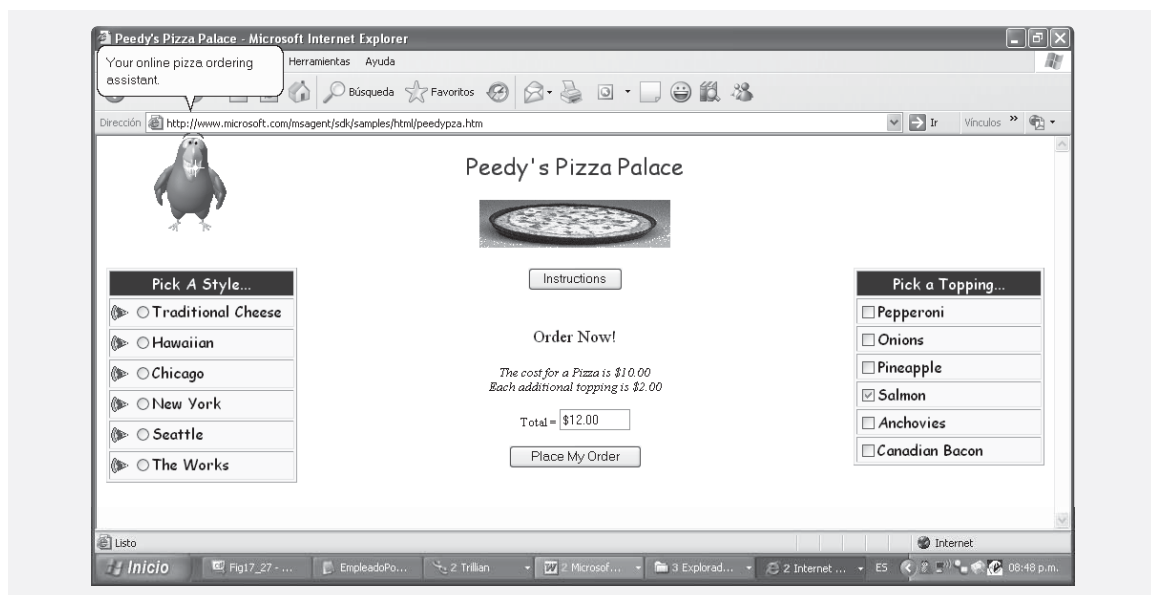


Figura 17.29 | Animación de Peedy *complacido*.



Observación de apariencia visual 17.1

Los personajes de Agent permanecen encima de todas las ventanas activas mientras haya una aplicación de Microsoft Agent en ejecución. Sus movimientos no se limitan por el contorno del explorador Web o de la ventana de una aplicación.

Peedy también responde a la entrada mediante el teclado y el ratón. La figura 17.30 muestra lo que ocurre cuando un usuario hace clic en Peedy con el puntero del ratón. Peedy salta, mueve sus plumas y exclama, “Hey, that tickles!” (“¡Hey, me haces cosquillas!”) o “Be careful with that pointer!” (“¡Ten cuidado con ese puntero!”). Los usuarios pueden reposicionar a Peedy en la pantalla, arrastrándolo con el ratón. No obstante, aun cuando el usuario mueva a Peedy a otra parte de la pantalla, él continuará realizando sus animaciones preestablecidas y sus cambios de ubicación.

Muchos cambios de ubicación implican animaciones. Por ejemplo, Peedy puede saltar de una ubicación en la pantalla a otra, o puede volar (figura 17.31).

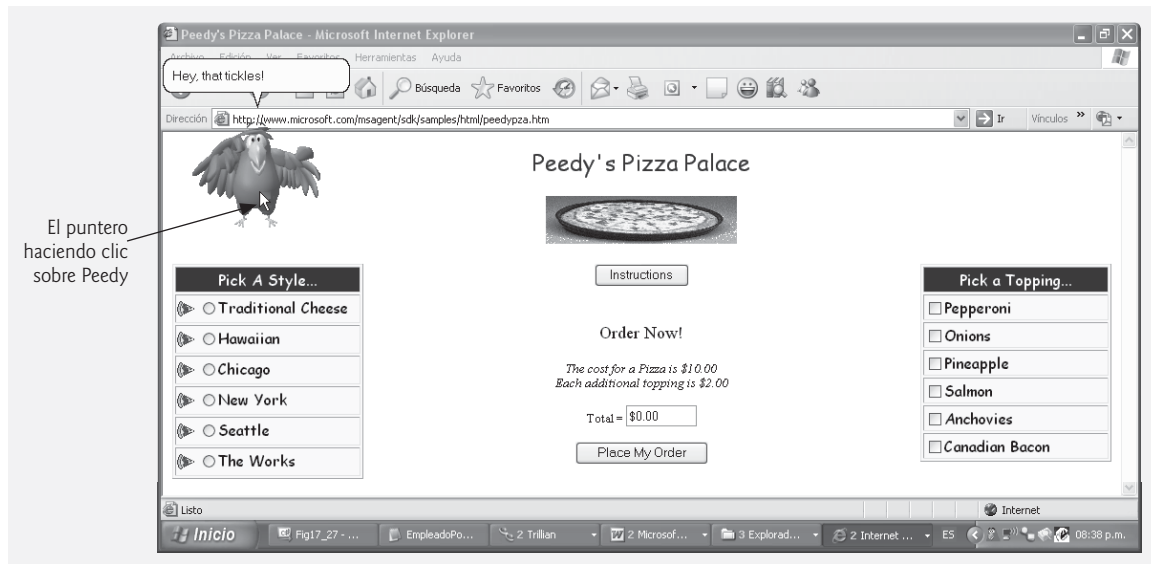


Figura 17.30 | Reacción de Peedy cuando hacemos clic con el botón del ratón sobre él.

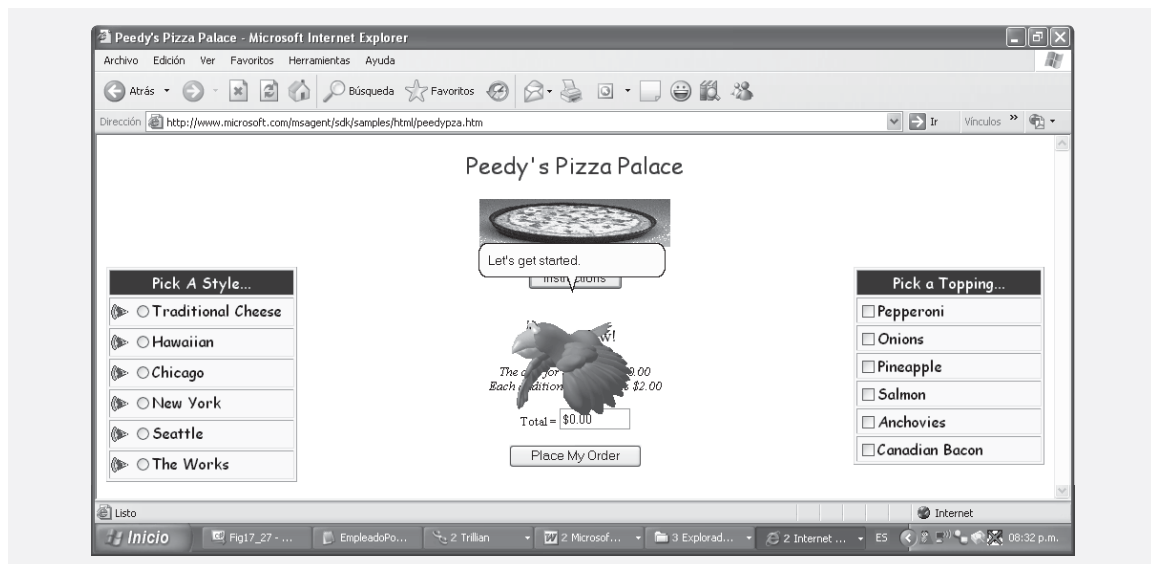


Figura 17.31 | Animación de Peedy volando.

Una vez que Peedy complete las instrucciones para realizar pedidos, aparecerá un cuadro de información sobre la herramienta debajo de él, indicando que está escuchando en espera de un comando de voz (figura 17.32). Puede indicar el tipo de pizza que desea ordenar, ya sea diciendo el nombre del estilo en un micrófono, o haciendo clic en el botón correspondiente a su elección.

Si selecciona la entrada de voz, aparecerá un cuadro debajo de Peedy, en el que se mostrarán las palabras que Peedy “escuchó” (es decir, las palabras que el motor de reconocimiento de voz traduce para el programa). Una vez que reconozca su entrada, Peedy le proporcionará una descripción de la pizza seleccionada. La figura 17.33 muestra lo que ocurre cuando seleccionamos **Seattle** como el estilo de la pizza.

Después Peedy le pedirá que seleccione ingredientes extra. De nuevo, puede hablar o usar el ratón para hacer una selección. Las casillas de verificación correspondientes a los ingredientes que incluye el estilo de pizza seleccionado ya están seleccionados por usted. La figura 17.34 muestra lo que ocurre cuando seleccionamos anchoas como ingrediente extra. Peedy hace un chiste relacionado con la elección que usted hizo.

Puede colocar el pedido, ya sea oprimiendo el botón **Place My Order** o diciendo las palabras “Place order” en el micrófono. Peedy contará el pedido mientras anota los artículos en su bloc de notas (figura 17.35). Después calculará las cifras en su calculadora y reportará el precio total (figura 17.36).

Creación de una aplicación que utiliza Microsoft Agent

[Nota: antes de ejecutar este ejemplo, primero debe descargar e instalar el control Microsoft Agent, un motor de reconocimiento de voz, un motor de texto a voz y las cuatro definiciones de los personajes del sitio Web de Microsoft Agent, como dijimos al principio de la sección.]

El siguiente ejemplo (figura 17.37) demuestra cómo crear una aplicación simple con el control Microsoft Agent. Esta aplicación contiene dos listas desplegables, desde las que el usuario puede seleccionar un personaje de Agent y una animación para el personaje. Cuando el usuario selecciona de estas listas, aparece el personaje seleccionado y ejecuta la animación seleccionada. La aplicación utiliza reconocimiento y síntesis de voz para controlar las animaciones y la voz de los personajes; usted puede indicar al personaje cuál animación debe realizar oprimiendo la tecla *Blog despl* y después pronunciando el nombre de la animación en un micrófono.

Este ejemplo también le permite cambiar a un nuevo personaje al decir su nombre, y crea un comando personalizado, *MoveToMouse*. Además, al oprimir el botón **Hablar**, los personajes dicen cualquier texto que usted haya escrito en el control TextBox.

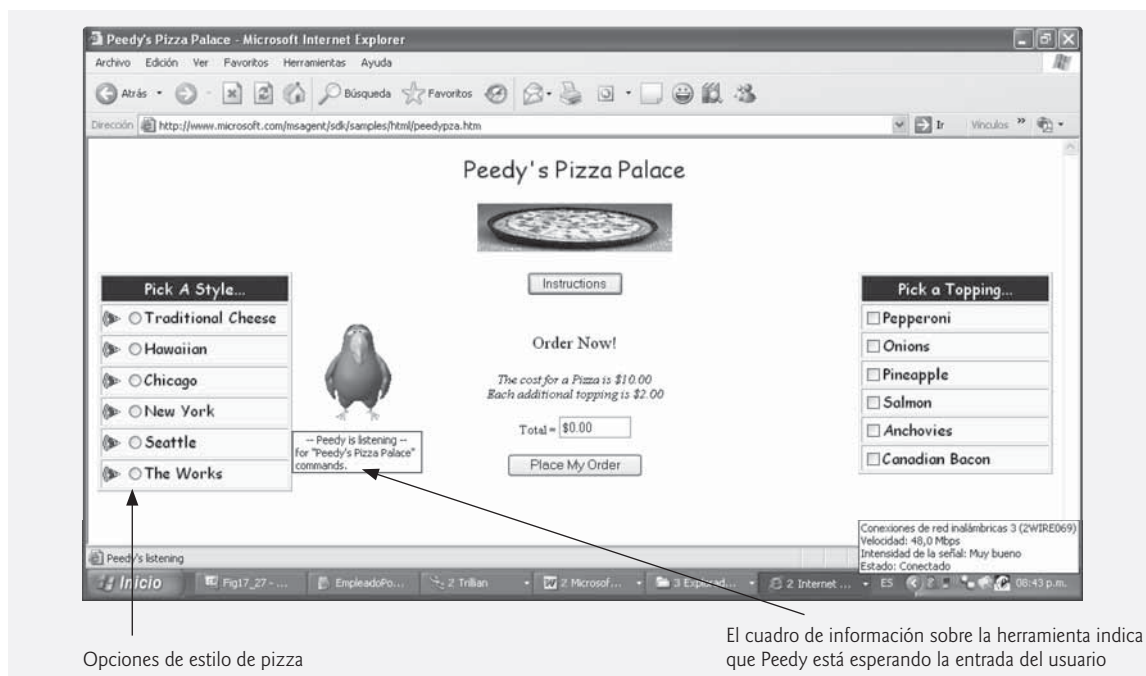


Figura 17.32 | Peedy esperando la entrada de voz.

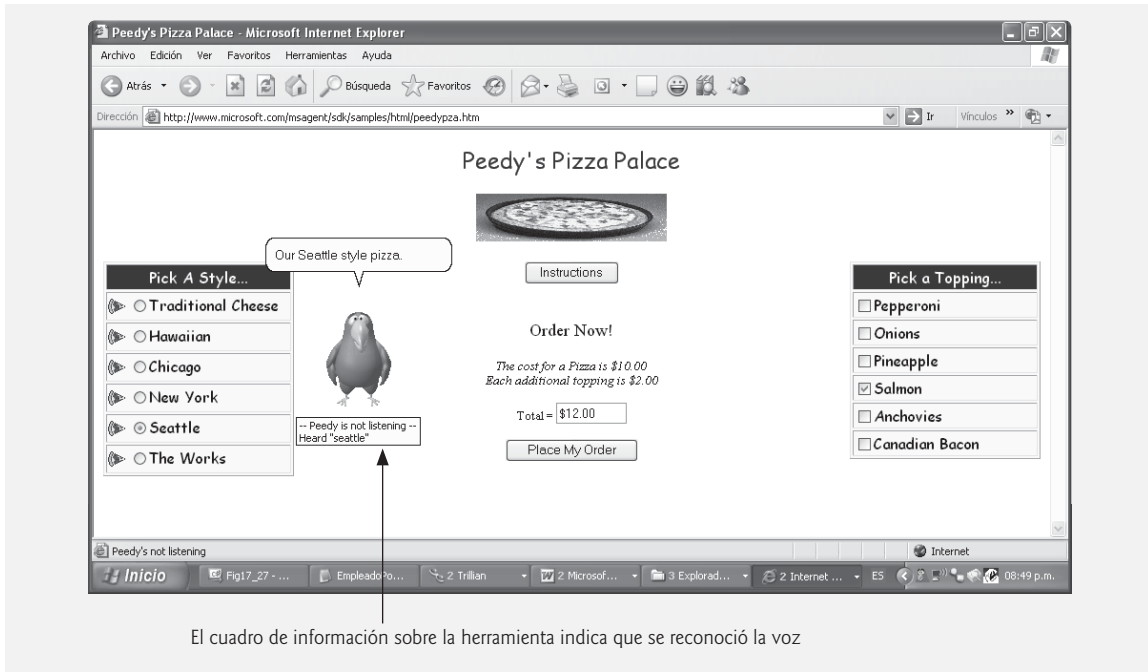


Figura 17.33 | Peedy repitiendo la orden de la pizza estilo Seattle.

Para usar el control Microsoft Agent, debe agregarlo al **Cuadro de herramientas**. Seleccione **Herramientas > Elegir elementos del cuadro de herramientas...** para que aparezca el cuadro de diálogo **Elegir elementos del cuadro de herramientas**. En el cuadro de diálogo, seleccione la ficha **Componentes COM**, desplácese hacia abajo y seleccione la opción **Microsoft Agent Control 2.0**. Cuando se selecciona esta opción de manera apropiada, aparece una pequeña marca de verificación en el cuadro a la izquierda de la opción. Haga clic en **Aceptar** para cerrar el cuadro de diálogo. Ahora el icono del control Microsoft Agent aparecerá en la parte inferior del **Cuadro de herramientas**. Arrastre el control **Microsoft Agent Control 2.0** en su formulario y cambie el nombre a **mainAgent**.

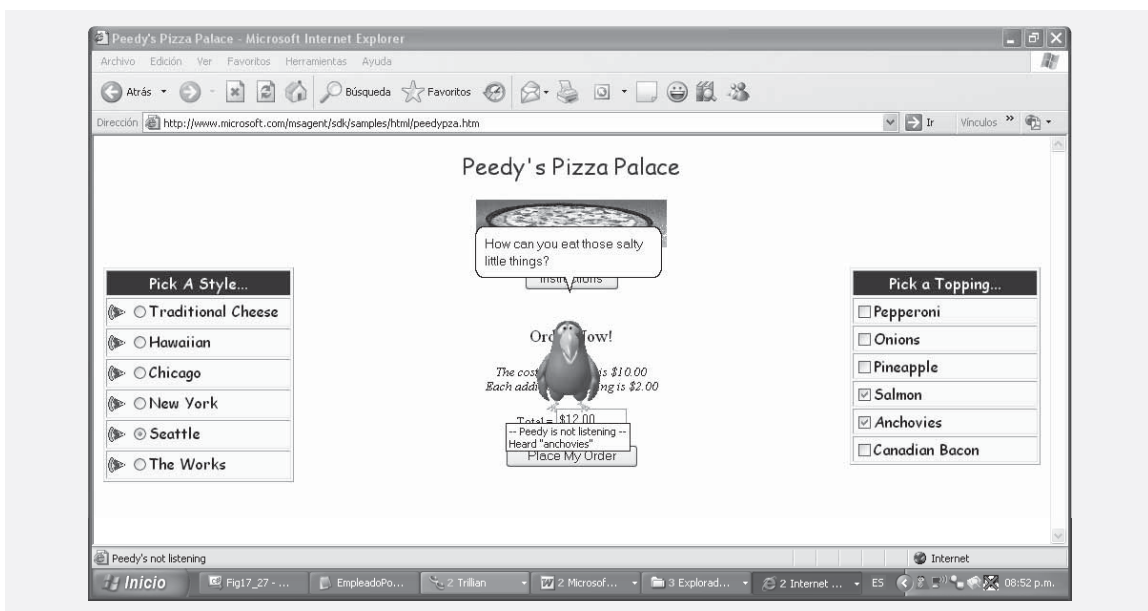


Figura 17.34 | Peedy repitiendo el pedido de anchoas como ingrediente extra.

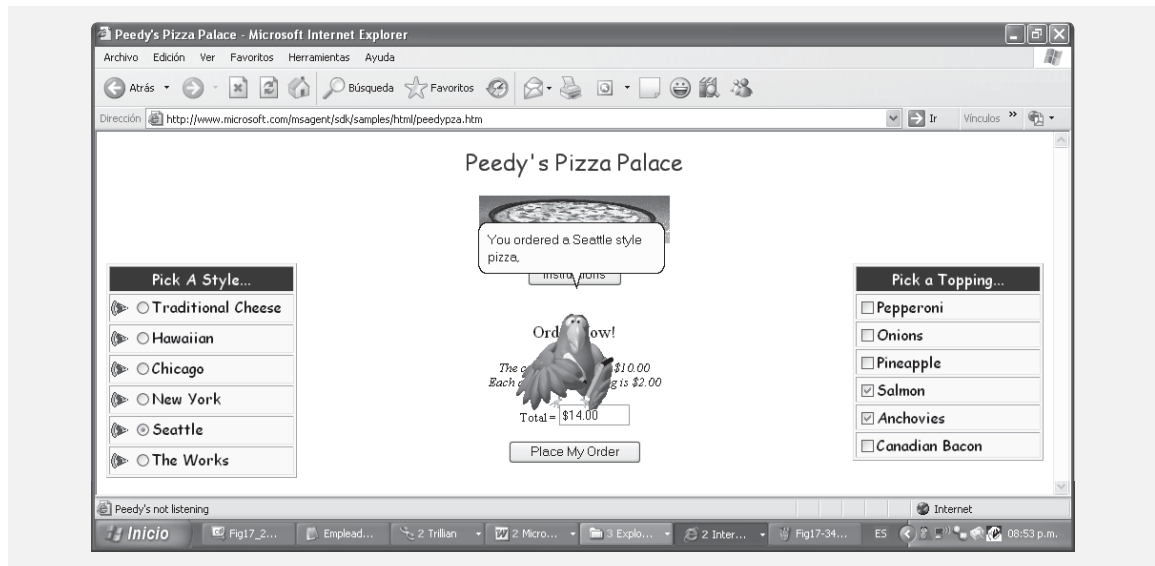


Figura 17.35 | Peedy contando el pedido.

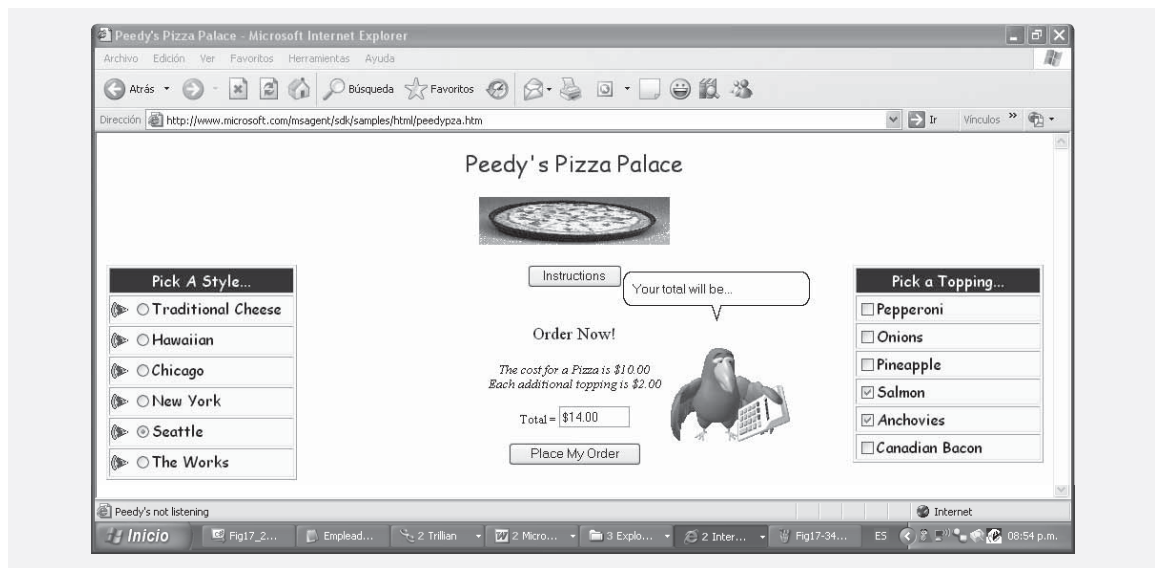


Figura 17.36 | Peedy calculando el total.

```

1 // Fig. 17.37: Agent.cs
2 // Demostración de Microsoft Agent.
3 using System;
4 using System.Collections;
5 using System.Windows.Forms;
6 using System.IO;
7
8 public partial class Agent : Form
9 {
10 // objeto agente actual

```

Figura 17.37 | Demostración de Microsoft Agent. (Parte I de 5).

```

11  private AgentObjects.IAgentCtlCharacter orador;
12
13  // constructor predeterminado
14  public Agent()
15  {
16      InitializeComponent();
17
18      // inicializa los personajes
19      try
20      {
21          // carga los personajes en el objeto agente
22          mainAgent.Characters.Load( "Merlín",
23              @"C:\windows\msagent\chars\Merlin.acs" );
24          mainAgent.Characters.Load( "Genio",
25              @"C:\windows\msagent\chars\Genie.acs" );
26          mainAgent.Characters.Load( "Lorito",
27              @"C:\windows\msagent\chars\Peedy.acs" );
28          mainAgent.Characters.Load( "Roby",
29              @"C:\windows\msagent\chars\Robby.acs" );
30
31          // establece el personaje actual a Merlín y lo muestra
32          orador = mainAgent.Characters[ "Merlín" ];
33          ObtenerNombresAnimaciones(); // obtiene una lista de nombres de animaciones
34          orador.Show( 0 ); // muestra a Merlín
35          personajeCombo.SelectedText = "Merlín";
36      } // fin de try
37      catch ( FileNotFoundException )
38      {
39          MessageBox.Show( "Ubicación del personaje inválida",
40              "Error", MessageBoxButtons.OK, MessageBoxIcon.Error );
41      } // fin de catch
42  } // fin del constructor
43
44  // manejador de eventos para el control hablarButton
45  private void hablarButton_Click( object sender, EventArgs e )
46  {
47      // si el cuadro de texto está vacío, hace que el personaje pida
48      // al usuario que escriba las palabras en el objeto TextBox; en caso contrario,
49      // hace que el personaje diga las palabras que haya en el objeto TextBox
50      if ( vozTextBox.Text == "" )
51          orador.Speak(
52              "Por favor, escribe las palabras que quieres que diga", "" );
53      else
54          orador.Speak( vozTextBox.Text, "" );
55  } // fin del método hablarButton_Click
56
57  // manejador de eventos para el evento ClickEvent del control Agent
58  private void mainAgent_ClickEvent(
59      object sender, AxAgentObjects._AgentEvents_ClickEvent e )
60  {
61      orador.Play( "Confused" );
62      orador.Speak( "Por qué me estás molestando?", "" );
63      orador.Play( "RestPose" );
64  } // fin del método mainAgent_ClickEvent
65
66  // el control ComboBox cambió el evento, cambia el personaje activo de Agent
67  private void personajeCombo_SelectedIndexChanged(
68      object sender, EventArgs e )
69  {

```

Figura 17.37 | Demostración de Microsoft Agent. (Parte 2 de 5).

```

70     CambiarPersonaje( personajeCombo.Text );
71 } // fin del método personajeCombo_SelectedIndexChanged
72
73 // método utilitario para cambiar personajes
74 private void CambiarPersonaje( string nombre )
75 {
76     orador.StopAll( "Play" );
77     orador.Hide( 0 );
78     orador = mainAgent.Characters[ nombre ];
79
80     // regenera la lista de nombres de animaciones
81     ObtenerNombresAnimaciones();
82     orador.Show( 0 );
83 } // fin del método CambiarPersonaje
84
85 // obtiene los nombres de las animaciones y los almacena en el objeto ArrayList
86 private void ObtenerNombresAnimaciones()
87 {
88     // preserva la seguridad de los subprocesos
89     lock ( this )
90     {
91         // obtiene los nombres de las animaciones
92         IEnumerator enumerador = mainAgent.Characters[
93             orador.Name ].AnimationNames.GetEnumerator();
94
95         string vozString;
96
97         // borra accionesCombo
98         accionesCombo.Items.Clear();
99         orador.Commands.RemoveAll();
100
101         // copia la enumeración al objeto ArrayList
102         while ( enumerador.MoveNext() )
103         {
104             // elimina los guiones bajos en la cadena de voz
105             vozString = ( string ) enumerador.Current;
106             vozString = vozString.Replace( "_", "underscore" );
107
108             accionesCombo.Items.Add( enumerador.Current );
109
110             // agrega todas las animaciones como comandos habilitados por voz
111             orador.Commands.Add( ( string ) enumerador.Current,
112                 enumerador.Current, vozString, true, false );
113         } // fin de while
114
115         // agrega comando personalizado
116         orador.Commands.Add( "MoveToMouse", "MoveToMouse",
117             "MoveToMouse", true, true );
118     } // fin de lock
119 } // fin del método ObtenerNombresAnimaciones
120
121 // el usuario selecciona una nueva acción
122 private void accionesCombo_SelectedIndexChanged(
123     object sender, EventArgs e )
124 {
125     orador.StopAll( "Play" );
126     orador.Play( accionesCombo.Text );
127     orador.Play( "RestPose" );
128 } // fin del método accionesCombo_SelectedIndexChanged

```

Figura 17.37 | Demostración de Microsoft Agent. (Parte 3 de 5).


```

129
130 // manejador de eventos para los comandos de Agent
131 private void mainAgent_Command(
132     object sender, AxAgentObjects._AgentEvents_CommandEvent e )
133 {
134     // obtiene objeto userInput
135     AgentObjects.IAgentCtlUserInput comando =
136         ( AgentObjects.IAgentCtlUserInput ) e.userInput;
137
138     // cambia de personaje si el usuario dice el nombre de un personaje
139     if ( comando.Voice == "Lorito" || comando.Voice == "Roby" ||
140         comando.Voice == "Merlín" || comando.Voice == "Genio" )
141     {
142         CambiarPersonaje( comando.Voice );
143         return;
144     } // fin de if
145
146     // envía agente al ratón
147     if ( comando.Voice == "MoveToMouse" )
148     {
149         orador.MoveTo( Convert.ToInt16( Cursor.Position.X - 60 ),
150             Convert.ToInt16( Cursor.Position.Y - 60 ), 5 );
151         return;
152     } // fin de if
153
154     // reproduce nueva animación
155     orador.StopAll( "Play" );
156     orador.Play( comando.Name );
157 }
158 } // fin de la clase Agent

```

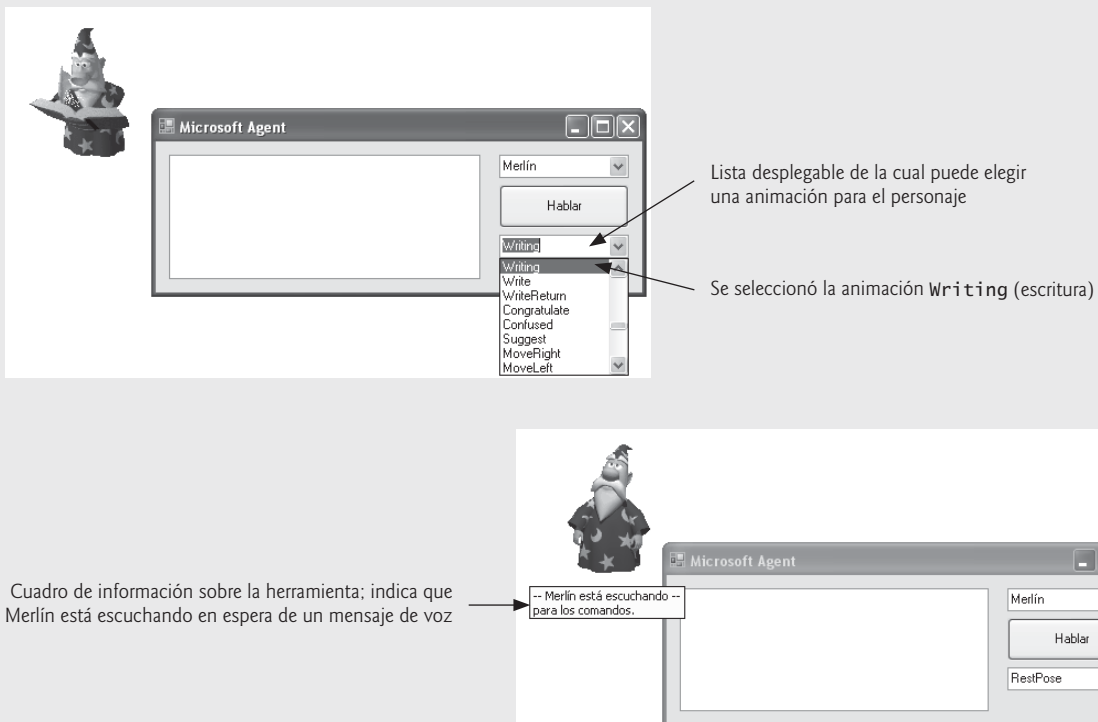


Figura 17.37 | Demostración de Microsoft Agent. (Parte 4 de 5).

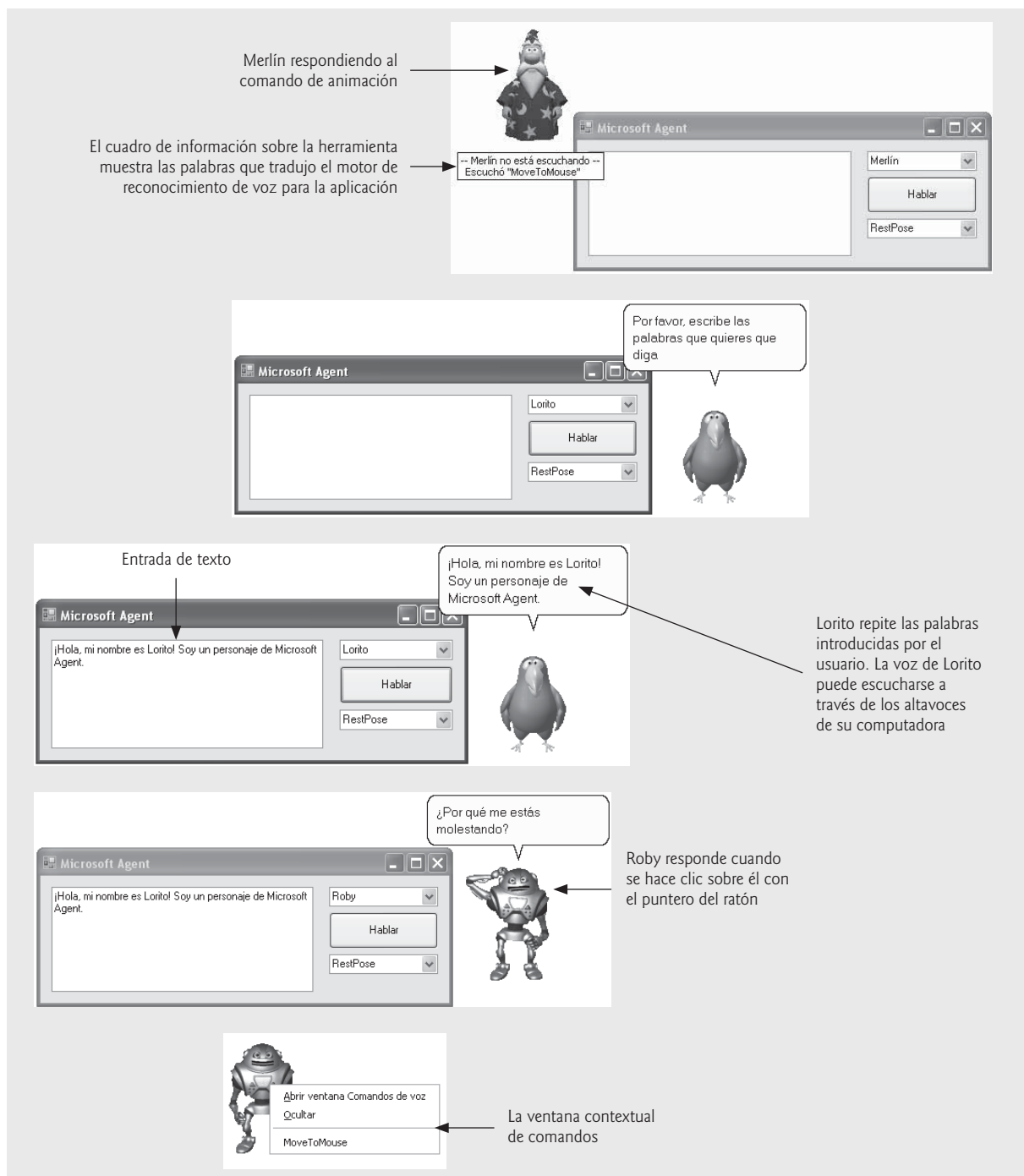


Figura 17.37 | Demostración de Microsoft Agent. (Parte 5 de 5).

Además del objeto `mainAgent` (de tipo ***AxAgent***) de Microsoft Agent que administra los personajes, también necesita una variable de tipo ***IAgentCtlCharacter*** para representar el personaje actual. En la línea 11 creamos esta variable, llamada `orador`.

Al ejecutar este programa, el constructor de la clase `Agent` (líneas 14-42) carga las descripciones de los personajes animados predefinidos (líneas 22-29). Si la ubicación especificada de los personajes es incorrecta, o si

falta alguno de los personajes, se lanza una excepción `FileNotFoundException`. De manera predeterminada, las descripciones de los personajes se almacenan en `C:\Windows\msagent\chars`. Si su sistema utiliza otro nombre para el directorio Windows, necesitará modificar las rutas en las líneas 22-29.

Las líneas 32-34 establecen a Merlín como el personaje predeterminado, obtiene los nombres de todas las animaciones a través de nuestro método utilitario `ObtenerNombresAnimaciones` y llama al método `Show` de `IAgentCtlCharacter` para mostrar el personaje. Para acceder a los personajes utilizamos la propiedad `Characters` de `mainAgent`, que contiene todos los personajes que se cargaron. Utilizamos el indexador de la propiedad `Characters` para especificar el nombre del personaje que deseamos cargar (Merlín).

Responder al evento ClickEvent del control Agent

Cuando un usuario hace clic con el botón del ratón sobre el personaje, se ejecuta el manejador de eventos `mainAgent_ClickEvent` (líneas 58-64). Primero, el método `Play` de orador reproduce una animación. Este método acepta como argumento un objeto `string` que representa una de las animaciones predefinidas para el personaje (en el sitio Web de Microsoft Agent encontrará una lista de animaciones para cada uno de los personajes; cada personaje ofrece más de 70 animaciones). En nuestro ejemplo, el argumento para `Play` es "Confused" (confundido; esta animación está definida para los cuatro personajes, cada uno de los cuales expresa esta emoción de una forma única. Después el personaje dice "¿Por qué me estás molestando?" a través de una llamada al método `Speak`. Por último, reproducimos la animación `RestPose`, que devuelve al carácter a su pose neutral de descanso.

Obtener la lista de animaciones de un personaje y definir sus comandos

La lista de comandos válidos para un personaje está contenida en la propiedad `Commands` del objeto `IAgentCtlCharacter` (en este ejemplo, orador). Puede ver los comandos para un personaje en la ventana contextual `Commands`, la cual aparece cuando el usuario hace clic con el botón derecho sobre un personaje de Agent (la última captura de pantalla de la figura 17.37). El método `Add` de la propiedad `Commands` agrega un nuevo comando a la lista de comandos. El método `Add` recibe tres argumentos `string` y dos argumentos `bool`. El primer argumento `string` identifica el nombre del comando, el cual utilizamos para identificar al comando mediante la programación. El segundo objeto `string` define el nombre de comando que debe aparecer en la ventana contextual `Commands`. El tercer objeto `string` define la entrada de voz que activa el comando. El primer objeto `bool` especifica si el comando está activo, y el segundo `bool` indica si el comando está visible en la ventana contextual `Commands`. Un comando se activa cuando el usuario lo selecciona desde la ventana contextual `Commands`, o cuando habla en un micrófono para generar la entrada de voz. La lógica del comando se maneja en el manejador de eventos `Command` del control `AxAgent` (en este ejemplo, `mainAgent`). Además, Agent define varios comandos globales que tienen funciones predefinidas (por ejemplo, al decir el nombre de un personaje aparece ese personaje en pantalla).

El método `ObtenerNombresAnimaciones` (líneas 86-119) llena el objeto `ComboBox accionesCombo` con el listado de animaciones del personaje actual, y define los comandos válidos que pueden usarse con ese personaje. El método contiene un bloque `lock` para evitar los errores que resulten de los cambios rápidos de los personajes. El método utiliza un objeto `IEnumerator` (líneas 92-93) para obtener las animaciones del personaje actual. Las líneas 98-99 borran los elementos existentes en el control `ComboBox` y la propiedad `Commands` del personaje. Las líneas 102-113 iteran a través de todos los elementos en el enumerador de los nombres de las animaciones. Para cada animación, la línea 105 asigna el nombre de la animación al objeto `string` `vozString`. La línea 106 elimina cualquier carácter de guión bajo (`_`) y lo sustituye con el objeto `string` "underscore"; esto cambia el objeto `string` para que un usuario pueda pronunciarlo y emplearlo como activador de comandos. La línea 108 agrega el nombre de la animación al control `ComboBox accionesCombo`. El método `Add` de la propiedad `Commands` (líneas 111-112) agrega un nuevo comando al personaje actual. En este ejemplo, agregamos cada nombre de animación como un comando. Cada llamada a `Add` recibe el nombre de la animación como el nombre del comando y como el objeto `string` que aparece en la ventana contextual `Commands`. El tercer argumento es el comando de voz, y los últimos dos argumentos habilitan el comando, pero indican que no está disponible a través de la ventana contextual `Commands`. Por ende, el comando puede activarse sólo mediante entrada de voz. Las líneas 116-117 crean un nuevo comando llamado `MoveToMouse`, el cual está visible en la ventana contextual `Commands`.

Responder a las selecciones del control ComboBox accionesCombo

Una vez que se ha llamado el método `ObtenerNombresAnimaciones`, el usuario puede seleccionar un valor del control `ComboBox accionesCombo`. El manejador de eventos `accionesCombo_SelectedIndexChanged` (líneas

122-128) detiene cualquier animación actual y después reproduce la animación que seleccionó el usuario del control ComboBox, seguida de la animación *RestPose*.

Decir el texto escrito por el usuario

También puede escribir texto en el control TextBox y hacer clic en **Hablar**. Esto hace que el manejador de eventos *hablarButton_Click* (línea 45-55) llame al método *Speak* de *orador*, suministrando el texto en *vozTextBox* como argumento. Si el usuario hace clic en **Hablar** sin proporcionar texto, el personaje dice, "Por favor, escribe las palabras que quieres que diga".

Cambiar personajes

En cualquier punto en el programa, el usuario puede seleccionar un personaje distinto del control ComboBox *personajeCombo*. Cuando esto ocurre, se ejecuta el manejador de eventos *SelectedIndexChanged* para *personajeCombo* (líneas 67-71). El manejador de eventos llama al método *CambiarPersonaje* (líneas 74-83) con el texto en el control *personajeCombo* como argumento. El método *CambiarPersonaje* detiene cualquier animación actual y después llama al método *Hide* de *orador* (línea 77) para eliminar el personaje actual de la pantalla. La línea 78 asigna el personaje recién seleccionado a *orador*, la línea 81 genera los nombres de las animaciones y comandos del personaje, y la línea 82 muestra el personaje mediante una llamada al método *Show*.

Responder a los comandos

Cada vez que un usuario oprime la tecla *Bloq Despl* y habla en un micrófono o selecciona un comando de la ventana contextual **Commands**, se hace una llamada al manejador de eventos *mainAgent_Command* (líneas 131-157). Este método recibe un argumento de tipo *AxAgentObjects._AgentEvents_CommandEvent*, el cual contiene una sola propiedad, *userInput*. Esta propiedad devuelve un objeto *Object* que puede convertirse al tipo *AgentObjects.IAgentCtlUserInput*. Las líneas 135-136 asignan el objeto *userInput* a un objeto *IAgentCtlUserInput* llamado *comando*, el cual se utiliza para identificar el comando, de manera que el programa pueda responder en forma apropiada. Las líneas 139-144 utilizan el método *CambiarPersonaje* para cambiar el personaje actual de *Agent* si el usuario pronuncia el nombre de un personaje. Microsoft Agent siempre mostrará un personaje cuando un usuario diga su nombre; no obstante, si controlamos el cambio del personaje, nos aseguraremos que sólo se muestre un personaje de *Agent* a la vez. Las líneas 147-152 desplazan el personaje a la ubicación actual del ratón si el usuario invoca el comando *MoveToMouse*. El método *MoveTo* de *Agent* recibe las coordenadas *x-y* como argumentos y desplaza el personaje a la posición especificada en la pantalla, aplicando las animaciones de movimiento apropiadas. Para todos los demás comandos, usamos el método *Play* para reproducir el nombre del comando como una animación en la línea 156.

17.15 Conclusión

Este capítulo comenzó con una introducción a las herramientas de dibujo del .NET Framework. Presentamos herramientas de dibujo más poderosas, como cambiar los estilos de las líneas usadas para dibujar figuras y controlar los colores y patrones de las figuras rellenas.

Aprendió las técnicas para manipular imágenes y crear animaciones uniformes. Hablamos sobre la clase *Image*, que puede almacenar y manipular imágenes de varios formatos. Explicamos cómo combinar las herramientas de representación gráfica cubiertas en las primeras secciones del capítulo, con las secciones dedicadas a la manipulación de imágenes.

También aprendió a incorporar el control Windows Media Player en una aplicación para reproducir audio o video. Por último, mostramos la tecnología Microsoft Agent para agregar personajes animados interactivos a las aplicaciones o páginas Web; después mostramos cómo incorporar Microsoft Agent en una aplicación para agregar capacidades de síntesis y reconocimiento de voz. En el siguiente capítulo hablaremos sobre las técnicas de procesamiento de archivos que permiten a los programas almacenar y recuperar datos del almacenamiento persistente, como el disco duro de su computadora. También exploraremos varios tipos de flujos incluidos en Visual Studio .NET.

18

Archivos y flujos

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Crear, leer, escribir y actualizar archivos.
- Usar la jerarquía de clases de flujos de C#.
- Usar las clases `File` y `Directory` para obtener información acerca de los archivos y directorios en su computadora.
- Familiarizarse con el procesamiento de archivos de acceso secuencial.
- Utilizar las clases `FileStream`, `StreamReader` y `StreamWriter` para leer texto desde, y escribir texto hacia, archivos.
- Utilizar las clases `FileStream` y `BinaryFormatter` para leer objetos desde, y escribir objetos hacia, archivos.

*Sólo puedo suponer que un documento
“No archivar” se archiva en un
archivo “No archivar”.*

—Senador Frank Church

Audiencia del subcomité de inteligencia
del Senado, 1975

*La conciencia... no aparece por sí
misma cortada en pequeños pedazos...
Un “río” o un “flujo” son las metáforas
por las cuales se describe con más
naturalidad.*

—William James

Leí una parte en su totalidad.

—Samuel Goldwyn

- 18.1 Introducción
- 18.2 Jerarquía de datos
- 18.3 Archivos y flujos
- 18.4 Las clases File y Directory
- 18.5 Creación de un archivo de texto de acceso secuencial
- 18.6 Lectura de datos desde un archivo de texto de acceso secuencial
- 18.7 Serialización
- 18.8 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos
- 18.9 Lectura y deserialización de datos de un archivo de texto de acceso secuencial
- 18.10 Conclusión

18.1 Introducción

Las variables y los arreglos ofrecen sólo un almacenamiento temporal de los datos; los datos se pierden cuando una variable local “queda fuera de alcance” o cuando el programa termina. En contraste, los archivos (y las bases de datos, que veremos en el capítulo 18) se utilizan para una retención a largo plazo de grandes cantidades de datos, incluso después de que termine el programa que creó los datos. A menudo, los datos que se mantienen en archivos se conocen como *datos persistentes*. Las computadoras almacenan archivos en *dispositivos de almacenamiento secundario*, como discos magnéticos, discos ópticos y cintas magnéticas. En este capítulo, explicaremos cómo crear, actualizar y procesar archivos de datos en programas en C#.

Empezaremos con una visión general de la jerarquía de datos, desde los bits hasta los archivos. Después veremos las generalidades acerca de las clases de procesamiento de archivos de la FCL. Luego presentaremos los ejemplos que demuestran cómo se puede determinar información acerca de los archivos y directorios en la computadora. El resto del capítulo muestra cómo escribir y leer hacia/desde archivos de texto elegibles para los humanos, y archivos binarios que almacenan objetos completos en formato binario.

18.2 Jerarquía de datos

Básicamente, todos los elementos de datos que procesan las computadoras se reducen a combinaciones de ceros y unos. Esto ocurre debido a que es simple y económico el construir dispositivos electrónicos que puedan suponer dos estados estables: un estado representa 0 y el otro, 1. Es increíble que las impresionantes funciones realizadas por las computadoras impliquen solamente las manipulaciones más fundamentales de 0s y 1s.

El elemento más pequeño de datos que soporta las computadoras es el *bit* (abreviatura de “*dígito binario*”: un dígito que puede asumir uno de dos valores). Cada elemento de datos, o bit, puede asumir ya sea el valor 0 o 1. Los circuitos de la computadora realizan varias manipulaciones simples de bits, cómo examinar el valor de un bit, establecer el valor de un bit e invertir un bit (de 1 a 0 o de 0 a 1).

Es muy difícil para los programadores trabajar con los datos en el formato de bits de bajo nivel. Es preferible programar con datos en formatos como *dígitos decimales* (es decir, 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9), *letras* (es decir, A-Z y a-z) y *símbolos especiales* (es decir, \$, @, %, &, *, (,), -, +, ", :, ?, / y muchos otros). Los dígitos, las letras y los símbolos especiales se conocen como *caracteres*. El *conjunto de caracteres* de una computadora es el conjunto de todos los caracteres que se utilizan para escribir programas y representar elementos de datos en una computadora específica. Debido a que las computadoras sólo pueden procesar 0s y 1s, cada carácter en el conjunto de caracteres de una computadora se representa como un patrón de 0s y 1s. Los *bytes* están compuestos de ocho bits. C# utiliza el *conjunto de caracteres Unicode*[®] (www.unicode.org), en el cual los caracteres están compuestos de 2 bytes. Los programadores crean programas y elementos de datos con los caracteres; las computadoras manipulan y procesan estos caracteres como patrones de bits.

Así como los caracteres están compuestos de bits, los campos están compuestos de caracteres. Un *campo* es un grupo de caracteres que transmiten cierto significado. Por ejemplo, un campo que consiste de letras mayúsculas y minúsculas puede representar el nombre de una persona.

Los elementos de datos procesados por las computadoras forman una *jerarquía de datos* (figura 18.1), en la cual los elementos de datos se hacen más grandes y complejos en estructura, a medida que progresamos de bits a caracteres, de caracteres a campos y de campos hasta llegar a conjuntos de datos más grandes.

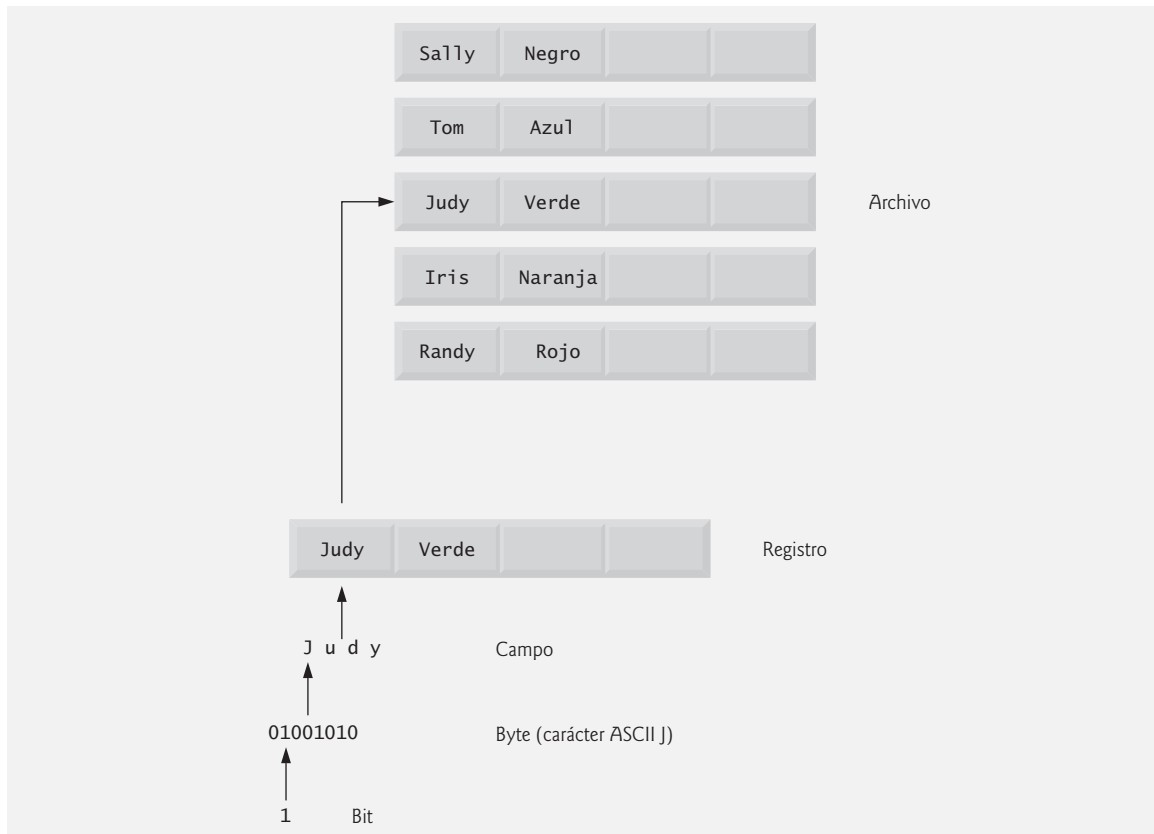


Figura 18.1 | Jerarquía de datos.

Por lo general, un *registro* (que puede representarse como una clase) está compuesto de varios campos relacionados. Por ejemplo, en un sistema de nóminas, un registro para un empleado específico podría incluir los siguientes campos:

1. Número de identificación del empleado.
2. Nombre.
3. Dirección.
4. Sueldo por hora.
5. Número de excepciones reclamadas.
6. Ingresos desde inicio de año a la fecha.
7. Monto de impuestos retenidos.

En el ejemplo anterior, cada campo está asociado con el mismo empleado. Un archivo es un grupo de registros relacionados.¹ Por lo general, el archivo de nóminas de una empresa contiene un registro para cada empleado. Un archivo de nóminas para una pequeña compañía podría contener sólo 22 registros, mientras que el de una compañía grande podría contener 100,000 registros. Es común para una compañía tener muchos archivos, algunos de ellos conteniendo millones, billones o incluso trillones de caracteres de información.

1. Por lo general, un archivo puede contener datos arbitrarios en formatos arbitrarios. En algunos sistemas operativos, un archivo se ve simplemente como una colección de bytes, y cualquier organización de los bytes en un archivo (como organizar los datos en registros) es una vista creada por el programador de aplicaciones.

Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse cuando menos un campo en cada registro como *clave de registro*, la cual identifica que un registro pertenece a una persona o entidad específica, y distingue a ese registro de todos los demás. Por ejemplo, en un registro de nóminas se utiliza, por lo general, el número de identificación de empleado como clave de registro.

Existen muchas formas de organizar los registros en un archivo. Una organización común se conoce como *archivo secuencial*, en el que por lo general los registros se almacenan en orden, mediante un campo que es la clave de registro. En un archivo de nóminas, es común que los registros se coloquen en orden, en base al número de identificación del empleado. El registro del primer empleado en el archivo contiene el número de identificación de empleado más pequeño, y los registros subsiguientes contienen números cada vez mayores.

La mayoría de las empresas usan muchos archivos distintos para almacenar datos. Por ejemplo, una compañía podría tener archivos de nóminas, archivos de cuentas por cobrar (listas del dinero que deben los clientes), archivos de cuentas por pagar (listas del dinero que se debe a los proveedores), archivos de inventarios (listas de información acerca de todos los artículos que maneja la empresa) y muchos otros tipos de archivos. A menudo, un grupo de archivos relacionados se almacena en una *base de datos*. A una colección de programas diseñados para crear y administrar bases de datos se le conoce como *sistema de administración de bases de datos (DBMS)*. En el capítulo 20 hablaremos sobre las bases de datos.

18.3 Archivos y flujos

C# considera a cada archivo como un *flujo* secuencial de bytes (figura 18.2). Cada archivo termina ya sea con un *marcador de fin de archivo*, o en un número específico de byte que se registra en una estructura de datos administrativa, mantenida por el sistema. Cuando se abre un archivo, se crea un objeto que se asocia con un flujo. Cuando se ejecuta un programa, el entorno en tiempo de ejecución crea tres objetos flujo, a los cuales se puede acceder mediante las propiedades **Console.Out**, **Console.In** y **Console.Error**, respectivamente. Estos objetos facilitan la comunicación entre un programa y un archivo o dispositivo específico. **Console.In** se refiere al *objeto flujo de entrada estándar*, el cual permite a un programa introducir datos desde el teclado. **Console.Out** se refiere al *objeto flujo de salida estándar*, el cual permite a un programa imprimir datos en la pantalla. **Console.Error** se refiere al *objeto flujo de error estándar*, el cual permite a un programa imprimir mensajes de error en la pantalla. Hemos estado usando **Console.Out** y **Console.In** en nuestras aplicaciones de consola; los métodos **Write** y **WriteLine** utilizan **Console.Out** para las operaciones de salida, y los métodos **Read** y **ReadLine** de **Console** utilizan **Console.In** para las operaciones de entrada.

Existen muchas clases de procesamiento de archivos en la FCL. El espacio de nombres **System.IO** incluye clases de flujos como **StreamReader** (para introducir texto desde un archivo), **StreamWriter** (para enviar texto a un archivo) y **FileStream** (para introducir y enviar texto desde/hacia un archivo). Estas clases de flujos heredan de las clases abstractas **TextReader**, **TextWriter** y **Stream**, respectivamente. En realidad, las propiedades **Console.In** y **Console.Out** son de tipo **TextReader** y **TextWriter**, respectivamente. El sistema crea objetos de las clases derivadas **TextReader** y **TextWriter** para inicializar las propiedades **Console.In** y **Console.Out** de **Console**.

La clase abstracta **Stream** ofrece la funcionalidad para representar flujos como bytes. Las clases **FileStream**, **MemoryStream** y **BufferedStream** (todas del espacio de nombres **System.IO**) heredan de la clase **Stream**. La clase **FileStream** puede utilizarse para escribir y leer datos hacia y desde, archivos respectivamente. La clase **MemoryStream** habilita la transferencia de datos directamente desde/hacia la memoria; esto es mucho más rápido que leer desde, y escribir hacia, dispositivos externos. La clase **BufferedStream** la técnica de *uso de un búfer* para transferir datos desde/hacia un flujo. El uso de búfer es una técnica para mejorar el rendimiento de E/S, en donde cada operación de salida se dirige a una región en memoria, llamada *búfer*, que es lo bastante grande como para almacenar los datos provenientes de muchas operaciones de salida. Después, la verdadera transferencia hacia el dispositivo de salida se realiza en una sola *operación de salida física* extensa, cada vez que se llena el búfer. Las

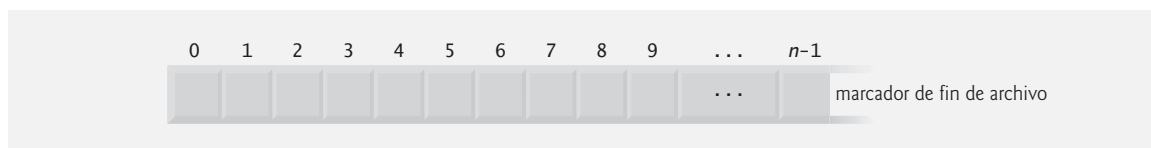


Figura 18.2 | La manera en que C# ve a un archivo de n bytes.

operaciones de salida dirigidas al búfer de salida en memoria se conocen comúnmente como *operaciones lógicas de salida*. Los búferes también se pueden utilizar para agilizar las operaciones de entrada, ya que al principio se leen más datos de los que se requieren y se colocan en un búfer, de manera que las operaciones subsiguientes de lectura obtengan los datos de memoria, en vez de obtenerlos de un dispositivo externo.

En este capítulo usaremos clases de flujo claves para implementar programas de procesamiento de archivos que crean y manipulan archivos de acceso secuencial. En el capítulo 23, Redes: sockets basados en flujos y datagramas, usaremos las clases de flujos para implementar aplicaciones de red.

18.4 Las clases File y Directory

La información se almacena en archivos, que se organizan en directorios. Las clases `File` y `Directory` permiten a los programas manipular archivos y directorios en el disco. La clase ***File*** puede determinar información acerca de los archivos y puede usarse para abrir archivos en modo de lectura o de escritura. En las secciones subsiguientes hablaremos sobre las técnicas para escribir hacia, y leer desde, archivos.

La figura 18.3 lista varios métodos de la clase `static File` para manipular y determinar información acerca de los archivos. En la figura 18.5 demostramos varios de estos métodos.

La clase ***Directory*** cuenta con herramientas para manipular directorios. La figura 18.4 lista algunos de los métodos `static` de la clase `Directory` para manipulación de directorios. La figura 18.5 demuestra también varios de estos métodos. El objeto ***DirectoryInfo*** devuelto por el método ***CreateDirectory*** contiene información acerca de un directorio. Gran parte de la información contenida en la clase `DirectoryInfo` también puede utilizarse a través de los métodos de la clase `Directory`.

Demostración de las clases File y Directory

La clase `PruebaArchivosForm` (figura 18.5) utiliza los métodos de `File` y `Directory` para acceder a la información de archivos y directorios. Este formulario contiene el control `entradaTextBox`, en el que el usuario

Método <code>static</code>	Descripción
<code>AppendText</code>	Devuelve un objeto <code>StreamWriter</code> que adjunta texto a un archivo existente, o crea un archivo si no existe.
<code>Copy</code>	Copia un archivo a un archivo nuevo.
<code>Create</code>	Crea un archivo y devuelve su objeto <code>FileStream</code> asociado.
<code>CreateText</code>	Crea un archivo de texto y devuelve su objeto <code>StreamWriter</code> asociado.
<code>Delete</code>	Elimina el archivo especificado.
<code>Exists</code>	Devuelve <code>true</code> si existe el archivo especificado, y <code>false</code> en caso contrario.
<code>GetCreationTime</code>	Devuelve un objeto <code>DateTime</code> que representa la fecha y hora en la que se creó el archivo.
<code>GetLastAccessTime</code>	Devuelve un objeto <code>DateTime</code> que representa la fecha y hora del último acceso al archivo.
<code>GetLastWriteTime</code>	Devuelve un objeto <code>DateTime</code> que representa la fecha y hora de la última modificación del archivo.
<code>Move</code>	Mueve el archivo especificado a una ubicación especificada.
<code>Open</code>	Devuelve un objeto <code>FileStream</code> asociado con el archivo especificado y equipado con los permisos de lectura/escritura especificados.
<code>OpenRead</code>	Devuelve un objeto <code>FileStream</code> de sólo lectura, asociado con el archivo especificado.
<code>OpenText</code>	Devuelve un objeto <code>StreamReader</code> asociado con el archivo especificado.
<code>OpenWrite</code>	Devuelve un objeto <code>FileStream</code> de lectura/escritura, asociado con el archivo especificado.

Figura 18.3 | Métodos `static` de la clase `File` (lista parcial).

Método static	Descripción
CreateDirectory	Crea un directorio y devuelve su objeto DirectoryInfo asociado.
Delete	Elimina el directorio especificado.
Exists	Devuelve true si existe el directorio especificado y false en caso contrario.
GetDirectories	Devuelve un arreglo string que contiene los nombres de los subdirectorios en el directorio especificado.
GetFiles	Devuelve un arreglo string que contiene los nombres de los archivos en el directorio especificado.
GetCreationTime	Devuelve un objeto DateTime que representa la fecha y hora de creación del directorio.
GetLastAccessTime	Devuelve un objeto DateTime que representa la fecha y hora del último acceso al directorio.
GetLastWriteTime	Devuelve un objeto DateTime que representa la fecha y hora en que se escribieron los últimos elementos en el directorio.
Move	Mueve el directorio especificado a una ubicación especificada.

Figura 18.4 | Métodos static de la clase Directory.

introduce el nombre de un archivo o directorio. Para cada tecla que oprima el usuario mientras escribe en el control TextBox, el programa llama al manejador de eventos entradaTextBox_KeyDown (líneas 17-74). Si el usuario oprime *Intro* (línea 20), este método muestra el contenido del archivo o del directorio, dependiendo del texto introducido por el usuario. (Si el usuario no oprime *Intro*, este método regresa sin mostrar ningún contenido.) La línea 28 utiliza el método Exists de File para determinar si el texto especificado por el usuario es el nombre de algún archivo existente. Si es así, la línea 32 invoca al método private ObtenerInformacion (líneas 77-97), el cual llama a los métodos GetCreationTime (línea 86), GetLastWriteTime (línea 90) y GetLastAccessTime (línea 94) de File para acceder a la información del archivo. Cuando el método ObtenerInformacion regresa, la línea 38 crea una instancia de un objeto StreamReader para leer texto del archivo. El constructor de StreamReader recibe como argumento un objeto string que contiene el nombre del archivo que se debe abrir. La línea 39 llama al método ReadToEnd de StreamReader para leer todo el contenido del archivo como un objeto string, y después adjunta ese objeto string al contenido del control salidaTextBox.

```

1 // Fig 18.5: PruebaArchivosForm.cs
2 // Uso de las clases File y Directory.
3 using System;
4 using System.Windows.Forms;
5 using System.IO;
6
7 // muestra el contenido de los archivos y los directorios
8 public partial class PruebaArchivosForm : Form
9 {
10     // constructor sin parámetros
11     public PruebaArchivosForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // se invoca cuando el usuario oprime una tecla
17     private void entradaTextBox_KeyDown( object sender, EventArgs e )

```

Figura 18.5 | Prueba de las clases File y Directory. (Parte I de 3).

```

18 {
19     // determina si el usuario oprimió la tecla Intro
20     if ( e.KeyCode == Keys.Enter )
21     {
22         string nombreArchivo; // nombre del archivo o directorio
23
24         // obtiene el archivo o directorio especificado por el usuario
25         nombreArchivo = entradaTextBox.Text;
26
27         // determina si nombreArchivo es un archivo
28         if ( File.Exists( nombreArchivo ) )
29         {
30             // obtiene la fecha de creación del archivo,
31             // su fecha de modificación, etc.
32             salidaTextBox.Text = ObtenerInformacion( nombreArchivo );
33
34             // muestra el contenido del archivo a través de StreamReader
35             try
36             {
37                 // obtiene lector y contenido del archivo
38                 StreamReader flujo = new StreamReader( nombreArchivo );
39                 salidaTextBox.Text += flujo.ReadToEnd();
40             } // fin de try
41             // maneja excepción si StreamReader no está disponible
42             catch ( IOException )
43             {
44                 MessageBox.Show( "Error al leer del archivo", "Error de archivo",
45                                 MessageBoxButtons.OK, MessageBoxIcon.Error );
46             } // fin de catch
47         } // fin de if
48         // determina si nombreArchivo es un directorio
49         else if ( Directory.Exists( nombreArchivo ) )
50         {
51             string[] listaDirectorios; // arreglo para los directorios
52
53             // obtiene la fecha de creación del directorio,
54             // su fecha de modificación, etc.
55             salidaTextBox.Text = ObtenerInformacion( nombreArchivo );
56
57             // obtiene la lista de archivos/directorios del directorio especificado
58             listaDirectorios = Directory.GetDirectories( nombreArchivo );
59
60             salidaTextBox.Text += "\r\n\r\nContenido del directorio:\r\n";
61
62             // imprime en pantalla el contenido de listaDirectorios
63             for ( int i = 0; i < listaDirectorios.Length; i++ )
64                 salidaTextBox.Text += listaDirectorios[ i ] + "\r\n";
65         } // fin de else if
66         else
67         {
68             // notifica al usuario que no existe el directorio o archivo
69             MessageBox.Show( entradaTextBox.Text +
70                             " no existe", "Error de archivo",
71                             MessageBoxButtons.OK, MessageBoxIcon.Error );
72         } // fin de else
73     } // fin de if
74 } // fin del método entradaTextBox_KeyDown
75
76 // obtiene información sobre el archivo o directorio

```

Figura 18.5 | Prueba de las clases File y Directory. (Parte 2 de 3).

```

77 private string ObtenerInformacion( string nombreArchivo )
78 {
79     string informacion;
80
81     // imprime mensaje indicando que existe el archivo o directorio
82     informacion = nombreArchivo + " existe\r\n\r\n";
83
84     // imprime en pantalla la fecha y hora de creación del archivo o directorio
85     informacion += "Creación: " +
86         File.GetCreationTime( nombreArchivo ) + "\r\n";
87
88     // imprime en pantalla la fecha de la última modificación del archivo o directorio
89     informacion += "Última modificación: " +
90         File.GetLastWriteTime( nombreArchivo ) + "\r\n";
91
92     // imprime en pantalla la fecha y hora del último acceso al archivo o directorio
93     informacion += "Último acceso: " +
94         File.GetLastAccessTime( nombreArchivo ) + "\r\n" + "\r\n";
95
96     return informacion;
97 } // fin del método ObtenerInformacion
98 } // fin de la clase PruebaArchivosForm

```

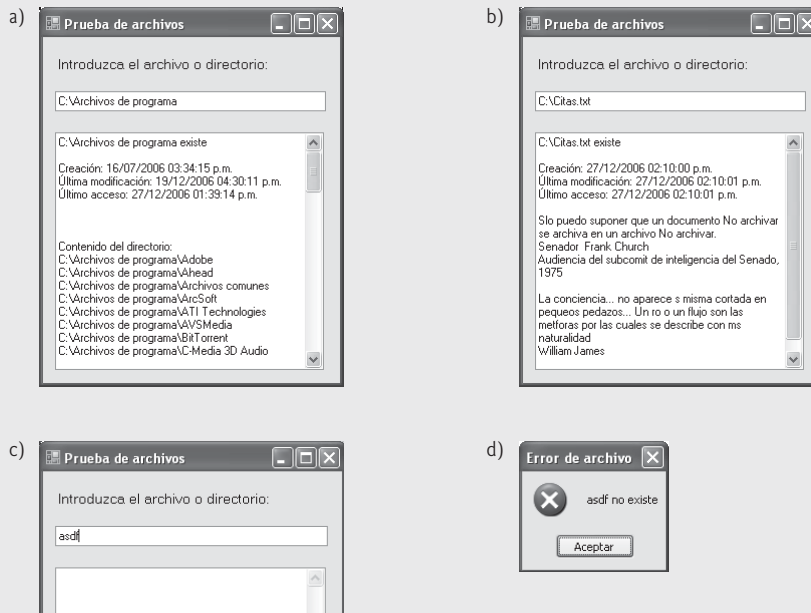


Figura 18.5 | Prueba de las clases File y Directory. (Parte 3 de 3).

Si la línea 28 determina que el texto especificado por el usuario no es un archivo, la línea 49 determina si es un directorio, usando el método **Exists** de **Directory**. Si el usuario especificó un directorio existente, la línea 55 invoca al método **ObtenerInformacion** para acceder a la información del directorio. La línea 58 llama al método **GetDirectories** de **Directory** para obtener un arreglo **string** que contiene los nombres de los subdirectorios en el directorio especificado. Las líneas 63-64 muestran a cada elemento en el arreglo **string**. Observe que, si la línea 49 determina que el texto especificado por el usuario no es un nombre de directorio, las líneas 69-71 notifican al usuario (a través de un cuadro **MessageBox**) que el nombre que introdujo no existe como archivo o directorio.

Búsqueda de directorios con expresiones regulares

Ahora consideraremos otro ejemplo que utiliza las herramientas de manipulación de archivos y directorios de C#. La clase BuscarArchivoForm (figura 18.6) utiliza las clases File y Directory, y las herramientas de expresiones regulares para reportar el número de archivos de cada tipo que exista en la ruta del directorio especificado. El programa también sirve como herramienta de “limpieza”; cuando encuentra un archivo que tiene la extensión .bak (es decir, un archivo de respaldo), el programa muestra un cuadro MessageBox, pidiendo al usuario si desea eliminar el archivo y después responde en forma apropiada a la entrada del usuario.

Cuando el usuario oprime *Intro* o hace clic en el botón *Buscar directorio*, el programa invoca al método buscarButton_Click (líneas 34-71), el cual busca en forma recursiva a través de la ruta del directorio que proporcionó el usuario. Si el usuario introduce texto en el control TextBox, la línea 40 llama al método Exists de Directory para determinar si ese texto es la ruta y el nombre de un directorio válido. Si no es así, las líneas 51-52 notifican al usuario sobre el error.

Si el usuario especifica un directorio válido, la línea 60 pasa el nombre del directorio como argumento para el método private BuscarDirectorio (líneas 74-153). Este método localiza los archivos que coinciden con la expresión regular definida en las líneas 82-83. Esta expresión regular coincide con cualquier secuencia de números o letras, seguida de un punto y de una o más letras. Observe la subcadena de formato (?<extension>\w+) en el argumento para el constructor Regex (línea 83). Esto indica que la parte de la cadena que coincide con \w+ (es decir, la extensión de archivo que aparece después de un punto en el nombre del archivo) debe colocarse en la

```

1  // Fig 18.6: ArchivoBuscarForm.cs
2  // Uso de expresiones regulares para determinar los tipos de archivos.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6  using System.Text.RegularExpressions;
7  using System.Collections.Specialized;
8
9  // usa expresiones regulares para determinar los tipos de archivos
10 public partial class BuscarArchivoForm : Form
11 {
12     string directorioActual = Directory.GetCurrentDirectory();
13     string[] listaDirectorios; // subdirectorios
14     string[] arregloArchivos;
15
16     // almacena las extensiones encontradas y el número encontrado
17     NameValueCollection encontrados = new NameValueCollection();
18
19     // constructor sin parámetros
20     public BuscarArchivoForm()
21     {
22         InitializeComponent();
23     } // fin del constructor
24
25     // se invoca cuando el usuario escribe en el cuadro de texto
26     private void entradaTextBox_KeyDown( object sender, KeyEventArgs e )
27     {
28         // determina si el usuario oprimió Intro
29         if ( e.KeyCode == Keys.Enter )
30             buscarButton_Click( sender, e );
31     } // fin del método entradaTextBox_KeyDown
32
33     // se invoca cuando el usuario hace clic en el botón "Buscar directorio"
34     private void buscarButton_Click( object sender, EventArgs e )
35     {
36         // comprueba la entrada del usuario; el valor predeterminado es el directorio actual

```

Figura 18.6 | Expresión regular utilizada para determinar los tipos de archivos. (Parte I de 4).

```

37     if ( entradaTextBox.Text != "" )
38     {
39         // verifica que la entrada del usuario sea un nombre de directorio válido
40         if ( Directory.Exists( entradaTextBox.Text ) )
41         {
42             directorioActual = entradaTextBox.Text;
43
44             // restablece el cuadro de texto de entrada y actualiza la pantalla
45             directorioLabel.Text = "Directorio actual:" +
46                 "\r\n" + directorioActual;
47         } // fin de if
48         else
49         {
50             // muestra error si el usuario no especifica un directorio válido
51             MessageBox.Show( "Directorio inválido", "Error",
52                 MessageBoxButtons.OK, MessageBoxIcon.Error );
53         } // fin de else
54     } // fin de if
55
56     // borra el contenido de los cuadros de texto
57     entradaTextBox.Text = "";
58     salidaTextBox.Text = "";
59
60     BuscarDirectorio( directorioActual ); // busca en el directorio
61
62     // sintetiza e imprime los resultados en pantalla
63     foreach ( string actual in encontrados )
64     {
65         salidaTextBox.Text += "Se encontraron " +
66             encontrados[ actual ] + " archivos " + actual + ".\r\n";
67     } // fin de foreach
68
69     // borra la salida para una nueva búsqueda
70     encontrados.Clear();
71 } // fin del método buscarButton_Click
72
73 // busca en el directorio usando expresión regular
74 private void BuscarDirectorio( string directorioActual )
75 {
76     // para el nombre de archivo sin ruta de directorio
77     try
78     {
79         string nombreArchivo = "";
80
81         // expresión regular para las extensiones que coinciden con el patrón
82         Regex expresionRegular = new Regex(
83             @"[a-zA-Z0-9]+\.(?<extension>\w+)" );
84
85         // almacena resultado de coincidencias de la expresión regular
86         Match resultadoCoincidencias;
87
88         string extensionArchivo; // almacena las extensiones de archivo
89
90         // número de archivos con la extensión dada en el directorio
91         int cuentaExtension;
92
93         // obtiene los directorios
94         listaDirectorios = Directory.GetDirectories( directorioActual );
95

```

Figura 18.6 | Expresión regular utilizada para determinar los tipos de archivos. (Parte 2 de 4).

```

96 // obtiene la lista de archivos en el directorio actual
97 arregloArchivos = Directory.GetFiles( directorioActual );
98
99 // itera a través de la lista de archivos
100 foreach ( string miArchivo in arregloArchivos )
101 {
102     // elimina la ruta del directorio del nombre del archivo
103     nombreArchivo = miArchivo.Substring( miArchivo.LastIndexOf( @"\" ) + 1 );
104
105     // obtiene resultado de la búsqueda de la expresión regular
106     resultadoCoincidencias = expresionRegular.Match( nombreArchivo );
107
108     // verifica que haya coincidencia
109     if ( resultadoCoincidencias.Success )
110         extensionArchivo = resultadoCoincidencias.Result( "${extension}" );
111     else
112         extensionArchivo = "[sin extensión]";
113
114     // almacena el valor del contenedor
115     if ( encontrados[ extensionArchivo ] == null )
116         encontrados.Add( extensionArchivo, "1" );
117     else
118     {
119         cuentaExtension = Int32.Parse( encontrados[ extensionArchivo ] ) + 1;
120         encontrados[ extensionArchivo ] = cuentaExtension.ToString();
121     } // fin de else
122
123     // busca archivos de respaldo( .bak )
124     if ( extensionArchivo == "bak" )
125     {
126         // pregunta al usuario si desea eliminar el archivo ( .bak )
127         DialogResult resultado =
128             MessageBox.Show( "Se encontró archivo de respaldo " +
129                             nombreArchivo + ". ¿Desea eliminarlo?", "Eliminar archivo de respaldo",
130                             MessageBoxButtons.YesNo, MessageBoxIcon.Question );
131
132         // elimina el archivo si el usuario hizo clic en 'sí'
133         if ( resultado == DialogResult.Yes )
134         {
135             File.Delete( miArchivo );
136             cuentaExtension = Int32.Parse( encontrados[ "bak" ] ) - 1;
137             encontrados[ "bak" ] = cuentaExtension.ToString();
138         } // fin de if
139     } // fin de if
140 } // fin de foreach
141
142 // llamada recursiva para buscar archivos en subdirectorio
143 foreach ( string miDirectorio in listaDirectorios )
144     BuscarDirectorio( miDirectorio );
145 } // fin de try
146 // maneja excepción si los archivos tienen acceso no autorizado
147 catch ( UnauthorizedAccessException )
148 {
149     MessageBox.Show( "Tal vez algunos archivos no puedan verse" +
150                     " debido a la configuración de los permisos", "Advertencia",
151                     MessageBoxButtons.OK, MessageBoxIcon.Information );
152 } // fin de catch
153 } // fin del método BuscarDirectorio
154 } // fin de la clase FileSearchForm

```

Figura 18.6 | Expresión regular utilizada para determinar los tipos de archivos. (Parte 3 de 4).

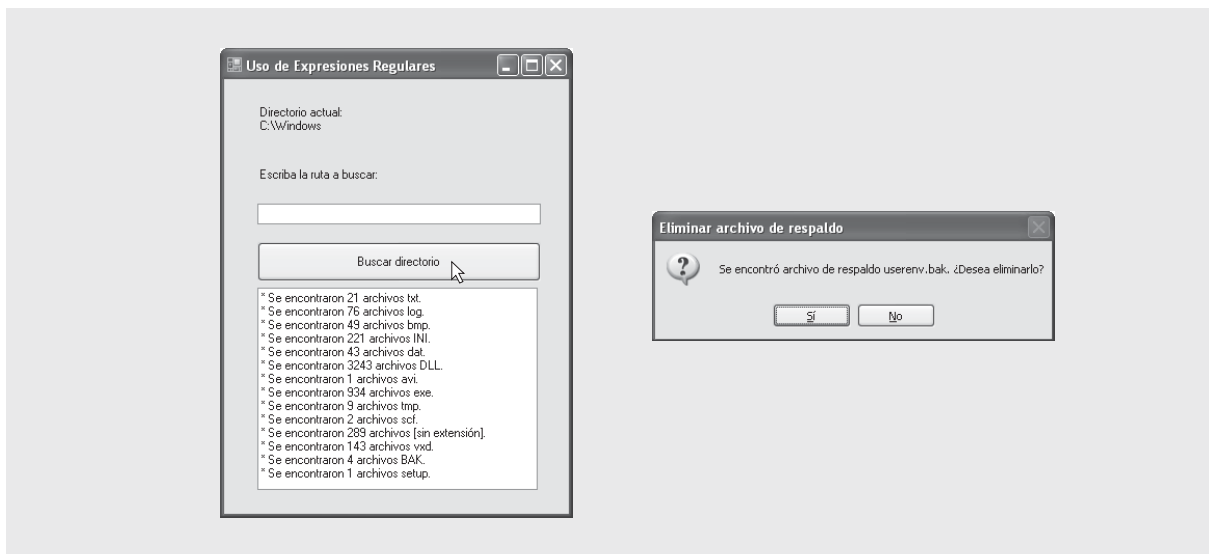


Figura 18.6 | Expresión regular utilizada para determinar los tipos de archivos. (Parte 4 de 4).

variable de la expresión regular llamada `extension`. El valor de esta variable se extrae posteriormente del objeto `Match` llamado `resultadoCoincidencias` para obtener la extensión de archivo, de manera que podamos sintetizar los tipos de archivos en el directorio.

La línea 94 llama al método `GetDirectorios` de `Directory` para recuperar los nombres de todos los subdirectorios que pertenecen al directorio actual. La línea 97 llama al método `GetFiles` de `Directory` para almacenar en el arreglo `string` `arregloArchivos` los nombres de los archivos en el directorio actual. El ciclo `foreach` en las líneas 100-140 busca todos los archivos con la extensión `.bak`. El ciclo en las líneas 143-144 llama después al método `BuscarDirectorio` en forma recursiva (línea 144) para cada subdirectorio dentro del directorio actual. La línea 103 elimina la ruta del directorio, por lo que el programa puede evaluar sólo el nombre del archivo al utilizar la expresión regular. La línea 106 utiliza el método `Match` de `Regex` para relacionar la expresión regular con el nombre de archivo y después asigna el resultado al objeto `Match` `resultadoCoincidencias`. Si hubo coincidencia, la línea 110 usa el método `Result` de `Match` para asignar a `extensionArchivo` el valor de la variable de expresión regular `extension` del objeto `resultadoCoincidencias`. Si no hubo coincidencia, la línea 112 establece `extensionArchivo` a “[sin extensión]”.

La clase `ArchivoBuscarForm` utiliza una instancia de la clase `NameValueCollection` (declarada en la línea 17) para almacenar cada tipo de extensión de nombre de archivo y el número de archivos para cada tipo. Un objeto `NameValueCollection` (espacio de nombres `System.Collections.Specialized`) contiene una colección de pares clave-valor de objetos `string`, y proporciona el método `Add` para agregar un par clave-valor a la colección. El indexador para esta clase puede indexar de acuerdo con el orden en que se agregaron los elementos, o de acuerdo con las claves. La línea 115 utiliza el objeto `NameValueCollection` `encontrado` para determinar si ésta es la primera ocurrencia de la extensión de archivo (la expresión devuelve `null` si la colección no contiene un par clave-valor para la extensión de archivo especificada en `extensionArchivo`). Si ésta es la primera ocurrencia, la línea 116 agrega esa extensión a `encontrado` como una clave con el valor 1. En caso contrario, la línea 119 incrementa el valor asociado con la extensión en `encontrado` para indicar otra ocurrencia de esa extensión de archivo, y la línea 120 asigna el nuevo valor al par clave-valor.

La línea 124 determina si `extensionArchivo` es igual a “bak” (es decir, si el archivo es de respaldo). De ser así, las líneas 127-130 preguntan al usuario si desea eliminar el archivo; si la respuesta es **Sí** (línea 133), las líneas 135-137 eliminan el archivo y decrementan el valor para el tipo de archivo “bak” en `encontrado`.

Las líneas 143-144 llaman al método `BuscarDirectorio` para cada subdirectorio. Mediante el uso de la recursividad, nos aseguramos que el programa lleve a cabo la misma lógica para buscar archivos `.bak` en cada subdirectorio. Después de revisar si hay archivos `.bak` en cada subdirectorio, el método `BuscarDirectorio` se completa y las líneas 64-67 muestran los resultados.

18.5 Creación de un archivo de texto de acceso secuencial

C# no impone una estructura en los archivos. Por ende, el concepto de un “registro” no existe en los archivos de C#. Esto significa que usted debe estructurar los archivos para cumplir con los requerimientos de sus aplicaciones. En los siguientes ejemplos, usaremos texto y caracteres especiales para organizar nuestro propio concepto de un “registro”.

Clase BancoUIForm

Los siguientes ejemplos demuestran el procesamiento de archivos en una aplicación de mantenimiento de cuentas bancarias. Estos programas tienen interfaces de usuario similares, por lo que creamos la clase reutilizable BancoUIForm (figura 18.7 del diseñador de Windows Forms) para encapsular la GUI de una clase base (vea la captura de pantalla de la figura 18.7). La clase BancoUIForm contiene cuatro controles Label y cuatro controles TextBox. Los métodos BorrarControlesTextBox (líneas 26-40), EstablecerValoresControlesTextBox (líneas 43-61) y ObtenerValoresControlesTextBox (líneas 64-75) borran, establecen los valores de, y obtienen los valores del texto en los controles TextBox, respectivamente.

```

1  // Fig. 18.7: BancoUIForm.cs
2  // Un formulario Windows Form reutilizable para los ejemplos en este capítulo.
3  using System;
4  using System.Windows.Forms;
5
6  public partial class BancoUIForm : Form
7  {
8      protected int CuentaTextBox = 4; // número de controles TextBox en el formulario
9
10     // las constantes de la enumeración especifican los índices de los controles TextBox
11     public enum IndicesTextBox
12     {
13         CUENTA,
14         NOMBRE,
15         APELLIDO,
16         SALDO
17     } // fin de enum
18
19     // constructor sin parámetros
20     public BancoUIForm()
21     {
22         InitializeComponent();
23     } // fin del constructor
24
25     // borra todos los controles TextBox
26     public void BorrarControlesTextBox()
27     {
28         // itera a través de cada Control en el formulario
29         for ( int i = 0; i < Controls.Count; i++ )
30         {
31             Control miControl = Controls[ i ]; // obtiene el control
32
33             // determina si el Control es TextBox
34             if ( miControl is TextBox )
35             {
36                 // borra la propiedad Text ( la establece a una cadena vacía )
37                 miControl.Text = "";
38             } // fin de if
39         } // fin de for
40     } // fin del método BorrarControlesTextBox

```

Figura 18.7 | Clase base para las GUIs en nuestras aplicaciones de procesamiento de archivos. (Parte 1 de 2).


```

41
42 // establece los valores de los controles TextBox con el arreglo string valores
43 public void EstablecerValoresControlesTextBox( string[] valores )
44 {
45     // determina si el arreglo string tiene la longitud correcta
46     if ( valores.Length != CuentaTextBox )
47     {
48         // lanza excepción si no tiene la longitud correcta
49         throw( new ArgumentException( "Debe haber " +
50             ( CuentaTextBox + 1 ) + " objetos string en el arreglo" ) );
51     } // fin de if
52     // establece el arreglo valores si el arreglo tiene la longitud correcta
53     else
54     {
55         // establece el arreglo valores con los valores de los controles TextBox
56         cuentaTextBox.Text = valores[ ( int ) IndicesTextBox.CUENTA ];
57         primerNombreTextBox.Text = valores[ ( int ) IndicesTextBox.NOMBRE ];
58         apellidoPaternoTextBox.Text = valores[ ( int ) IndicesTextBox.APELLIDO ];
59         saldoTextBox.Text = valores[ ( int ) IndicesTextBox.SALDO ];
60     } // fin de else
61 } // fin del método EstablecerValoresControlesTextBox
62
63 // devuelve los valores de los controles TextBox como un arreglo string
64 public string[] ObtenerValoresControlesTextBox()
65 {
66     string[] valores = new string[ CuentaTextBox ];
67
68     // copia los campos de los controles TextBox al arreglo string
69     valores[ ( int ) IndicesTextBox.CUENTA ] = cuentaTextBox.Text;
70     valores[ ( int ) IndicesTextBox.NOMBRE ] = primerNombreTextBox.Text;
71     valores[ ( int ) IndicesTextBox.APELLIDO ] = apellidoPaternoTextBox.Text;
72     valores[ ( int ) IndicesTextBox.SALDO ] = saldoTextBox.Text;
73
74     return valores;
75 } // fin del método ObtenerValoresControlesTextBox
76 } // fin de la clase BankUIForm

```

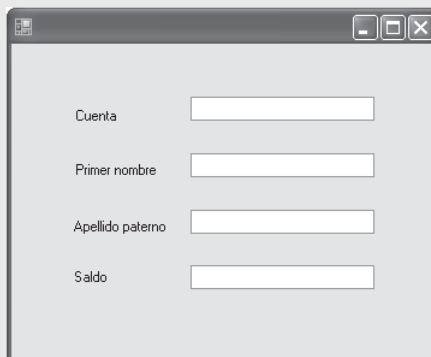


Figura 18.7 | Clase base para las GUIs en nuestras aplicaciones de procesamiento de archivos. (Parte 2 de 2).

Para reutilizar la clase BancoUIForm, debe compilar la GUI en un archivo DLL, creando un proyecto de tipo **Biblioteca de controles de Windows** (a nuestro proyecto lo nombramos BibliotecaBanco). Esta biblioteca se proporciona con el código para este capítulo. Tal vez tenga que cambiar las referencias a esta biblioteca en nuestros ejemplos cuando los copie a su sistema, ya que es muy probable que la biblioteca resida en una ubicación distinta en su sistema.

Clase Registro

La figura 18.8 contiene la clase Registro que utilizan las figuras 18.9, 18.11 y 18.12 para mantener la información en cada registro que se escribe hacia, o se lee desde, un archivo. Esta clase también pertenece al archivo DLL de la BibliotecaBanco, por lo que se encuentra en el mismo proyecto que la clase BancoUIForm.

La clase Registro contiene las variables de instancia `private cuenta`, `primerNombre`, `apellidoPaterno` y `saldo` (líneas 9-12), que en conjunto representan toda la información para un registro. El constructor sin parámetros (líneas 15-17) establece todos estos miembros, mediante una llamada al constructor de cuatro argumentos, con 0 para el número de cuenta, cadenas vacías (""), para el primer nombre y el apellido paterno, y 0.0M para el saldo. El constructor de cuatro argumentos (líneas 20-27) establece estos miembros a los valores de los parámetros especificados. La clase Registro también proporciona las propiedades Cuenta (líneas 30-40), PrimerNombre (líneas 43-53), ApellidoPaterno (líneas 56-66) y Saldo (líneas 69-79) para acceder al número de cuenta, primer nombre, apellido paterno y saldo de cada registro, respectivamente.

```

1  // Fig. 18.8: Registro.cs
2  // Clase serializable que representa a un registro de datos.
3  using System;
4  using System.Collections.Generic;
5  using System.Text;
6
7  public class Registro
8  {
9      private int cuenta;
10     private string primerNombre;
11     private string apellidoPaterno;
12     private decimal saldo;
13
14     // el constructor sin parámetros establece los miembros a los valores predeterminados
15     public Registro() : this( 0, "", "", 0.0M )
16     {
17     } // fin del constructor
18
19     // el constructor sobrecargado, establece los miembros a los valores de los parámetros
20     public Registro( int valorCuenta, string valorPrimerNombre,
21         string valorApellidoPaterno, decimal valorSaldo )
22     {
23         Cuenta = valorCuenta;
24         PrimerNombre = valorPrimerNombre;
25         ApellidoPaterno = valorApellidoPaterno;
26         Saldo = valorSaldo;
27     } // fin del constructor
28
29     // propiedad que obtiene y establece Cuenta
30     public int Cuenta
31     {
32         get
33         {
34             return cuenta;
35         } // fin de get
36         set
37         {
38             cuenta = value;
39         } // fin de set
40     } // fin de la propiedad Cuenta
41
42     // propiedad que obtiene y establece PrimerNombre
43     public string PrimerNombre

```

Figura 18.8 | Registro para las aplicaciones de procesamiento de archivos de acceso secuencial. (Parte 1 de 2).

```

44     {
45         get
46         {
47             return primerNombre;
48         } // fin de get
49         set
50         {
51             primerNombre = value;
52         } // fin de set
53     } // fin de la propiedad PrimerNombre
54
55     // propiedad que obtiene y establece ApellidoPaterno
56     public string ApellidoPaterno
57     {
58         get
59         {
60             return apellidoPaterno;
61         } // fin de get
62         set
63         {
64             apellidoPaterno = value;
65         } // fin de set
66     } // fin de la propiedad ApellidoPaterno
67
68     // propiedad que obtiene y establece Saldo
69     public decimal Saldo
70     {
71         get
72         {
73             return saldo;
74         } // fin de get
75         set
76         {
77             saldo = value;
78         } // fin de set
79     } // fin de la propiedad Saldo
80 } // fin de la clase Registro

```

Figura 18.8 | Registro para las aplicaciones de procesamiento de archivos de acceso secuencial. (Parte 2 de 2).

Uso de un flujo de caracteres para crear un archivo de salida

La clase `CrearArchivoForm` (figura 18.9) utiliza instancias de la clase `Registro` para crear un archivo de acceso secuencial que podría usarse en un sistema de cuentas por cobrar (es decir, un programa que organice los datos en relación con el dinero que deben los clientes de crédito a una compañía. Para cada cliente, el programa obtiene un número de cuenta y el primer nombre, apellido paterno y saldo del cliente (es decir, el monto de dinero que el cliente debe a la compañía por los bienes y servicios que recibió previamente). Los datos obtenidos para cada cliente constituyen un registro para ese cliente. En esta aplicación, el número de cuenta se utiliza como la clave del registro; los archivos se crean y se mantienen en orden por número de cuenta. Este programa asume que el usuario introduce los registros en orden por número de cuenta. No obstante, un sistema de cuentas por cobrar completo proporcionaría una herramienta para ordenar, para que el usuario pudiera introducir los registros en cualquier orden.

La clase `CrearArchivoForm` crea o abre un archivo (dependiendo de que exista uno) y después permite al usuario escribir registros en ese archivo. La directiva `using` en la línea 6 nos permite utilizar las clases del espacio de nombres `BibliotecaBanco`; este espacio de nombres contiene la clase `BancoUIForm`, y la clase `CrearArchivoForm` hereda de esta clase (línea 8). La GUI de la clase `CrearArchivoForm` agrega funcionalidad a la GUI de la clase `BancoUIForm` a través de los botones **Guardar como**, **Entrar** y **Salir**.

Cuando el usuario hace clic en el botón **Guardar como**, el programa invoca al manejador de eventos `guardar_Click` (líneas 20-63). La línea 23 instancia un objeto de la clase `SaveFileDialog` (espacio de nombres

System.Windows.Forms). Los objetos de esta clase se utilizan para seleccionar archivos (vea la segunda pantalla en la figura 18.9). La línea 24 llama al método ShowDialog de la clase SaveFileDialog para mostrar el cuadro de diálogo. Al aparecer, un cuadro de diálogo SaveFileDialog evita que el usuario interactúe con cualquier otra ventana en el programa hasta que lo cierre, haciendo clic ya sea en **Guardar** o en **Cancelar**. Los cuadros de diálogo que se comportan de esta manera se conocen como *cuadros de diálogo modales*. El usuario selecciona la unidad, directorio y nombre de archivo apropiados, después hace clic en **Guardar**. El método *ShowDialog* devuelve un objeto DialogResult, el cual especifica cuál fue el botón (**Guardar** o **Cancelar**) en el que el usuario hizo clic para cerrar el cuadro de diálogo. Esto se asigna a la variable DialogResult llamada resultado (línea 24). La línea 30 evalúa si el usuario hizo clic en **Cancelar** mediante la comparación de este valor con DialogResult.Cancel. Si los valores son iguales, el método guardarButton_Click regresa (línea 31). En caso contrario, la línea 33 utiliza la propiedad FileName de SaveFileDialog para obtener el archivo seleccionado por el usuario.

```

1  // Fig. 18.9: CrearArchivoForm.cs
2  // Creación de un archivo de acceso secuencial.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6  using BibliotecaBanco;
7
8  public partial class CrearArchivoForm : BancoUIForm
9  {
10     private StreamWriter archivoWriter; // escribe datos en el archivo de texto
11     private FileStream salida; // mantiene la conexión con el archivo
12
13     // constructor sin parámetros
14     public CrearArchivoForm()
15     {
16         InitializeComponent();
17     } // fin del constructor
18
19     // manejador de eventos para el botón Guardar
20     private void guardarButton_Click( object sender, EventArgs e )
21     {
22         // crea un cuadro de diálogo que permite al usuario guardar el archivo
23         SaveFileDialog selectorArchivo = new SaveFileDialog();
24         DialogResult resultado = selectorArchivo.ShowDialog();
25         string nombreArchivo; // nombre del archivo en el que se van a guardar los datos
26
27         selectorArchivo.CheckFileExists = false; // permite al usuario crear el archivo
28
29         // sale del manejador de eventos si el usuario hace clic en "Cancelar"
30         if ( resultado == DialogResult.Cancel )
31             return;
32
33         nombreArchivo = selectorArchivo.FileName; // obtiene el nombre del archivo
34         especificado
35
36         // muestra error si el usuario especificó un archivo inválido
37         if ( nombreArchivo == "" || nombreArchivo == null )
38             MessageBox.Show( "Nombre de archivo inválido", "Error",
39                             MessageBoxButtons.OK, MessageBoxIcon.Error );
40         else
41         {
42             // guarda el archivo mediante el objeto FileStream, si el usuario especificó
43             un archivo válido
44             try

```

Figura 18.9 | Creación de un archivo de acceso secuencial para escribir en él. (Parte I de 4).

```

43     {
44         // abre el archivo con acceso de escritura
45         salida = new FileStream( nombreArchivo,
46             FileMode.OpenOrCreate, FileAccess.Write );
47
48         // establece el archivo para escribir los datos
49         archivoWriter = new StreamWriter( salida );
50
51         // deshabilita el botón Guardar y habilita el botón Entrar
52         guardarButton.Enabled = false;
53         entrarButton.Enabled = true;
54     } // fin de try
55     // maneja la excepción si hay un problema al abrir el archivo
56     catch ( IOException )
57     {
58         // notifica al usuario si el archivo no existe
59         MessageBox.Show( "Error al abrir el archivo", "Error",
60             MessageBoxButtons.OK, MessageBoxIcon.Error );
61     } // fin de catch
62 } // fin de else
63 } // fin del método guardarButton_Click
64
65 // manejador para entrarButton Click
66 private void entrarButton_Click( object sender, EventArgs e )
67 {
68     // almacena el arreglo string de valores de los controles TextBox
69     string[] valores = ObtenerValoresControlesTextBox();
70
71     // Registro que contiene los valores de los controles TextBox a serializar
72     Registro registro = new Registro();
73
74     // determina si el campo del control TextBox está vacío
75     if ( valores[ ( int ) IndicesTextBox.CUENTA ] != "" )
76     {
77         // almacena los valores de los controles TextBox en Registro y lo serializa
78         try
79         {
80             // obtiene el valor del número de cuenta del control TextBox
81             int numeroCuenta = Int32.Parse(
82                 valores[ ( int ) IndicesTextBox.CUENTA ] );
83
84             // determina si numeroCuenta es válido
85             if ( numeroCuenta > 0 )
86             {
87                 // almacena los campos TextBox en Registro
88                 registro.Cuenta = numeroCuenta;
89                 registro.PrimerNombre = valores[ ( int ) IndicesTextBox.NOMBRE ];
90                 registro.ApellidoPaterno = valores[ ( int ) IndicesTextBox.APELLIDO ];
91                 registro.Saldo = Decimal.Parse(
92                     valores[ ( int ) IndicesTextBox.SALDO ] );
93
94                 // escribe el Registro al archivo, los campos separados por comas
95                 archivoWriter.WriteLine(
96                     registro.Cuenta + "," + registro.PrimerNombre + "," +
97                     registro.ApellidoPaterno + "," + registro.Saldo );
98             } // fin de if
99         }
100         else
101         {
102             // notifica al usuario si el número de cuenta es inválido

```

Figura 18.9 | Creación de un archivo de acceso secuencial para escribir en él. (Parte 2 de 4).

```

102         MessageBox.Show( "Número de cuenta inválido", "Error",
103             MessageBoxButtons.OK, MessageBoxIcon.Error );
104     } // fin de else
105 } // fin de try
106 // notifica al usuario si ocurre un error en la serialización
107 catch ( IOException )
108 {
109     MessageBox.Show( "Error al escribir en archivo", "Error",
110         MessageBoxButtons.OK, MessageBoxIcon.Error );
111 } // fin de catch
112 // notifica al usuario si ocurre un error en relación con el formato de los
    parámetros
113 catch ( FormatException )
114 {
115     MessageBox.Show( "Formato inválido", "Error",
116         MessageBoxButtons.OK, MessageBoxIcon.Error );
117 } // fin de catch
118 } // fin de if
119
120 BorrarControlesTextBox(); // borra los valores de los controles TextBox
121 } // fin del método entrarButton_Click
122
123 // manejador para el evento Click de salirButton
124 private void salirButton_Click( object sender, EventArgs e )
125 {
126     // determina si el archivo existe o no
127     if ( salida != null )
128     {
129         try
130         {
131             archivoWriter.Close(); // cierra StreamWriter
132             salida.Close(); // cierra el archivo
133         } // fin de try
134         // notifica al usuario del error al cerrar el archivo
135         catch ( IOException )
136         {
137             MessageBox.Show( "No se puede cerrar el archivo", "Error",
138                 MessageBoxButtons.OK, MessageBoxIcon.Error );
139         } // fin de catch
140     } // fin de if
141
142     Application.Exit();
143 } // fin del método salirButton_Click
144 } // fin de la clase CrearArchivoForm

```

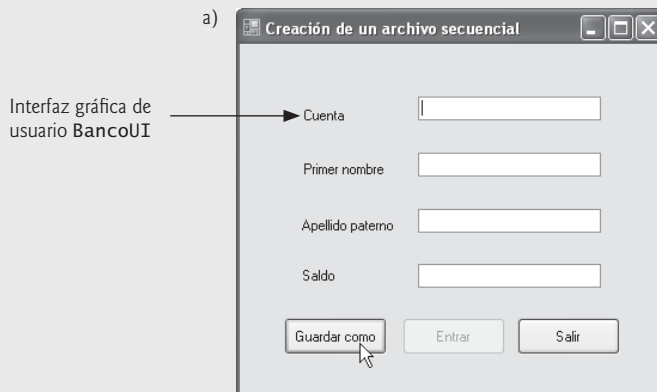


Figura 18.9 | Creación de un archivo de acceso secuencial para escribir en él. (Parte 3 de 4).

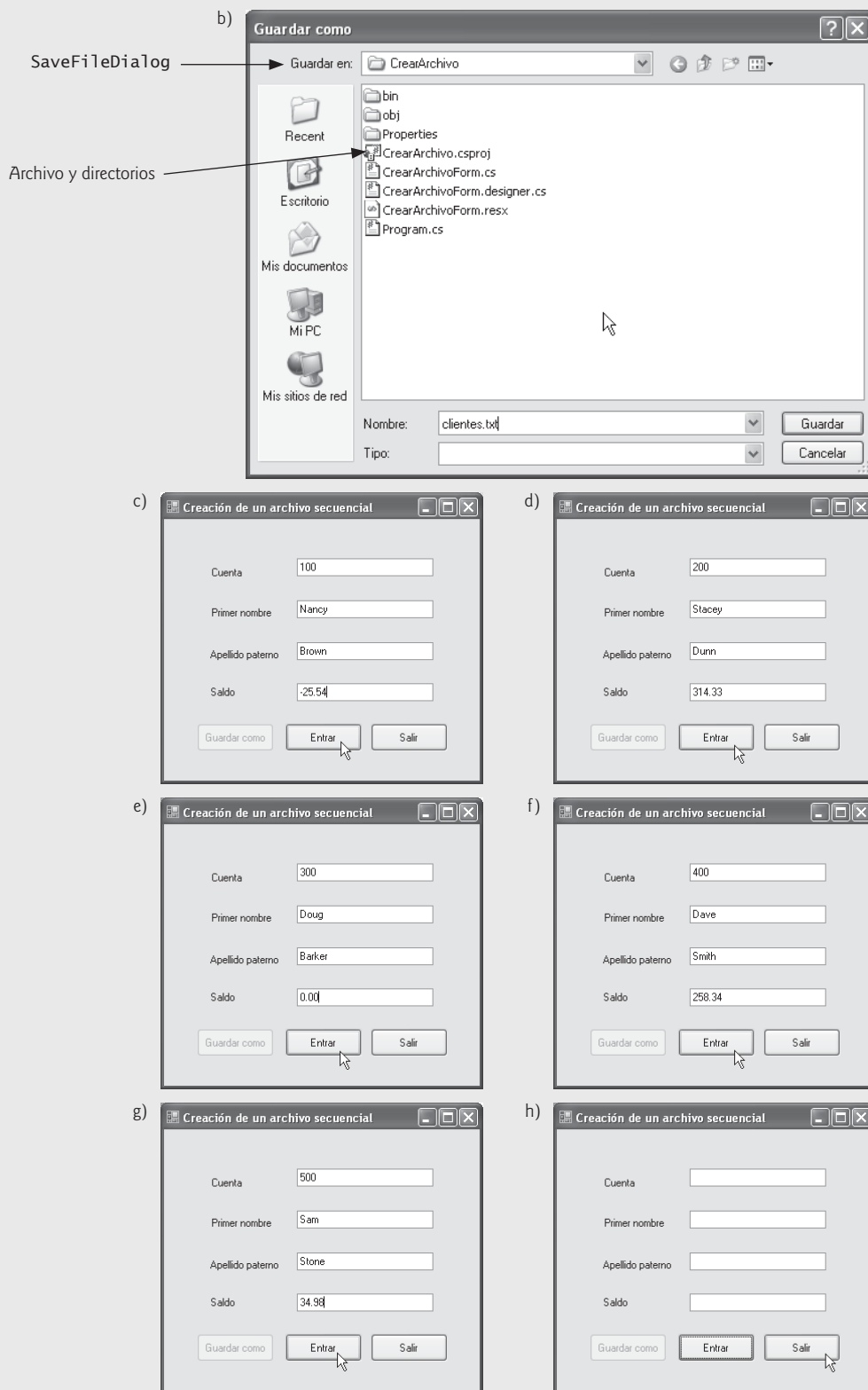


Figura 18.9 | Creación de un archivo de acceso secuencial para escribir en él. (Parte 4 de 4).

Puede abrir archivos para realizar la manipulación de texto mediante la creación de objetos `FileStream`. En este ejemplo, queremos abrir el archivo en modo de salida, por lo que las líneas 45-46 crean un objeto `FileStream`. El constructor de `FileStream` que usamos recibe tres argumentos: un objeto `string` que contiene la ruta y el nombre del archivo a abrir, una constante que describe cómo abrir el archivo y una constante que describe los permisos del archivo. La constante `FileMode.OpenOrCreate` (línea 46) indica que el objeto `FileStream` debe abrir el archivo si éste existe, o debe crearlo si no existe. Hay otras constantes de `FileMode` que describen cómo abrir archivos; introduciremos estas constantes a medida que las usemos en ejemplos. La constante `FileAccess.Write` indica que el programa sólo puede realizar operaciones de escritura con el objeto `FileStream`. Hay otras dos constantes para el tercer parámetro del constructor: `FileAccess.Read` para el acceso de sólo lectura y `FileAccess.ReadWrite` para el acceso de lectura y escritura. La línea 56 atrapa una excepción ***IOException*** si hay un problema al abrir el archivo, o al crear el objeto `StreamWriter`. De ser así, el programa muestra un mensaje de error (líneas 59-60). Si no ocurre una excepción, el archivo se abre en modo de escritura.



Buena práctica de programación 18.1

*Al abrir archivos, use la enumeración **FileAccess** para controlar el acceso del usuario a estos archivos.*



Error común de programación 18.1

Si no se abre un archivo antes de intentar hacer referencia al mismo en un programa, se produce un error lógico.

Una vez que el usuario escribe información en cada uno de los controles `TextBox`, hace clic en el botón **Entrar**, el cual llama al manejador de eventos `entrarButton_Click` (líneas 66-121) para guardar los datos de los controles `TextBox` en el archivo especificado por el usuario. Si el usuario introdujo un número de cuenta válido (es decir, un entero mayor que cero), las líneas 88-92 almacenan los valores de los controles `TextBox` en un objeto de tipo `Registro` (creado en la línea 72). Si el usuario introdujo datos inválidos en uno de los controles `TextBox` (como caracteres no numéricos en el campo **Saldo**), el programa lanza una excepción `FormatException`. El bloque `catch` en las líneas 113-117 maneja este tipo de excepciones, notificando al usuario (a través de un cuadro de diálogo `MessageBox`) sobre el formato inapropiado.

Si el usuario escribió datos válidos, las líneas 95-97 escriben el registro en el archivo mediante la invocación al método `WriteLine` del objeto `StreamWriter` que se creó en la línea 49. El método `WriteLine` escribe una secuencia de caracteres a un archivo. El objeto `StreamWriter` se construye con un argumento `FileStream`, que especifica el archivo al que el objeto `StreamWriter` enviará texto. La clase `StreamWriter` pertenece al espacio de nombres `System.IO`.

Cuando el usuario hace clic en el botón **Salir**, el manejador de eventos `salirButton_Click` (líneas 124-143) sale de la aplicación. La línea 131 cierra el objeto `StreamWriter` y la línea 132 cierra el objeto `FileStream`; después la línea 142 termina el programa. Observe que la llamada al método `Close` se encuentra en un bloque `try`. El método ***Close*** lanza una excepción `IOException` si el archivo o flujo no pueden cerrarse en forma apropiada. En este caso, es importante notificar al usuario que la información en el archivo o flujo podría estar corrupta.



Tip de rendimiento 18.1

Cierre cada archivo en forma explícita cuando el programa ya no necesite hacer referencia al mismo. Esto puede reducir el uso de recursos en los programas que siguen ejecutándose mucho tiempo después de que terminan de usar un archivo específico. La práctica de cerrar los archivos en forma explícita también mejora la legibilidad del programa.



Tip de rendimiento 18.2

Al liberar recursos en forma explícita cuando ya no se necesitan, éstos se vuelven disponibles de inmediato para que otros programas los reutilicen, con lo cual se mejora la utilización de recursos.

En la ejecución de ejemplo para el programa en la figura 18.9, introdujimos información para las cinco cuentas que se muestran en la figura 18.10. El programa no ilustra cómo se representan los registros de datos en el archivo. Para verificar que el archivo se haya creado con éxito, creamos un programa en la siguiente sección para leer y mostrar el archivo. Como éste es un archivo de texto, puede abrirlo en cualquier editor de texto para ver su contenido.

Número de cuenta	Primer nombre	Apellido paterno	Saldo
100	Nancy	Brown	-25.54
200	Stacey	Duna	314.33
300	Doug	Barrer	0.00
400	Dave	Smith	258.34
500	Sam	Stone	34.98

Figura 18.10 | Datos de ejemplo para el programa de la figura 18.9.

18.6 Lectura de datos desde un archivo de texto de acceso secuencial

La sección anterior demostró cómo crear un archivo para usarlo en aplicaciones con acceso secuencial. En esta sección veremos cómo leer (o recuperar) datos de un archivo en forma secuencial.

La clase `LeerArchivoAccesoSecuencialForm` (figura 18.11) lee registros del archivo creado por el programa en la figura 18.9 y después muestra el contenido de cada registro. La mayor parte del código en este ejemplo es similar al de la figura 18.9, por lo que sólo hablaremos de los aspectos únicos de la aplicación.

Cuando el usuario hace clic en el botón **Abrir archivo**, el programa llama al manejador de eventos `abrirButton_Click` (líneas 20-50). La línea 23 crea un cuadro de diálogo **OpenFileDialog**, y la línea 24 llama a su método **ShowDialog** para mostrar el cuadro de diálogo **Abrir** (vea la segunda captura de pantalla en la figura 18.11). El comportamiento y la GUI para los tipos de cuadros de diálogo **Guardar** y **Abrir** es idéntico, excepto que **Guardar** se sustituye por **Abrir**. Si el usuario introduce un nombre de archivo válido, las líneas 41-42 crean un

```

1  // Fig. 18.11: LeerArchivoAccesoSecuencialForm.cs
2  // Lectura de un archivo de acceso secuencial.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6  using BibliotecaBanco;
7
8  public partial class LeerArchivoAccesoSecuencialForm : BancoUIForm
9  {
10     private FileStream entrada; // mantiene la conexión a un archivo
11     private StreamReader archivoReader; // lee datos de un archivo de texto
12
13     // constructor sin parámetros
14     public LeerArchivoAccesoSecuencialForm()
15     {
16         InitializeComponent();
17     } // fin del constructor
18
19     // se invoca cuando el usuario hace clic en el botón Abrir
20     private void abrirButton_Click( object sender, EventArgs e )
21     {
22         // crea un cuadro de diálogo que permite al usuario abrir el archivo
23         OpenFileDialog selectorArchivo = new OpenFileDialog();
24         DialogResult resultado = selectorArchivo.ShowDialog();
25         string nombreArchivo; // nombre del archivo que contiene los datos
26
27         // sale del manejador de eventos si el usuario hace clic en Cancelar
28         if ( resultado == DialogResult.Cancel )
29             return;

```

Figura 18.11 | Lectura de archivos de acceso secuencial. (Parte I de 4).

```

30
31     nombreArchivo = selectorArchivo.FileName; // obtiene el nombre de archivo
        especificado
32     BorrarControlesTextBox();
33
34     // muestra error si el usuario especifica un archivo inválido
35     if ( nombreArchivo == "" || nombreArchivo == null )
36         MessageBox.Show( "Nombre de archivo inválido", "Error",
37             MessageBoxButtons.OK, MessageBoxIcon.Error );
38     else
39     {
40         // crea objeto FileStream para obtener acceso de lectura al archivo
41         entrada = new FileStream( nombreArchivo, FileMode.Open,
42             FileAccess.Read );
43
44         // establece el archivo del que se van a leer los datos
45         archivoReader = new StreamReader( entrada );
46
47         abrirButton.Enabled = false; // deshabilita el botón Abrir archivo
48         siguienteButton.Enabled = true; // habilita el botón Siguiente registro
49     } // fin de else
50 } // fin del método abrirButton_Click
51
52 // se invoca cuando el usuario hace clic en el botón Siguiente
53 private void siguienteButton_Click( object sender, EventArgs e )
54 {
55     try
56     {
57         // obtiene el siguiente registro disponible en el archivo
58         string registroEntrada = archivoReader.ReadLine();
59         string[] camposEntrada; // almacena piezas individuales de datos
60
61         if ( registroEntrada != null )
62         {
63             camposEntrada = registroEntrada.Split( ',' );
64
65             Registro registro = new Registro(
66                 Convert.ToInt32( camposEntrada[ 0 ] ), camposEntrada[ 1 ],
67                 camposEntrada[ 2 ], Convert.ToDecimal( camposEntrada[ 3 ] ) );
68
69             // copia los valores del arreglo string a los valores de los controles
                TextBox
70             EstablecerValoresControlesTextBox( camposEntrada );
71         } // fin de if
72     else
73     {
74         archivoReader.Close(); // cierra StreamReader
75         entrada.Close(); // cierra FileStream si no hay registros en el archivo
76         abrirButton.Enabled = true; // habilita el botón Abrir archivo
77         siguienteButton.Enabled = false; // deshabilita el botón Siguiente registro
78         BorrarControlesTextBox();
79
80         // notifica al usuario si no hay registros en el archivo
81         MessageBox.Show( "No hay más registros en el archivo", "",
82             MessageBoxButtons.OK, MessageBoxIcon.Information );
83     } // fin de else
84 } // fin de try
85 catch ( IOException )

```

Figura 18.11 | Lectura de archivos de acceso secuencial. (Parte 2 de 4).

```

86     {
87         MessageBox.Show( "Error al leer del archivo", "Error",
88             MessageBoxButtons.OK, MessageBoxIcon.Error );
89     } // fin de catch
90 } // fin del método siguienteButton_Click
91 } // fin de la clase LeerArchivoAccesoSecuencialForm

```

a)

b)

c)

d)

Figura 18.11 | Lectura de archivos de acceso secuencial. (Parte 3 de 4).

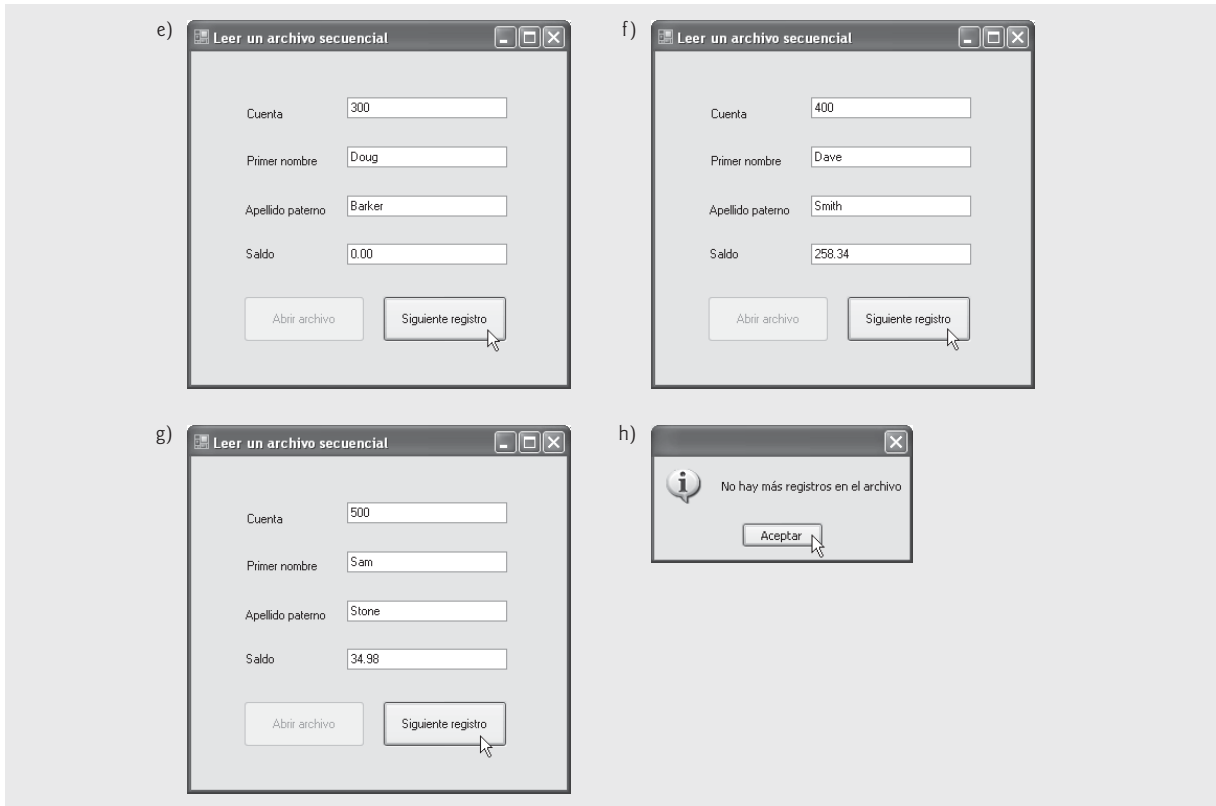


Figura 18.11 | Lectura de archivos de acceso secuencial. (Parte 4 de 4).

objeto `FileStream` y lo asignan a la referencia entrada. Pasamos la constante `FileMode.Open` como el segundo argumento para el constructor `FileStream`, para indicar que el objeto `FileStream` debe abrir el archivo si existe, o debe lanzar una excepción `FileNotFoundException` si el archivo no existe. (En este ejemplo, el constructor de `FileStream` no lanza una excepción `FileNotFoundException`, ya que el cuadro de diálogo `OpenFileDialog` requiere que el usuario introduzca el nombre de un archivo que exista.) En el ejemplo anterior (figura 18.9), escribimos texto en el archivo usando un objeto `FileStream` con acceso de sólo escritura. En este ejemplo (figura 18.11), especificamos un acceso de sólo lectura para el archivo, pasando la constante `FileAccess.Read` como el tercer argumento para el constructor de `FileStream`. El objeto `FileStream` se utiliza para crear un objeto `StreamReader` en la línea 45. El objeto `FileStream` especifica el archivo desde el que el objeto `StreamReader` leerá texto.



Tip de prevención de errores 18.1

Abra un archivo con el modo de apertura de archivo `FileAccess.Read` si el contenido del archivo no debe modificarse. Esto evita que el contenido se modifique de manera accidental.

Cuando el usuario hace clic en el botón **Siguiente registro**, el programa llama al manejador de eventos `siguienteButton_Click` (líneas 53-90), el cual lee el siguiente registro del archivo especificado por el usuario. (El usuario debe hacer clic en **Siguiente registro** después de abrir el archivo, para poder ver el primer registro.) La línea 58 llama al método `ReadLine` de `StreamReader` para leer el siguiente registro. Si ocurre un error mientras se lee el archivo, se lanza una excepción `IOException` (la cual se atrapa en la línea 85), y se le notifica al usuario (líneas 87-88). En caso contrario, la línea 61 determina si el método `ReadLine` de `StreamReader` devolvió `null` (es decir, que no hay más texto en el archivo). En caso contrario, la línea 63 usa el método `Split` de la clase `string` para separar el flujo de caracteres que se leyeron del archivo en objetos `string` que representen las propiedades de `Registro`. Después, estas propiedades se almacenan mediante la construcción de un objeto

Registro, usando las propiedades como argumentos (líneas 65-67). La línea 70 muestra los valores de Registro en los controles TextBox. Si ReadLine devuelve null, el programa cierra los objetos StreamReader (línea 74) y FileStream (línea 75), y después notifica al usuario que ya no hay más registros (líneas 81-82).

Búsqueda en un archivo de acceso secuencial

Para recuperar los datos en forma secuencial de un archivo, lo común es que un programa empiece desde el principio del archivo, leyendo en forma consecutiva hasta encontrar los datos deseados. Algunas veces es necesario procesar un archivo en forma secuencial varias veces (desde el principio del archivo) durante la ejecución de un programa. Un objeto FileStream puede reposicionar su *apuntador de posición de archivo* (el cual contiene el número del siguiente byte a leer desde, o escribir en, el archivo) a cualquier posición en el archivo. Cuando se abre un objeto FileStream, su apuntador de posición de archivo se establece a la posición de byte 0 (es decir, el principio del archivo).

Ahora presentaremos un programa que se basa en los conceptos empleados en la figura 18.11. La clase ConsultaCreditoForm (figura 18.12) es un programa de consulta de créditos, que permite a un gerente de créditos buscar y mostrar la información sobre las cuentas de los clientes que tienen saldos con crédito (es decir, los clientes que deben dinero a la compañía), saldos en cero (es decir, los clientes que no deben dinero a la compañía) y saldos con débito (es decir, los clientes que pagan dinero por los bienes y servicios recibidos previamente). Utilizamos un control RichTextBox en el programa para mostrar la información de las cuentas. Los controles RichTextBox ofrecen más funcionalidad que los controles TextBox ordinarios; por ejemplo, los controles RichTextBox ofrecen el método Find para buscar en cadenas individuales y el método LoadFile para mostrar el contenido de un archivo. Las clases RichTextBox y TextBox heredan de la clase abstract System.Windows.Forms.TextBoxBase. Elegimos un control RichTextBox en este ejemplo, ya que muestra varias líneas de texto de manera predeterminada, mientras que un control TextBox ordinario sólo muestra una. De manera alternativa, podríamos haber especificado que un objeto TextBox mostrara varias líneas de texto, estableciendo su propiedad Multiline a true.

```

1  // Fig. 18.12: ConsultaCreditoForm.cs
2  // Lee un archivo en forma secuencial y muestra el contenido con base en
3  // el tipo de cuenta especificado por el usuario ( saldos con crédito, débito o en cero ).
4  using System;
5  using System.Windows.Forms;
6  using System.IO;
7  using BibliotecaBanco;
8
9  public partial class ConsultaCreditoForm : Form
10 {
11     private FileStream entrada; // mantiene la conexión con el archivo
12     private StreamReader archivoReader; // lee datos del archivo de texto
13
14     // nombre del archivo que almacena los saldos con crédito, débito o en cero
15     private String nombreArchivo;
16
17     // constructor sin parámetros
18     public ConsultaCreditoForm()
19     {
20         InitializeComponent();
21     } // fin del constructor
22
23     // se invoca cuando el usuario hace clic en el botón Abrir archivo
24     private void abrirButton_Click( object sender, EventArgs e )
25     {
26         // crea un cuadro de diálogo que permite al usuario abrir un archivo
27         OpenFileDialog selectorArchivo = new OpenFileDialog();
28         DialogResult result = selectorArchivo.ShowDialog();

```

Figura 18.12 | Programa de consulta de créditos. (Parte I de 5).

```

29
30 // sale del manejador de eventos si el usuario hace clic en Cancelar
31 if ( result == DialogResult.Cancel )
32     return;
33
34 nombreArchivo = selectorArchivo.FileName; // obtiene el nombre de archivo del
    usuario
35
36 // muestra error si el usuario especificó un archivo inválido
37 if ( nombreArchivo == "" || nombreArchivo == null )
38     MessageBox.Show( "Nombre de archivo inválido", "Error",
39         MessageBoxButtons.OK, MessageBoxIcon.Error );
40 else
41 {
42     // crea objeto FileStream para obtener acceso de lectura al archivo
43     entrada = new FileStream( nombreArchivo,
44         FileMode.Open, FileAccess.Read );
45
46     // establece el archivo del que se van a leer los archivos
47     archivoReader = new StreamReader( entrada );
48
49     // habilita todos los botones de la GUI, excepto Abrir archivo
50     abrirButton.Enabled = false;
51     creditoButton.Enabled = true;
52     debitoButton.Enabled = true;
53     ceroButton.Enabled = true;
54 } // fin de else
55 } // fin del método abrirButton_Click
56
57 // se invoca cuando el usuario hace clic en el botón de saldos con crédito,
58 // saldos con débito o saldos en cero
59 private void obtenerSaldos_Click( object sender, System.EventArgs e )
60 {
61     // convierte el emisor explícitamente a un objeto de tipo Button
62     Button emisorButton = ( Button )sender;
63
64     // obtiene el texto del botón en el que se hizo clic, y que almacena el tipo de
        la cuenta
65     string tipoCuenta = emisorButton.Text;
66
67     // lee y muestra la información del archivo
68     try
69     {
70         // regresa al principio del archivo
71         entrada.Seek( 0, SeekOrigin.Begin );
72
73         mostrarTextBox.Text = "Las cuentas son:\r\n";
74
75         // recorre el archivo hasta llegar a su fin
76         while ( true )
77         {
78             string[] camposEntrada; // almacena piezas de datos individuales
79             Registro registro; // almacena cada Registro a medida que se lee el archivo
80             decimal saldo; // almacena el saldo de cada Registro
81
82             // obtiene el siguiente Registro disponible en el archivo
83             string registroEntrada = archivoReader.ReadLine();
84
85             // cuando está al final del archivo, sale del método

```

Figura 18.12 | Programa de consulta de créditos. (Parte 2 de 5).

```

86         if ( registroEntrada == null )
87             return;
88
89         camposEntrada = registroEntrada.Split( ',' ); // analiza la entrada
90
91         // crea el Registro a partir de entrada
92         registro = new Registro(
93             Convert.ToInt32( camposEntrada[ 0 ] ), camposEntrada[ 1 ],
94             camposEntrada[ 2 ], Convert.ToDecimal( camposEntrada[ 3 ] ) );
95
96         // almacena el último campo del registro en saldo
97         saldo = registro.Saldo;
98
99         // determina si va a mostrar el saldo o no
100        if ( DebeMostrar( saldo, tipoCuenta ) )
101        {
102            // muestra el registro
103            string salida = registro.Cuenta + "\t" +
104                registro.PrimerNombre + "\t" + registro.ApellidoPaterno + "\t";
105
106            // muestra el saldo con el formato monetario correcto
107            salida += String.Format( "{0:F}", saldo ) + "\r\n";
108
109            mostrarTextBox.Text += salida; // copia la salida a la pantalla
110        } // fin de if
111    } // fin de while
112 } // fin de try
113 // maneja la excepción cuando no puede leerse el archivo
114 catch ( IOException )
115 {
116     MessageBox.Show( "No se puede leer el archivo", "Error",
117         MessageBoxButtons.OK, MessageBoxIcon.Error );
118 } // fin de catch
119 } // fin del método obtenerSaldos_Click
120
121 // determina si se va a mostrar el registro dado
122 private bool DebeMostrar( decimal saldo, string tipoCuenta )
123 {
124     if ( saldo > 0 )
125     {
126         // muestra los saldos con crédito
127         if ( tipoCuenta == "Saldos con crédito" )
128             return true;
129     } // fin de if
130     else if ( saldo < 0 )
131     {
132         // mostrar los saldos con débito
133         if ( tipoCuenta == "Saldos con débito" )
134             return true;
135     } // fin de else if
136     else // saldo == 0
137     {
138         // muestra los saldos en cero
139         if ( tipoCuenta == "Saldos en cero" )
140             return true;
141     } // fin de else
142
143     return false;
144 } // fin del método DebeMostrar

```

Figura 18.12 | Programa de consulta de créditos. (Parte 3 de 5).

```

145
146 // se invoca cuando el usuario hace clic en el botón Terminar
147 private void terminarButton_Click( object sender, EventArgs e )
148 {
149     // determina si existe el archivo o no
150     if ( entrada != null )
151     {
152         // cierra el archivo y el objeto StreamReader
153         try
154         {
155             entrada.Close();
156             archivoReader.Close();
157         } // fin de try
158         // maneja la excepción si el objeto FileStream no existe
159         catch( IOException )
160         {
161             // notifica al usuario del error al cerrar el archivo
162             MessageBox.Show( "No se puede cerrar el archivo", "Error",
163                             MessageBoxButtons.OK, MessageBoxIcon.Error );
164         } // fin de catch
165     } // fin de if
166
167     Application.Exit();
168 } // fin del método terminarButton_Click
169 } // fin de la clase ConsultaCreditoForm

```

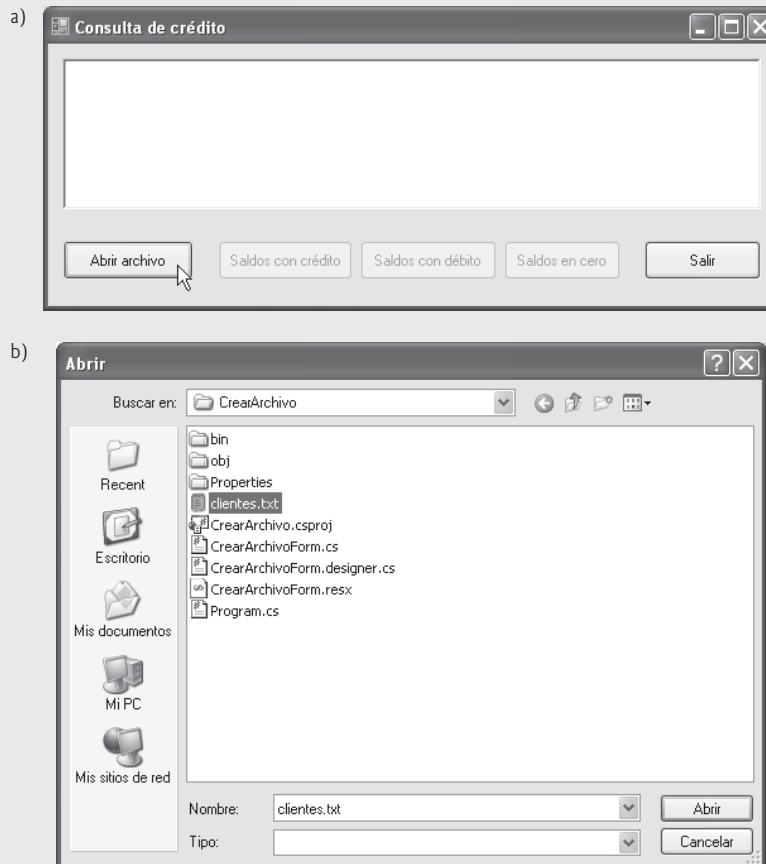


Figura 18.12 | Programa de consulta de créditos. (Parte 4 de 5).

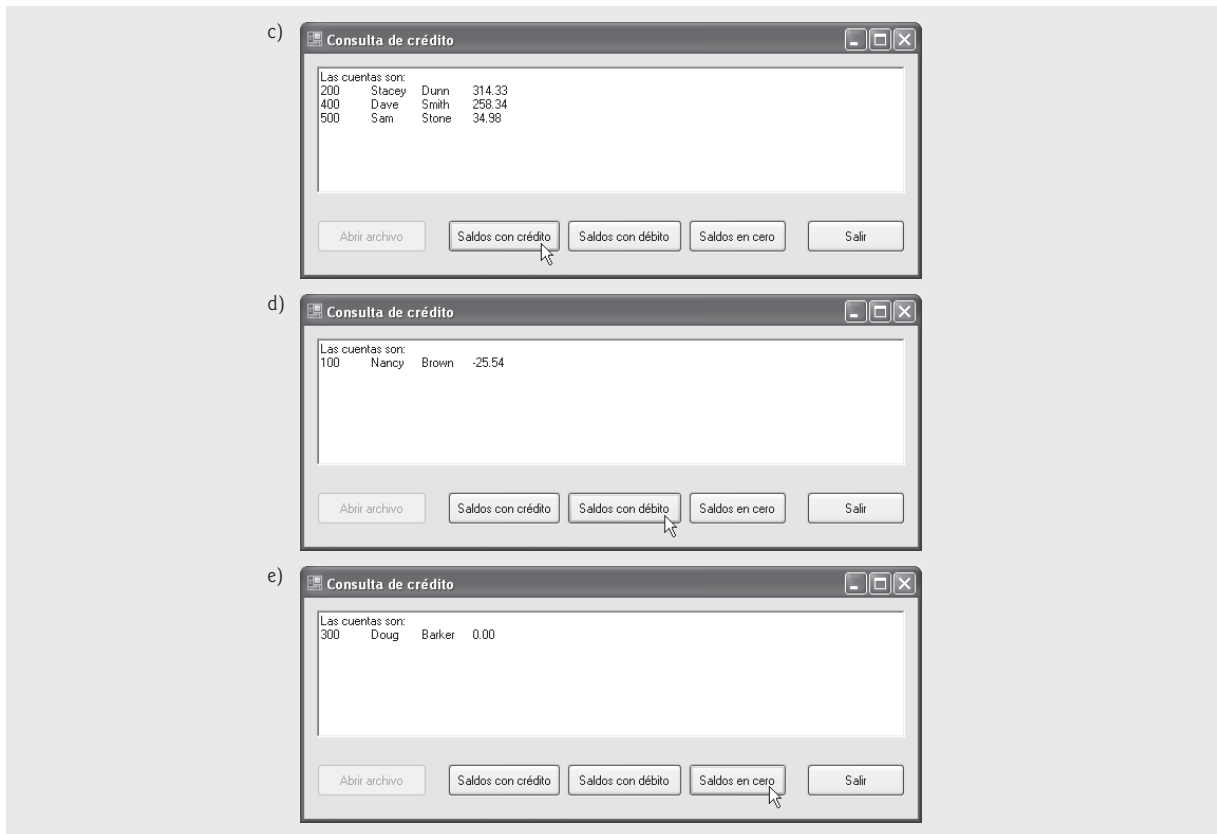


Figura 18.12 | Programa de consulta de créditos. (Parte 5 de 5).

El programa muestra botones que permiten a un gerente de crédito obtener la información sobre los créditos. El botón **Abrir archivo** abre un archivo para recopilar datos. El botón **Saldos con crédito** muestra una lista de cuentas que tienen saldos con crédito, el botón **Saldos con débito** muestra una lista de cuentas que tienen saldos con débito y el botón **Saldos en cero** muestra una lista de las cuentas que tienen saldos en cero. El botón **Terminar** sale de la aplicación.

Cuando el usuario hace clic en el botón **Abrir archivo**, el programa llama al manejador de eventos `abrir-Button_Click` (líneas 24-55). La línea 27 crea un cuadro de diálogo `OpenFileDialog` y la línea 28 llama a su método `ShowDialog` para mostrar el cuadro de diálogo **Abrir**, en el cual el usuario selecciona el archivo a abrir. Las líneas 43-44 crean un objeto `FileStream` con acceso de sólo lectura para el archivo, y lo asignan a la referencia entrada. La línea 47 crea un objeto `StreamReader` que utilizamos para leer texto del objeto `FileStream`.

Cuando el usuario hace clic en **Saldos con crédito**, **Saldos con débito** o **Saldos en cero**, el programa invoca al método `obtenerSaldos_Click` (líneas 59-119). La línea 62 convierte el parámetro `sender`, el cual es una referencia `object` al control que generó el evento, en un objeto `Button`. La línea 65 extrae el texto del objeto `Button`, que el programa utiliza para determinar el tipo de cuentas a mostrar. La línea 71 usa el método `Seek` de `FileStream` para restablecer el apuntador de posición del archivo de vuelta al principio del archivo. El método `Seek` de `FileStream` nos permite restablecer el apuntador de posición del archivo especificando el número de bytes que debe desplazarse a partir del principio del archivo, del final o de la posición actual. La parte del archivo a partir de la cual queremos desplazarnos se elige mediante el uso de constantes de la enumeración `SeekOrigin`. En este caso, nuestro flujo se desplaza 0 bytes a partir del principio del archivo (`SeekOrigin.Begin`). Las líneas 76-111 definen un ciclo `while` que utiliza el método `private DebeMostrar` (líneas 122-144) para determinar si se debe mostrar cada registro en el archivo. El ciclo `while` obtiene cada registro llamando repetidas veces al método `ReadLine` de `StreamReader` (línea 83), y dividiendo el texto en símbolos (tokens) que se utilizan para inicializar el objeto `registro` (líneas 89-94). La línea 86 determina si

el apuntador de posición del archivo ha llegado al final del archivo. De ser así, el programa regresa del método `obtenerSaldo_Click` (línea 87).

18.7 Serialización

La sección 18.5 demostró cómo escribir los campos individuales de un objeto `Registro` en un archivo de texto, y la sección 18.6 demostró cómo leer esos campos de un archivo y colocar sus valores en un objeto `Registro` en memoria. En los ejemplos, `Registro` se utilizó para juntar la información para un solo registro. Cuando las variables de instancia para un `Registro` se enviaron a un archivo en disco, se perdió cierta información como el tipo de cada valor. Por ejemplo, si se lee el valor "3" de un archivo, no hay forma de saber si el valor proviene de un `int`, de un `string` o de un `decimal`. Sólo tenemos datos en el disco, no información sobre el tipo. Si el programa que va a leer estos datos "sabe" a qué tipo de objeto corresponden los datos, entonces pueden leerse directamente en objetos de ese tipo. Por ejemplo, en la figura 18.9 sabemos que introduciremos un `int` (el número de cuenta), seguido de dos objetos `string` (el primer nombre y el apellido paterno) y un `decimal` (el saldo). También sabemos que estos valores están separados por comas, y sólo hay un registro en cada línea. Por lo tanto, podemos analizar los objetos `string` y convertir el número de cuenta en un `int` y el balance en un `decimal`. Algunas veces sería más sencillo leer o escribir objetos completos. C# cuenta con un mecanismo así, al cual se le conoce como *serialización de objetos*. Un *objeto serializado* se representa como una secuencia de bytes que incluye los datos de ese objeto, así como información acerca del tipo del objeto y los tipos de los datos almacenados en ese objeto. Una vez que se escribe un objeto serializado en un archivo, puede leerse desde ese archivo y *deserializarse*; esto es, la información sobre el tipo y los bytes que representan al objeto y sus datos pueden usarse para recrear al objeto en la memoria.

La clase **`BinaryFormatter`** (espacio de nombres **`System.Runtime.Serialization.Formatters.Binary`**) permite escribir o leer objetos completos en/desde un flujo. El método **`Serialize`** de `BinaryFormatter` escribe en un archivo la representación de un objeto. El método **`Deserialize`** de `BinaryFormatter` lee esta representación de un archivo y reconstruye el objeto original. Ambos métodos lanzan una excepción **`SerializationException`** si ocurre un error durante la serialización o la deserialización. Ambos métodos requieren un objeto `Stream` (por ejemplo, `FileStream`) como parámetro, para que el objeto `BinaryFormatter` pueda acceder al flujo correcto. Como veremos en el capítulo 23, Redes: sockets basados en flujos y datagramas, la serialización puede usarse para transmitir objetos entre aplicaciones en una red.

En las secciones 18.8-18.9 creamos y manipulamos archivos de acceso secuencial, usando la serialización de objetos, que se realiza con flujos basados en bytes, de manera que los archivos secuenciales que se creen y se manipulen serán archivos binarios. Los humanos no podemos leer archivos binarios. Por esta razón, escribimos una aplicación separada que lee y muestra en pantalla objetos serializados.

18.8 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos

Empezaremos por crear y escribir objetos serializados en un archivo de acceso secuencial. En esta sección reutilizaremos la mayor parte del código de la sección 18.5, por lo que nos concentraremos sólo en las nuevas características.

Definición de la clase `RegistroSerializable`

Comenzaremos por modificar nuestra clase `Registro` (figura 18.8), de manera que los objetos de esta clase puedan serializarse. La clase `RegistroSerializable` (figura 18.13) está marcada con el atributo **`[Serializable]`** (línea 5), el cual indica al CLR que los objetos de la clase `Registro` pueden serializarse. Las clases para los objetos que queremos escribir o leer en/desde un flujo deben incluir este atributo en sus declaraciones, o deben implementar la interfaz **`ISerializable`**. La clase `RegistroSerializable` contiene los miembros de datos `private` `cuenta`, `primerNombre`, `apellidoPaterno` y `saldo`. Esta clase también cuenta con propiedades `public` para acceder a los campos `private`.

En una clase marcada con el atributo **`[Serializable]`** o que implementa la interfaz **`ISerializable`**, debemos asegurarnos que toda variable de instancia de la clase también sea serializable. Todas las variables de tipos simples y los objetos `string` son serializables. Para las variables de tipos por referencia, hay que comprobar la declaración de la clase (y posiblemente sus clases base) para asegurarnos que el tipo sea serializable. Los objetos tipo arreglo son serializables de manera predeterminada. No obstante, si el arreglo contiene referencias a otros objetos, esos objetos podrían o no ser serializables.

```

1  // Fig. 18.13: RegistroSerializable.cs
2  // Clase serializable que representa a un registro de datos.
3  using System;
4
5  [ Serializable ]
6  public class RegistroSerializable
7  {
8      private int cuenta;
9      private string primerNombre;
10     private string apellidoPaterno;
11     private decimal saldo;
12
13     // el constructor predeterminado establece los miembros a los valores predeterminados
14     public RegistroSerializable()
15         : this( 0, "", "", 0.0M )
16     {
17     } // fin del constructor
18
19     // el constructor sobrecargado establece los miembros a los valores de los parámetros
20     public RegistroSerializable( int valorCuenta, string valorPrimerNombre,
21                                string valorApellidoPaterno, decimal valorSaldo )
22     {
23         Cuenta = valorCuenta;
24         PrimerNombre = valorPrimerNombre;
25         ApellidoPaterno = valorApellidoPaterno;
26         Saldo = valorSaldo;
27     } // fin del constructor
28
29     // propiedad que obtiene y establece Cuenta
30     public int Cuenta
31     {
32         get
33         {
34             return cuenta;
35         } // fin de get
36         set
37         {
38             cuenta = value;
39         } // fin de set
40     } // fin de la propiedad Cuenta
41
42     // propiedad que obtiene y establece PrimerNombre
43     public string PrimerNombre
44     {
45         get
46         {
47             return primerNombre;
48         } // fin de get
49         set
50         {
51             primerNombre = value;
52         } // fin de set
53     } // fin de la propiedad PrimerNombre
54
55     // propiedad que obtiene y establece ApellidoPaterno
56     public string ApellidoPaterno
57     {
58         get
59         {

```

Figura 18.13 | La clase RegistroSerializable para objetos serializables. (Parte 1 de 2).

```

60         return apellidoPaterno;
61     } // fin de get
62     set
63     {
64         apellidoPaterno = value;
65     } // fin de set
66 } // fin de la propiedad ApellidoPaterno
67
68 // propiedad que obtiene y establece Saldo
69 public decimal Saldo
70 {
71     get
72     {
73         return saldo;
74     } // fin de get
75     set
76     {
77         saldo = value;
78     } // fin de set
79 } // fin de la propiedad Saldo
80 } // fin de la clase RegistroSerializable

```

Figura 18.13 | La clase RegistroSerializable para objetos serializables. (Parte 2 de 2).

Uso de un flujo de serialización para crear un archivo de salida

Ahora crearemos un archivo de acceso secuencial mediante la serialización (figura 18.14). La línea 13 crea un objeto `BinaryFormatter` para escribir objetos serializados. Las líneas 48-49 abren el objeto `FileStream`, en el que este programa escribe los objetos serializados. El argumento `string` que se pasa al constructor de `FileStream` representa el nombre y la ruta del archivo que se abrirá. Este argumento especifica el archivo en el cual se van a escribir los objetos serializados.



Error común de programación 18.2

Es un error lógico abrir un archivo existente en modo de salida, cuando el usuario desea preservar el archivo, ya que se perderá el contenido original del archivo.

Este programa asume que los datos se introducen en forma correcta y en el orden numérico apropiado de los registros. El manejador de eventos `introducirButton_Click` (líneas 66-119) realiza la operación de escritura. La línea 72 crea un objeto `RegistroSerializable`, al cual se le asignan los valores en las líneas 88-92. La línea 95 llama al método `Serialize` para escribir el objeto `RegistroSerializable` en el archivo de salida. El método `Serialize` recibe el objeto `FileStream` como el primer argumento, de manera que el objeto `BinaryFormatter` pueda escribir su segundo argumento en el archivo correcto. Observe que sólo se requiere una instrucción para escribir todo el objeto.

En la ejecución de ejemplo para el programa en la figura 18.14, introducimos información para cinco cuentas; la misma información que se muestra en la figura 18.10. El programa no muestra cómo aparecen en realidad los registros de datos en el archivo. Recuerde que ahora estamos usando archivos binarios, que no pueden leer los humanos. Para verificar que el archivo se haya creado con éxito, la siguiente sección presenta un programa para leer el contenido del archivo.

```

1  // Fig 18.14: CrearArchivoForm.cs
2  // Creación de un archivo de acceso secuencial mediante la serialización.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;

```

Figura 18.14 | Archivo secuencial creado mediante la serialización. (Parte 1 de 5).

```

6  using System.Runtime.Serialization.Formatters.Binary;
7  using System.Runtime.Serialization;
8  using BibliotecaBanco;
9
10 public partial class CrearArchivoForm : BancoUIForm
11 {
12     // objeto para serializar Registros en formato binario
13     private BinaryFormatter aplicadorFormato = new BinaryFormatter();
14     private FileStream salida; // flujo para escribir en un archivo
15
16     // constructor sin parámetros
17     public CrearArchivoForm()
18     {
19         InitializeComponent();
20     } // fin del constructor
21
22     // manejador para guardarButton_Click
23     private void guardarButton_Click( object sender, EventArgs e )
24     {
25         // crea un cuadro de diálogo que permite al usuario guardar el archivo
26         SaveFileDialog selectorArchivo = new SaveFileDialog();
27         DialogResult resultado = selectorArchivo.ShowDialog();
28         string nombreArchivo; // nombre del archivo para guardar los datos
29
30         selectorArchivo.CheckFileExists = false; // permite al usuario crear el archivo
31
32         // sale del manejador de eventos si el usuario hace clic en "Cancelar"
33         if ( resultado == DialogResult.Cancel )
34             return;
35
36         nombreArchivo = selectorArchivo.FileName; // obtiene el nombre de archivo especificado
37
38         // muestra error si el usuario especificó un archivo inválido
39         if ( nombreArchivo == "" || nombreArchivo == null )
40             MessageBox.Show( "Nombre de archivo inválido", "Error",
41                             MessageBoxButtons.OK, MessageBoxIcon.Error );
42         else
43         {
44             // guarda el archivo a través del objeto FileStream si el usuario especificó un archivo válido
45             try
46             {
47                 // abre el archivo con acceso de escritura
48                 salida = new FileStream( nombreArchivo,
49                                         FileMode.OpenOrCreate, FileAccess.Write );
50
51                 // deshabilita el botón Guardar y habilita el botón Introducir
52                 guardarButton.Enabled = false;
53                 introducirButton.Enabled = true;
54             } // fin de try
55             // maneja la excepción si hay un problema al abrir el archivo
56             catch ( IOException )
57             {
58                 // notifica al usuario si el archivo no existe
59                 MessageBox.Show( "Error al abrir el archivo", "Error",
60                                 MessageBoxButtons.OK, MessageBoxIcon.Error );
61             } // fin de catch
62         } // fin de else

```

Figura 18.14 | Archivo secuencial creado mediante la serialización. (Parte 2 de 5).

```

63     } // fin del método guardarButton_Click
64
65     // manejador para el evento Click de introducirButton
66     private void introducirButton_Click( object sender, EventArgs e )
67     {
68         // almacena los valores de los controles TextBox en el arreglo string valores
69         string[] valores = ObtenerValoresControlesTextBox();
70
71         // Registro que contiene los valores de los controles TextBox a serializar
72         RegistroSerializable registro = new RegistroSerializable();
73
74         // determina si el campo del control TextBox de la cuenta está vacío
75         if ( valores[ ( int ) IndicesTextBox.CUENTA ] != "" )
76         {
77             // almacena los valores de los controles TextBox en Registro y lo serializa
78             try
79             {
80                 // obtiene el valor del número de cuenta del control TextBox
81                 int numeroCuenta = Int32.Parse(
82                     valores[ ( int ) IndicesTextBox.CUENTA ] );
83
84                 // determina si el numeroCuenta es válido
85                 if ( numeroCuenta > 0 )
86                 {
87                     // almacena los campos de los controles TextBox en Registro
88                     registro.Cuenta = numeroCuenta;
89                     registro.PrimerNombre = valores[ ( int ) IndicesTextBox.NOMBRE ];
90                     registro.ApellidoPaterno = valores[ ( int ) IndicesTextBox.APELLIDO ];
91                     registro.Saldo = Decimal.Parse( valores[
92                         ( int ) IndicesTextBox.SALDO ] );
93
94                     // escribe Record al objeto FileStream ( serializa el objeto )
95                     aplicadorFormato.Serialize( salida, registro );
96                 } // fin de if
97             } else
98             {
99                 // notifica al usuario si el número de cuenta es inválido
100                 MessageBox.Show( "Número de cuenta inválido", "Error",
101                     MessageBoxButtons.OK, MessageBoxIcon.Error );
102             } // fin de else
103         } // fin de try
104         // notifica al usuario si ocurre un error en la serialización
105         catch ( SerializationException )
106         {
107             MessageBox.Show( "Error al escribir en el archivo", "Error",
108                 MessageBoxButtons.OK, MessageBoxIcon.Error );
109         } // fin de catch
110         // notifica al usuario si ocurre un error en relación con el formato de los
111         // parámetros
112         catch ( FormatException )
113         {
114             MessageBox.Show( "Formato inválido", "Error",
115                 MessageBoxButtons.OK, MessageBoxIcon.Error );
116         } // fin de catch
117     } // fin de if
118     BorrarControlesTextBox(); // borra los valores de los controles TextBox
119 } // fin del método introducirButton_Click
120

```

Figura 18.14 | Archivo secuencial creado mediante la serialización. (Parte 3 de 5).

```

121 // manejador para el evento Click de salirButton
122 private void salirButton_Click( object sender, EventArgs e )
123 {
124     // determina si existe el archivo
125     if ( salida != null )
126     {
127         // cierra el archivo
128         try
129         {
130             salida.Close();
131         } // fin de try
132         // notifica al usuario del error al cerrar el archivo
133         catch ( IOException )
134         {
135             MessageBox.Show( "No se pudo cerrar el archivo", "Error",
136                             MessageBoxButtons.OK, MessageBoxIcon.Error );
137         } // fin de catch
138     } // fin de if
139
140     Application.Exit();
141 } // fin del método salirButton_Click
142 } // fin de la clase CrearArchivoForm

```

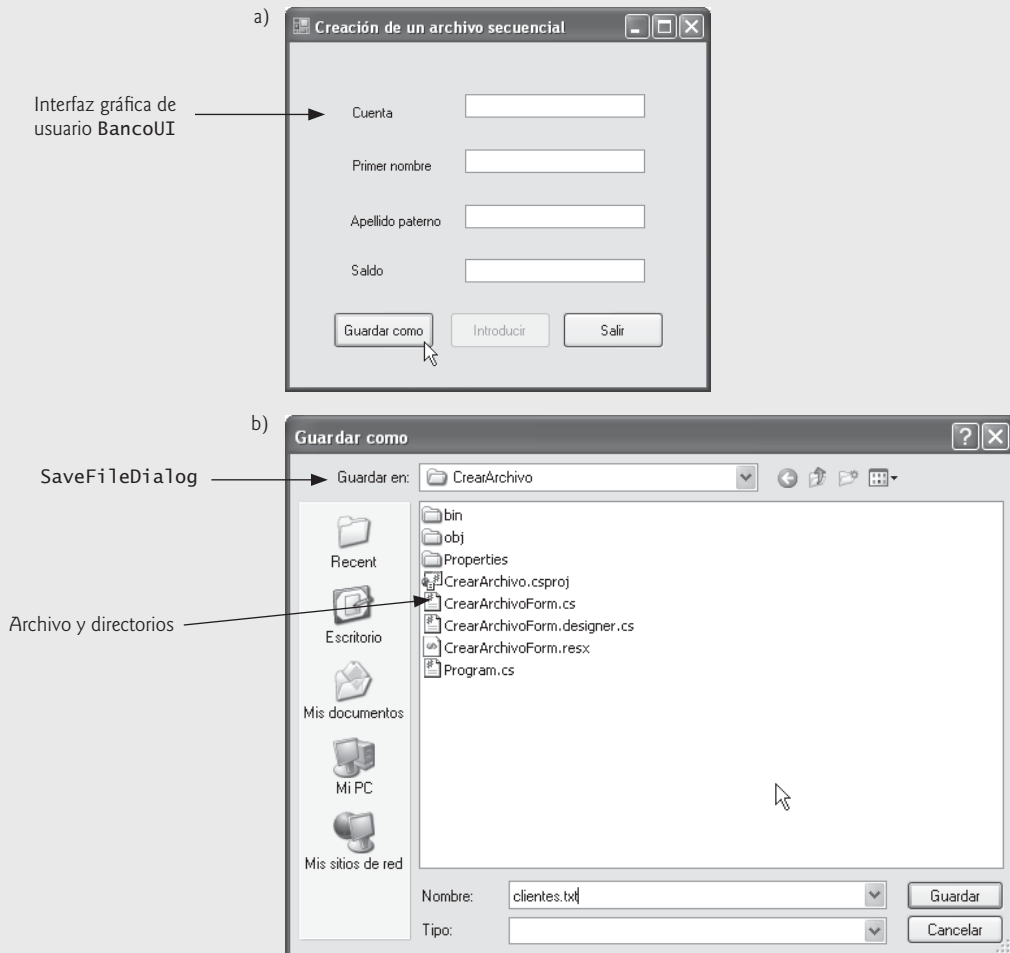


Figura 18.14 | Archivo secuencial creado mediante la serialización. (Parte 4 de 5).

Figura 18.14 | Archivo secuencial creado mediante la serialización. (Parte 5 de 5).

18.9 Lectura y deserialización de datos de un archivo de texto de acceso secuencial

La sección anterior mostró cómo crear un archivo de acceso secuencial usando la serialización de objetos. En esta sección veremos cómo leer objetos serializados de un archivo en forma secuencial.

La figura 18.15 lee y muestra el contenido del archivo creado por el programa en la figura 18.14. La línea 13 crea el objeto `BinaryFormatter` que se utilizará para leer objetos. El programa abre el archivo en modo de

```

1 // Fig. 18.15: LeerArchivoAccesoSecuencialForm.cs
2 // Lectura de un archivo de acceso secuencial mediante la serialización.
3 using System;
4 using System.Windows.Forms;

```

Figura 18.15 | Lectura de un archivo secuencial mediante la deserialización. (Parte 1 de 4).


```

5  using System.IO;
6  using System.Runtime.Serialization.Formatters.Binary;
7  using System.Runtime.Serialization;
8  using BibliotecaBanco;
9
10 public partial class LeerArchivoAccesoSecuencialForm : BancoUIForm
11 {
12     // objeto para deserializar el Registro en formato binario
13     private BinaryFormatter lector = new BinaryFormatter();
14     private FileStream entrada; // flujo para leer de un archivo
15
16     // constructor sin parámetros
17     public LeerArchivoAccesoSecuencialForm()
18     {
19         InitializeComponent();
20     } // fin del constructor
21
22     // se invoca cuando el usuario hace clic en el botón Abrir
23     private void abrirButton_Click( object sender, EventArgs e )
24     {
25         // crea un cuadro de diálogo que permite al usuario abrir un archivo
26         OpenFileDialog selectorArchivo = new OpenFileDialog();
27         DialogResult resultado = selectorArchivo.ShowDialog();
28         string nombreArchivo; // nombre del archivo que contiene los datos
29
30         // sale del manejador de eventos si el usuario hace clic en Cancelar
31         if ( resultado == DialogResult.Cancel )
32             return;
33
34         nombreArchivo = selectorArchivo.FileName; // obtiene el nombre de archivo especificado
35         BorrarControlesTextBox();
36
37         // muestra error si el usuario especificó un archivo inválido
38         if ( nombreArchivo == "" || nombreArchivo == null )
39             MessageBox.Show( "Nombre de archivo inválido", "Error",
40                             MessageBoxButtons.OK, MessageBoxIcon.Error );
41         else
42         {
43             // crea objeto FileStream para obtener acceso de lectura al archivo
44             entrada = new FileStream(
45                 nombreArchivo, FileMode.Open, FileAccess.Read );
46
47             abrirButton.Enabled = false; // deshabilita el botón Abrir archivo
48             siguienteButton.Enabled = true; // habilita el botón Siguiente registro
49         } // fin de else
50     } // fin del método abrirButton_Click
51
52     // se invoca cuando el usuario hace clic en el botón Siguiente
53     private void siguienteButton_Click( object sender, EventArgs e )
54     {
55         // deserializa el Registro y almacena los datos en controles TextBox
56         try
57         {
58             // obtiene el siguiente RegistroSerializable disponible en el archivo
59             RegistroSerializable registro =
60                 ( RegistroSerializable ) lector.Deserialize( entrada );
61
62             // almacena los valores del Registro en un arreglo string temporal
63             string[] valores = new string[] {

```

Figura 18.15 | Lectura de un archivo secuencial mediante la deserialización. (Parte 2 de 4).

```

64         registro.Cuenta.ToString(),
65         registro.PrimerNombre.ToString(),
66         registro.ApellidoPaterno.ToString(),
67         registro.Saldo.ToString()
68     };
69     // copia los valores del arreglo string a los controles TextBox
70     EstablecerValoresControlesTextBox( valores );
71 } // fin de try
72 // maneja la excepción cuando no hay registros en el archivo
73 catch( SerializationException )
74 {
75     entrada.Close(); // cierra objeto FileStream si no hay registros en el archivo
76     abrirButton.Enabled = true; // habilita el botón Abrir archivo
77     siguienteButton.Enabled = false; // deshabilita el botón Siguiente registro
78
79     BorrarControlesTextBox();
80
81     // notifica al usuario si no hay registros en el archivo
82     MessageBox.Show( "No hay más registros en el archivo", "",
83                     MessageBoxButtons.OK, MessageBoxIcon.Information );
84 } // fin de catch
85 } // fin del método siguienteButton_Click
86 } // fin de la clase LeerArchivoAccesoSecuencialForm

```

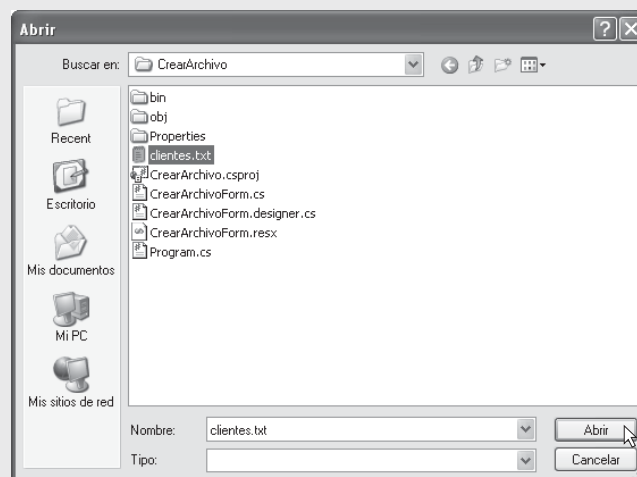


Figura 18.15 | Lectura de un archivo secuencial mediante la deserialización. (Parte 3 de 4).

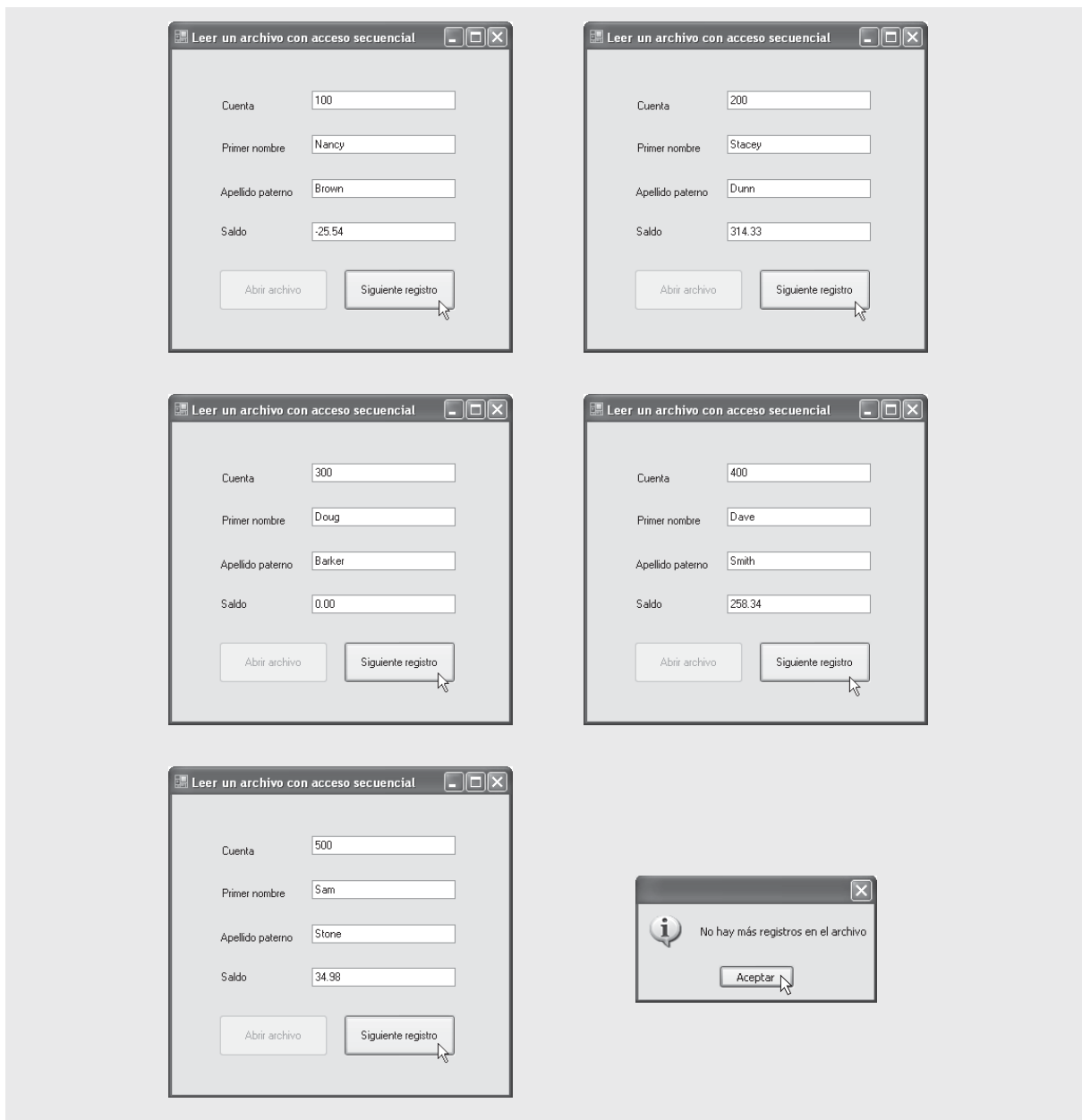


Figura 18.15 | Lectura de un archivo secuencial mediante la deserialización. (Parte 4 de 4).

entrada mediante la creación de un objeto `FileStream` (líneas 44-45). El nombre del archivo a abrir se especifica como el primer argumento para el constructor de `FileStream`.

El programa lee objetos de un archivo en el manejador de eventos `siguienteButton_Click` (líneas 53-85). Utilizamos el método `Deserialize` (del objeto `BinaryFormatter` creado en la línea 13) para leer los datos (líneas 59-60). Observe que convertimos el resultado de `Deserialize` al tipo `RegistroSerializable` (línea 60); esta conversión es necesaria, ya que `Deserialize` devuelve una referencia de tipo `object` y necesitamos acceder a propiedades que pertenecen a la clase `RegistroSerializable`. Si ocurre un error durante la deserialización, se lanza una excepción `SerializationException`, y se cierra el objeto `FileStream` (línea 75).

18.10 Conclusión

En este capítulo vimos cómo utilizar el procesamiento de archivos para manipular datos persistentes. Aprendió que los datos se almacenan en las computadoras como 0s y 1s, y que las combinaciones de estos valores se utilizan para formar bytes, campos, registros y, en un momento dado, archivos. Vimos las generalidades acerca de las diferencias entre los flujos basados en carácter y los flujos basados en byte, así como varias clases de procesamiento de archivos del espacio de nombres `System.IO`. Utilizó la clase `File` para manipular archivos y la clase `Directory` para manipular directorios. Después aprendió a utilizar el procesamiento de archivos de acceso secuencial para manipular registros en archivos de texto. Más adelante hablamos sobre las diferencias entre el procesamiento de archivos de texto y la serialización de objetos, y utilizamos la serialización para almacenar objetos completos en, y recuperar objetos completos de, los archivos.

En el siguiente capítulo presentamos el Lenguaje de marcado extensible (XML): una tecnología para describir datos con amplio soporte. Mediante el uso de XML, podemos describir cualquier tipo de datos, como fórmulas matemáticas, música y reportes financieros. Le demostraremos cómo describir datos con XML y cómo escribir programas que puedan procesar los datos codificados en XML.

19

Lenguaje de marcado extensible (XML)

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Marcar datos mediante el uso de XML.
- Cómo los espacios de nombres de XML ayudan a proporcionar nombres únicos para los elementos y atributos de XML.
- Crear DTDs y esquemas para especificar y validar la estructura de un documento XML.
- Crear y utilizar hojas de estilo XSL simples para representar los datos de documentos XML.
- Recuperar y modificar los datos XML mediante la programación, usando las clases del .NET Framework.
- Validar los documentos XML contra los esquemas, usando la clase `XmlReader`.
- Transformar documentos XML en XHTML, mediante el uso de la clase `XsltCompiledTransform`.

*Conociendo a los árboles, comprendo
el significado de la paciencia.
Conociendo al césped, puedo
apreciar la persistencia.*

—Hal Borland

*Al igual que todo
lo metafísico, la armonía
entre el pensamiento y la realidad
se encuentra en la gramática
del lenguaje.*

—Ludwig Wittgenstein

*Jugué con una idea, y
crecí voluntarioso; la sacudí
en el aire; la transformé, dejé
que escapara y la recobré;
la hice iridiscente con la
suposición, y dejé que
volara con paradoja.*

—Oscar Wilde

- 19.1 Introducción
- 19.2 Fundamentos de XML
- 19.3 Estructuración de datos
- 19.4 Espacios de nombres de XML
- 19.5 Definiciones de tipo de documento (DTDs)
- 19.6 Documentos de esquemas XML del W3C
- 19.7 (Opcional) Lenguaje de hojas de estilos extensible y transformaciones XSL
- 19.8 (Opcional) Modelo de objetos de documento (DOM)
- 19.9 (Opcional) Validación de esquemas con la clase `XmlReader`
- 19.10 (Opcional) XSLT con la clase `XsltCompiledTransform`
- 19.11 Conclusión
- 19.12 Recursos Web

19.1 Introducción

El *Lenguaje de marcado extensible (XML)* fue desarrollado en 1996 por el Grupo de trabajo de XML del *Consortio World Wide Web (W3C)*. XML es una *tecnología abierta* (es decir, tecnología no propietaria) con amplio soporte para describir datos, y se ha convertido en el formato estándar para el intercambio de datos entre aplicaciones a través de Internet.

El .NET Framework hace un uso extenso de XML. La Biblioteca de clases del .NET Framework proporciona un conjunto extenso de clases relacionadas con el XML, y gran parte de la implementación interna de Visual Studio también emplea XML. Las secciones 19.2-19.6 introducen el XML y las tecnologías relacionadas con XML: los espacios de nombres de XML para proporcionar nombres únicos para los elementos y atributos de XML, y las Definiciones de tipo de documento (DTDs) y los Esquemas de XML para validar documentos de XML. Estas secciones son obligatorias para soportar el uso de XML en los capítulos 20 al 22. Las secciones 19.7-19.10 presentan tecnologías de XML adicionales y clases clave del .NET Framework para crear y manipular documentos XML mediante la programación; este material es opcional, pero lo recomendamos para los lectores que planeen emplear XML en sus propias aplicaciones en C#.

19.2 Fundamentos de XML

XML permite a los autores de documentos crear *marcado* (es decir, una notación basada en texto para describir datos) para casi cualquier tipo de información. Esto permite a los autores de documentos crear lenguajes de marcado completamente nuevos para describir cualquier tipo de datos, como las fórmulas matemáticas, las instrucciones de configuración de software, las estructuras moleculares químicas, la música, las noticias, las recetas y los reportes financieros. XML describe los datos en una forma que tanto los seres humanos como las computadoras pueden comprender.

La figura 19.1 es un documento XML simple que describe la información para un jugador de béisbol. Nos enfocaremos en las líneas 5-11 para presentar la sintaxis básica de XML. En la sección 19.3 aprenderá acerca de los demás elementos de este documento.

Los documentos de XML contienen texto que representa contenido (es decir, datos), por ejemplo, John (línea 6 de la figura 19.1), y *elementos* que especifican la estructura del documento, como `primerNombre` (línea 6 de la figura 19.1). Los documentos de XML delimitan los elementos con *etiquetas iniciales* y *etiquetas finales*. Una etiqueta inicial consiste en el nombre del elemento entre un *signo menor que*, `<`, y un *signo mayor que*, `>` (por ejemplo, `<jugador>` y `<primerNombre>` en las líneas 5 y 6, respectivamente). Una etiqueta final consiste en el nombre del elemento precedido por una *barra diagonal* (`/`) entre corchetes (por ejemplo, `</primerNombre>` y `</jugador>` en las líneas 6 y 11, respectivamente). Las etiquetas inicial y final de un elemento encierran texto que representa una pieza de datos (por ejemplo, el `primerNombre` del jugador, John, en la línea 6, que va encerrado por la etiqueta inicial `<primerNombre>` y la etiqueta final `</primerNombre>`). Todo documento de XML debe

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.1: jugador.xml -->
3  <!-- Jugador de béisbol estructurado con XML -->
4
5  <jugador>
6      <primerNombre>John</primerNombre>
7
8      <apellidoPaterno>Doe</apellidoPaterno>
9
10     <promedioBateo>0.375</promedioBateo>
11 </jugador>

```

Figura 19.1 | XML que describe la información de un jugador de béisbol.

tener exactamente un *elemento raíz* que contenga a todos los demás elementos. En la figura 19.1, jugador (líneas 5-11) es el elemento raíz.

Algunos lenguajes de marcado basados en XML incluyen XHTML (Lenguaje de marcado de hipertexto extensible: el sustituto de HTML para marcar el contenido Web), MathML (para matemáticas), VoiceXML™ (para voz), CML (Lenguaje de marcado químico: para química) y XBRL (Lenguaje de informes empresariales extensible: para el intercambio de datos financieros). A estos lenguajes de marcado se les conoce como *vocabularios* de XML, y proporcionan los medios para describir tipos específicos de datos, en formas estandarizadas y estructuradas.

En la actualidad se almacenan cantidades masivas de datos en Internet, en una variedad de formatos (por ejemplo: bases de datos, páginas Web, archivos de texto). Con base en las tendencias actuales, es probable que la mayor parte de estos datos, en especial los que se intercambian entre sistemas, pronto tomen la forma de XML. Las organizaciones ven a XML como el futuro de la codificación de los datos. Los grupos de tecnología de información están planeando formas de integrar XML en sus sistemas. Los grupos industriales están desarrollando vocabularios de XML personalizados para la mayoría de las grandes industrias, para permitir que las aplicaciones empresariales basadas en computadora se comuniquen en lenguajes comunes. Por ejemplo, los servicios Web, que veremos en el capítulo 22, permiten que las aplicaciones basadas en Web intercambien datos sin problemas, a través de los protocolos estándar basados en XML.

Es casi un hecho que la siguiente generación de Internet y World Wide Web estará basada en XML, el cual permitirá el desarrollo de aplicaciones basadas en Web más sofisticadas. Como veremos en este capítulo, XML le permite asignar significado a lo que de otra manera estaría constituido por simples piezas aleatorias de datos. Como resultado, los programas pueden “comprender” los datos que manipulan. Por ejemplo, un explorador Web podría ver una dirección postal listada en una página Web de HTML simple como una cadena de caracteres, sin ningún significado real. Sin embargo, en un documento de XML estos datos pueden identificarse con claridad (marcarse) como una dirección. Un programa que utilice el documento puede reconocer estos datos como una dirección y proporcionar vínculos a un mapa de esa ubicación, indicaciones de manejo desde esa ubicación, o cualquier otro tipo de información específica de esa ubicación. De igual forma, una aplicación puede reconocer los nombres de personas, fechas, números ISBN y cualquier otro tipo de datos codificados en XML. Con base en estos datos, la aplicación puede presentar a los usuarios otra información relacionada, ofreciendo una experiencia más completa y significativa para el usuario.

Ver y modificar documentos de XML

Los documentos de XML son altamente portables. Para ver o modificar un documento de XML (que es un archivo de texto con la extensión de archivo *.xml*) no se requiere software especial, aunque existen muchas herramientas de software, y con frecuencia surgen nuevos productos en el mercado que hacen más conveniente el desarrollo de aplicaciones basadas en XML. Cualquier editor de texto con soporte para los caracteres del código ASCII/Unicode puede abrir documentos de XML para verlos y editarlos. Además, la mayoría de los exploradores Web pueden desplegar documentos en XML con un formato que facilite ver la estructura del XML. En la sección 19.3 demostramos esto mediante el uso de Internet Explorer. Una característica importante de XML es que tanto los humanos como las máquinas pueden leerlo.

Procesamiento de documentos de XML

Para procesar un documento de XML se requiere un software llamado *analizador de XML* (o *procesador de XML*). Un analizador hace que los datos del documento estén disponibles para las aplicaciones. Mientras lee el contenido de un documento XML, el analizador comprueba que el documento cumpla con las reglas de sintaxis especificadas por la Recomendación de XML del consorcio W3C (www.w3.org/XML). La sintaxis de XML requiere un solo elemento raíz, una etiqueta inicial y una etiqueta final para cada elemento, y etiquetas propiamente anidadas (es decir, la etiqueta final para un elemento anidado debe aparecer antes de la etiqueta final del elemento de cierre). Lo que es más, XML es sensible a mayúsculas o minúsculas, por lo que debe utilizarse la capitalización apropiada en los elementos. Un documento que se conforma a esta sintaxis es un *documento de XML bien formado*, y es sintácticamente correcto. En la sección 19.3 presentaremos la sintaxis fundamental de XML. Si un analizador de XML puede procesar un documento de XML con éxito, ese documento está bien formado. Los analizadores pueden proporcionar acceso a los datos codificados en XML sólo en documentos bien formados.

A menudo, los analizadores de XML están integrados en software como Visual Studio, o están disponibles para descargarse a través de Internet. Algunos analizadores populares son *Microsoft XML Core Services (MSXML)*, *Xerces* de la Fundación de software Apache (xml.apache.org) y el software de código fuente abierto *Expat XML Parser* (expat.sourceforge.net). En este capítulo utilizaremos MSXML.

Validación de documentos de XML

Un documento de XML puede hacer referencia de manera opcional a una *Definición de tipo de documento (DTD)* o a un *esquema* que defina la estructura apropiada del documento de XML. Cuando un documento de XML hace referencia a una DTD o a un esquema, algunos analizadores (conocidos como *analizadores de validación*) pueden leer la DTD/esquema y comprobar que el documento de XML cumpla con la estructura definida por la DTD/esquema. Si el documento de XML se conforma a la DTD/esquema (es decir, que el documento tenga la estructura apropiada), es *válido*. Por ejemplo, si en la figura 19.1 hacemos referencia a una DTD que especifica que un elemento jugador debe tener los elementos `primerNombre`, `apellidoPaterno` y `promedioBateo`, y después omitimos el elemento `apellidoPaterno` (línea 8 en la figura 19.1), el documento de XML `jugador.xml` sería inválido. No obstante, este documento aún estaría bien formado, ya que cumple con la sintaxis apropiada de XML (es decir, tiene un elemento raíz, y cada elemento tiene una etiqueta inicial y una etiqueta final). Por definición, un documento de XML válido está bien formado. Los analizadores que no pueden comprobar la conformidad del documento con DTDs o esquemas son *analizadores no validadores* (sólo determinan si un documento de XML está bien formado, no si es válido).

En las secciones 19.5 y 19.6 hablaremos sobre la validación, las DTDs y los esquemas, así como de las diferencias clave entre estos dos tipos de especificaciones de estructura. Por ahora, sólo tome en cuenta que los esquemas son documentos de XML en sí, mientras que las DTDs no lo son. Como aprenderá en la sección 19.6, esta diferencia presenta varias ventajas en cuanto al uso de esquemas en vez de DTDs.



Observación de ingeniería de software 19.1

Las DTDs y los esquemas son esenciales para las transacciones de negocio a negocio (B2B) y los sistemas de misión crítica. Validar los documentos de XML asegura que sistemas distintos puedan manipular datos estructurados en formas estandarizadas, y evita los errores producidos por datos faltantes o mal formados.

Formato y manipulación de documentos de XML

Los documentos de XML sólo contienen datos y no instrucciones de formato, por lo que las aplicaciones que procesan documentos de XML deben decidir cómo manipular o mostrar los datos de cada documento. Por ejemplo, un PDA (asistente personal digital) puede representar a un documento de XML de manera distinta a como lo representarían un teléfono inalámbrico o una computadora de escritorio. Usted puede usar el *Lenguaje de hojas de estilos extensible (XSL)* para especificar instrucciones de representación para distintas plataformas. En la sección 19.7 hablaremos sobre el XSL.

Los programas de procesamiento de XML pueden también buscar, ordenar y manipular datos de XML mediante el uso de tecnologías tales como XSL. Algunas otras tecnologías relacionadas con XML son XPath (Lenguaje de rutas XML: un lenguaje para acceder a las partes de un documento XML), XSL-FO (Objetos de formato XML: un vocabulario de XML utilizado para describir el formato de los documentos) y XSLT (Transformaciones

XSL: un lenguaje para transformar documentos de XML en otros documentos). En la sección 19.7 presentaremos el XSLT. También presentaremos a XPath en la sección 19.7, y después hablaremos sobre este lenguaje con más detalle en la sección 19.8.

19.3 Estructuración de datos

En esta sección y a lo largo de este capítulo, crearemos nuestro propio marcado de XML. Este lenguaje nos permite describir datos con precisión, en un formato bien estructurado.

Marcado de XML para un artículo

En la figura 19.2 presentamos un documento de XML que marca un artículo simple mediante el uso de XML. Los números de línea que se muestran son para fines de referencia solamente, por lo que no forman parte del documento de XML.

Este documento comienza con una *declaración XML* (línea 1), la cual identifica al documento como un documento de XML. El *atributo version* especifica la versión de XML a la cual se conforma el documento. El estándar actual de XML es la versión 1.0. Aunque el consorcio W3C publicó la especificación de la versión 1.1 en febrero de 2004, esta versión más reciente aún no cuenta con un soporte amplio. El consorcio W3C podría seguir publicando nuevas versiones a medida que XML evolucione, para cumplir con los requerimientos de distintos campos.



Tip de portabilidad 19.1

Los documentos deben incluir la declaración XML para identificar la versión utilizada de XML. Podríamos asumir que un documento sin declaración XML se conforma a la versión más reciente de XML; pero si no es así, podrían producirse errores.



Error común de programación 19.1

Es un error colocar caracteres de espacio en blanco antes de la declaración XML.

Los comentarios de XML (líneas 2-3), que comienzan con `<!--` y terminan con `-->`, pueden colocarse casi en cualquier parte de un documento XML. Los comentarios de XML pueden abarcar varias líneas; no se requiere una etiqueta final en cada línea. La etiqueta final puede aparecer en una línea subsiguiente, siempre y cuando haya

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.2: articulo.xml -->
3  <!-- Artículo estructurado con XML -->
4
5  <articulo>
6    <titulo>XML simple</titulo>
7
8    <fecha>Mayo 5, 2005</fecha>
9
10   <autor>
11     <primerNombre>John</primerNombre>
12     <apellidoPaterno>Doe</apellidoPaterno>
13   </autor>
14
15   <resumen>XML es bastante sencillo.</resumen>
16
17   <contenido>
18     En este capítulo, presentamos una amplia variedad de ejemplos que utilizan XML.
19   </contenido>
20 </articulo>

```

Figura 19.2 | XML utilizado para marcar un artículo.

sólo una etiqueta final (--) para cada etiqueta inicial (<!--). Al igual que en un programa en C#, los comentarios en XML se utilizan para fines de documentación. La línea 4 está en blanco. Al igual que en un programa en C#, las líneas en blanco, los espacios en blanco y la sangría se utilizan en XML para mejorar la legibilidad. Más adelante veremos que, por lo general, los analizadores de XML ignoran las líneas en blanco.



Error común de programación 19.2

En un documento de XML, cada etiqueta inicial debe tener una etiqueta final asociada; si se omite cualquiera de las etiquetas se produce un error. Pronto aprenderá cómo se detectan dichos errores.



Error común de programación 19.3

XML es sensible a mayúsculas o minúsculas. Utilizar distintas combinaciones de mayúsculas/minúsculas para los nombres de la etiqueta inicial y la etiqueta final del mismo elemento es un error de sintaxis.

En la figura 19.2, artículo (líneas 5-20) es el elemento raíz. Las líneas que van antes del elemento raíz (líneas 1-4) son el **prólogo** de XML. En un prólogo, la declaración XML debe aparecer antes de los comentarios y de cualquier otro tipo de marcado.

Los elementos que utilizamos en el ejemplo no provienen de algún lenguaje de marcado específico. En vez de ello, elegimos los nombres de los elementos y la estructura de marcado para describir nuestros datos específicos de la mejor manera posible. Usted puede inventar elementos para marcar sus datos. Por ejemplo, el elemento título (línea 6) contiene texto que describe el título del artículo (por ejemplo, XML simple). De manera similar, los elementos fecha (línea 8), autor (líneas 10-13), primerNombre (línea 11), apellidoPaterno (línea 12), resumen (línea 15) y contenido (líneas 17-19) contienen texto que describe la fecha, el autor, el primer nombre del autor, su apellido, un resumen y el contenido del documento, respectivamente. Los nombres de los elementos de XML pueden tener cualquier longitud y pueden contener letras, dígitos, guiones bajos, guiones y puntos. No obstante, deben empezar con una letra o un guión bajo, y no deben empezar con “xml” en ninguna combinación de mayúsculas o minúsculas (por ejemplo: XML, Xml, xMl), ya que su uso está reservado para los estándares XML.



Error común de programación 19.4

Usar un carácter de espacio en blanco en un elemento de XML es un error.



Buena práctica de programación 19.1

Los nombres de los elementos de XML deben tener un significado que puedan entender los humanos y no deben utilizar abreviaciones.

Los elementos de XML se **anidan** para formar jerarquías, con el elemento raíz en la parte superior de la jerarquía. Esto permite a los autores de documentos crear relaciones tipo padre/hijo entre los datos. Por ejemplo, los elementos título, fecha, autor, resumen y contenido se encuentran anidados dentro de artículo. Los elementos primerNombre y apellidoPaterno se encuentran anidados dentro de autor. La figura 19.21 muestra la jerarquía de la figura 19.2.



Error común de programación 19.5

Anidar etiquetas de XML en forma inapropiada es un error de sintaxis. Por ejemplo, <x><y>hola</x></y> es un error, ya que la etiqueta </y> debe ir antes de la etiqueta </x>.

Cualquier elemento que contenga otros elementos (por ejemplo, artículo o autor) es un **elemento contenedor**. Los elementos contenedores también se conocen como **elementos padres**. Los elementos anidados dentro de un elemento contenedor son **elementos hijos** (o hijos) de ese elemento contenedor.

Ver un documento de XML en Internet Explorer

El documento de XML en la figura 19.2 es tan sólo un archivo de texto llamado `articulo.xml`. Este documento no contiene información de formato para el artículo. Esto se debe a que XML es una tecnología para describir la estructura de los datos. El formato y la visualización de datos de un documento XML son cuestiones específicas de cada aplicación. Por ejemplo, cuando el usuario carga `articulo.xml` en Internet Explorer (IE), MSXML (Microsoft XML Core Services) analiza y visualiza los datos del documento. Internet Explorer utiliza una *hoja de estilo* integrada para dar formato a los datos. Observe que el formato resultante de los datos (figura 19.3) es similar al formato del listado en la figura 19.2. En la sección 19.7 le mostraremos cómo crear hojas de estilo para transformar sus datos XML en varios formatos adecuados para visualizarlos.

Observe los signos menos (-) y más (+) en las capturas de pantalla de la figura 19.3. Aunque estos símbolos no forman parte del documento de XML, Internet Explorer los coloca enseguida de cada elemento contenedor. Un signo menos indica que Internet Explorer está mostrando los elementos hijos del elemento contenedor. Al hacer clic en el signo menos enseguida de un elemento, se contrae ese elemento (es decir, Internet Explorer oculta los hijos del elemento contenedor y sustituye el signo menos con un signo más). Por el contrario, al hacer clic en el signo más enseguida de un elemento, expande ese elemento (es decir, Internet Explorer muestra los hijos del elemento y sustituye el signo más por un signo menos). Este comportamiento es similar a la acción de ver la estructura de directorios mediante el Explorador de Windows. De hecho, con frecuencia la estructura de un directorio se modela como una serie de estructuras tipo árbol, en donde la *raíz* de un árbol representa a la letra de una unidad (por ejemplo, C:) y los *nodos* en el árbol representan a los directorios. A menudo, los analizadores guardan los datos XML como estructuras de árbol para facilitar su manipulación de una manera eficiente, como veremos en la sección 19.8.

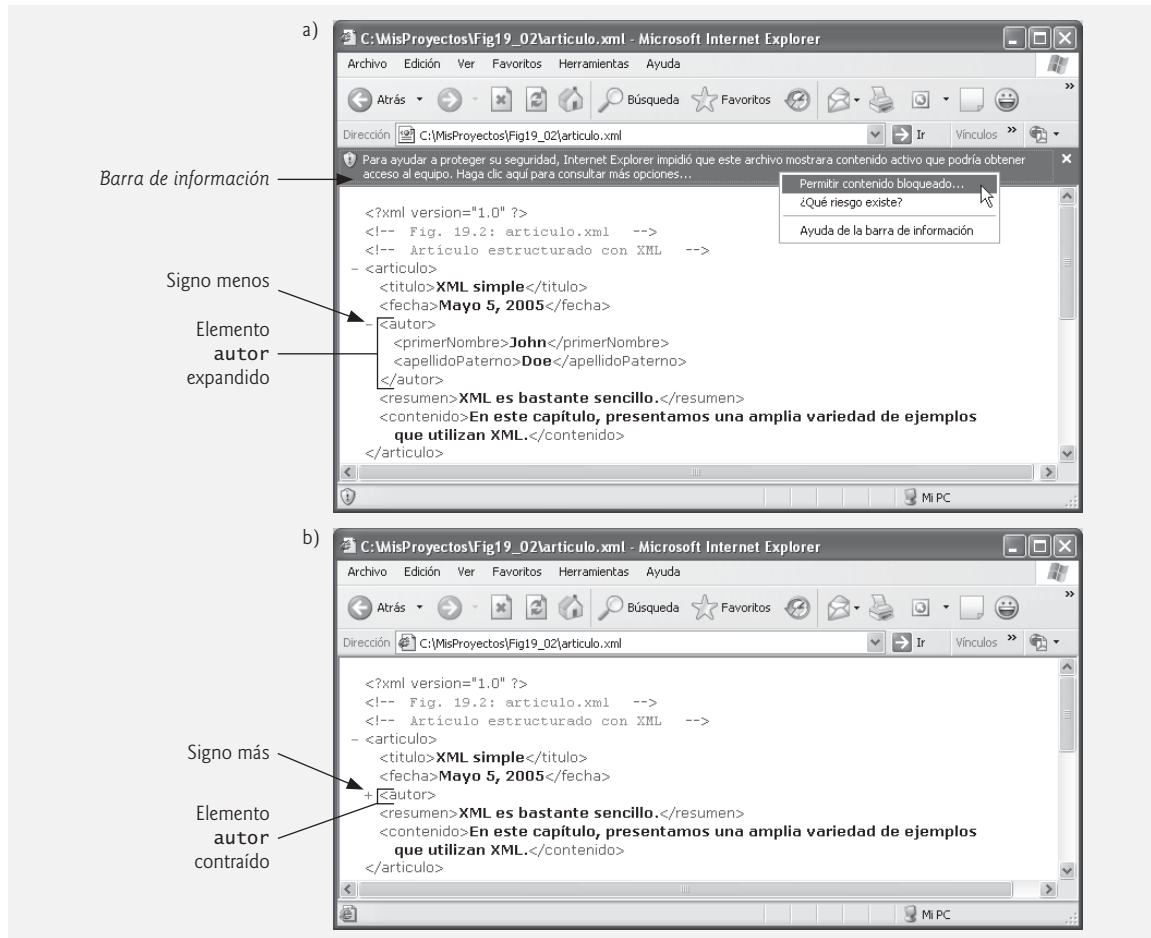


Figura 19.3 | El archivo `articulo.xml` mostrado por Internet Explorer.

[Nota: en Windows XP Service Pack 2, Internet Explorer muestra de manera predeterminada todos los elementos de XML en vista expandida, por lo que si hacemos clic en el signo menos (figura 19.3(a)) no pasa nada. Así que, de manera predeterminada, Windows no podrá contraer el elemento. Para habilitar esta funcionalidad, haga clic con el botón derecho del ratón en la *Barra de información* justo debajo del campo **Dirección** y seleccione **Permitir contenido bloqueado....** Después haga clic en el botón **Sí** en la ventana contextual que aparezca.]

Marcado de XML para una carta de negocios

Ahora que hemos visto un documento simple de XML, examinemos un documento de XML más complejo para marcar una carta de negocios (figura 19.4). De nuevo, empezamos el documento con la declaración XML (línea 1), que declara la versión de XML a la que se conforma el documento.

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.4: cart a.xml -->
3  <!-- Carta de negocios marcada como XML -->
4
5  <!DOCTYPE carta SYSTEM "carta.dtd">
6
7  <carta>
8      <contacto tipo = "remitente">
9          <nombre>Jane Doe</nombre>
10         <direccion1>Box 12345</direccion1>
11         <direccion2>15 Any Ave.</direccion2>
12         <ciudad>Othertown</ciudad>
13         <estado>Otherstate</estado>
14         <cp>67890</cp>
15         <telefono>555-4321</telefono>
16         <bandera genero = "F" />
17     </contacto>
18
19     <contacto tipo = "destinatario">
20         <nombre>John Doe</nombre>
21         <direccion1>123 Main St.</direccion1>
22         <direccion2></direccion2>
23         <ciudad>Anytown</ciudad>
24         <estado>Anystate</estado>
25         <cp>12345</cp>
26         <telefono>555-1234</telefono>
27         <bandera genero = "M" />
28     </contacto>
29
30     <saludo>Estimado señor:</saludo>
31
32     <parrafo>Es nuestro privilegio informarle acerca de nuestra nueva base de datos
33         administrada con XML. Este nuevo sistema le permite reducir la
34         carga en su servidor de lista de inventario, al hacer que la máquina cliente
35         realice el trabajo de ordenar y filtrar los datos.
36     </parrafo>
37
38     <parrafo>Por favor visite nuestro sitio Web para ver la disponibilidad
39         y los precios.
40     </parrafo>
41
42     <cierre>Sinceramente,</cierre>
43     <firma>Ms. Jane Doe</firma>
44 </carta>

```

Figura 19.4 | Carta de negocios marcada como XML.

La línea 5 especifica que este documento de XML hace referencia a una DTD. En la sección 19.2 vimos que las DTDs definen la estructura de los datos para un documento de XML. Por ejemplo, una DTD especifica los elementos y las relaciones padre-hijo entre los elementos que se permiten en un documento de XML.



Tip de prevención de errores 19.1

No se requiere que un documento de XML haga referencia a una DTD, pero los analizadores de XML con validación pueden usar una DTD para asegurarse de que el documento tenga la estructura apropiada.



Tip de portabilidad 19.2

Validar un documento de XML ayuda a garantizar que los desarrolladores independientes intercambien datos en una forma estandarizada que se conforme a la DTD.

La referencia DTD (línea 5) contiene tres elementos, el nombre del elemento raíz que especifica la DTD (carta); la palabra clave **SYSTEM** (que indica una *DTD externa*: una DTD que se declara en un archivo separado, a diferencia de una DTD que se declara en forma local, dentro del mismo archivo); y el nombre y la ubicación de la DTD (es decir, carta.dtd en el directorio actual). Por lo general, los nombres de archivo de los documentos DTD terminan con la extensión **.dtd**. En la sección 19.5 hablaremos sobre las DTDs y sobre carta.dtd con detalle.

Existen varias herramientas (muchas de las cuales son gratuitas) que validan documentos, comparándolos con DTDs y esquemas (que veremos en las secciones 19.5 y 19.6, respectivamente). El *Validador de XML* de Microsoft está disponible sin costo del vínculo **Download Sample** en

msdn.microsoft.com/archive/en-us/samples/internet/xml/xml_validator/default.asp

Este validador puede validar documentos XML, comparándolos con DTDs y con Esquemas. Para instalarlo, ejecute el archivo descargado xml_validator.exe y siga los pasos para completar la instalación. Una vez que la instalación tenga éxito, abra en IE el archivo validate_js.htm, ubicado en el directorio de instalación de XML Validator, para validar sus documentos de XML. Nosotros instalamos el XML Validator en C:\XMLValidator (figura 19.5). La salida (figura 19.6) muestra los resultados de validar el documento usando el XML Validator de Microsoft. En el sitio www.w3.org/XML/Schema podrá obtener una lista de las herramientas de validación adicionales.

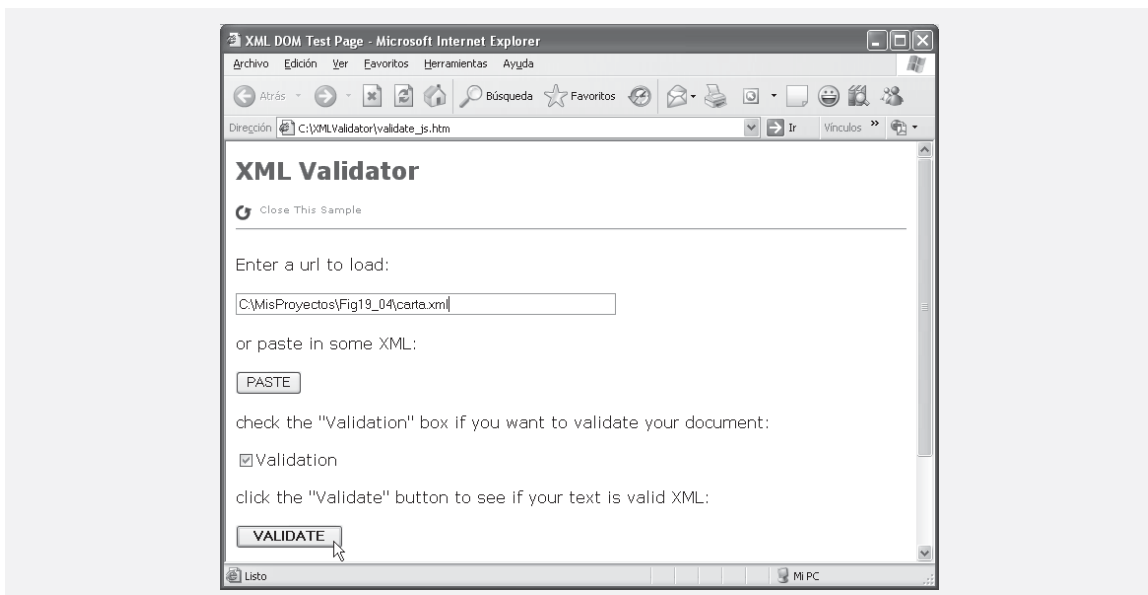


Figura 19.5 | Validación de un documento de XML con el XML Validator de Microsoft.

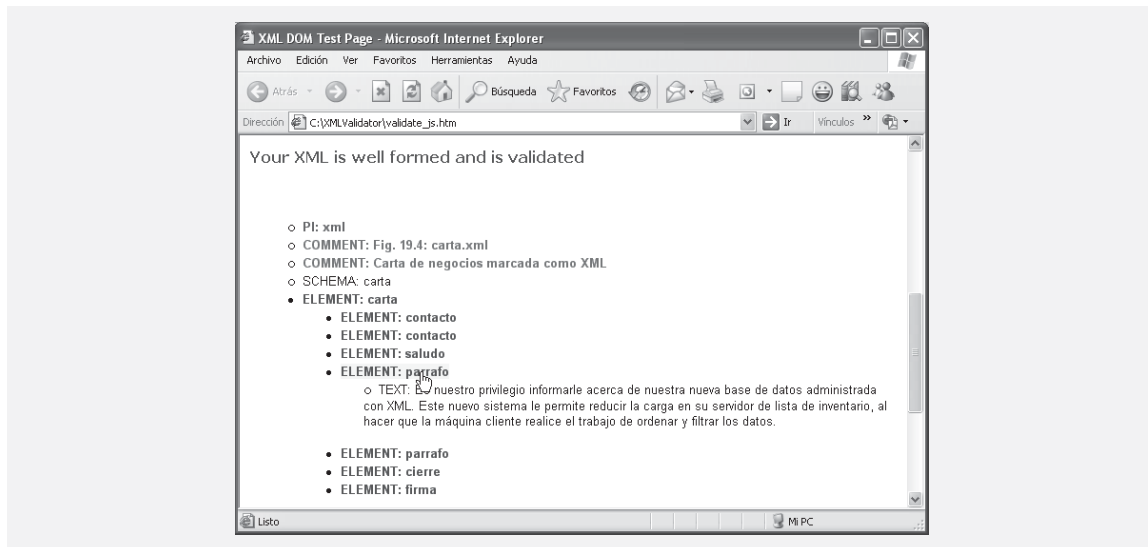


Figura 19.6 | Resultado de la validación, usando el XML Validator de Microsoft.

El elemento raíz *carta* (líneas 7-44 de la figura 19.4) contiene los elementos hijos *contacto*, *contacto*, *saludo*, *parrafo*, *parrafo*, *cierre* y *firma*. Además de colocarse entre etiquetas, los datos también pueden colocarse en **atributos**: pares nombre-valor que aparecen dentro de los signos `<` y `>` de las etiquetas iniciales. Los elementos pueden tener cualquier número de atributos (separados por espacios) en sus marcas iniciales. El primer elemento *contacto* (líneas 8-17) tiene un atributo llamado *tipo*, con el *valor de atributo* "remitente", que indica que este elemento *contacto* identifica al remitente de la carta. El segundo elemento *contacto* (líneas 19-28) tiene el atributo *tipo* con el valor "destinatario", el cual indica que este elemento *contacto* identifica al destinatario de la carta. Al igual que los nombres de los elementos, los nombres de los atributos son sensibles al uso de mayúsculas o minúsculas, pueden tener cualquier longitud, pueden contener letras, dígitos, guiones bajos, guiones y puntos, y deben empezar con una letra o con un carácter de guión bajo. Un elemento *contacto* almacena varios elementos de información acerca de un contacto, como el nombre del contacto (representado por el elemento *nombre*), la dirección (representada por los elementos *direccion1*, *direccion2*, *ciudad*, *estado* y *cp*), el número telefónico (representado por el elemento *telefono* y el género (representado por el atributo *genero* del elemento *bandera*). El elemento *saludo* (línea 30) marca el saludo de la carta. Las líneas 32-40 marcan el cuerpo de la carta, usando dos elementos *parrafo*. Los elementos *cierre* (línea 42) y *firma* (línea 43) marcan el enunciado de cierre y la "firma" del autor, respectivamente.



Error común de programación 19.6

Si no se encierran los valores de los atributos entre comillas dobles (") o sencillas ('), se produce un error de sintaxis.

La línea 16 presenta el *elemento vacío bandera*. Un elemento vacío no tiene contenido. En vez de ello, un elemento vacío algunas veces contiene datos en forma de atributos. El elemento vacío *bandera* contiene un atributo que indica el género del contacto (representado por el elemento padre *contacto*). Los autores de documentos pueden cerrar cualquier elemento vacío, ya sea colocando una barra diagonal inmediatamente después del signo `<`, como se muestra en la línea 16, o colocando explícitamente una etiqueta final, como en la línea 22:

```
<direccion2></direccion2>
```

Observe que el elemento *direccion2* en la línea 22 está vacío, ya que no hay segunda parte para la dirección de este contacto. Sin embargo, debemos incluir este elemento para estar en conformidad con las reglas estructurales especificadas en la DTD del documento de XML: *carta.dtd* (que presentaremos en la sección 19.5). Esta DTD especifica que cada elemento *contacto* debe tener un elemento hijo *direccion2* (aun si está vacío). En la sección 19.5 aprenderá cómo las DTDs indican que ciertos elementos son requeridos, mientras que otros son opcionales.

19.4 Espacios de nombres de XML

XML permite a los autores de documentos crear elementos personalizados. Esta extensibilidad puede producir *conflictos de nombres* entre los elementos de un documento XML en el que cada elemento tiene el mismo nombre. Por ejemplo, podemos usar el elemento `libro` para marcar datos acerca de una publicación Deitel. Un coleccionista de estampillas podría utilizar el elemento `libro` para marcar datos acerca de un libro de estampillas. Si utilizáramos ambos elementos en el mismo documento, se produciría un conflicto de nombres, y se nos dificultaría la labor de determinar qué tipo de datos contiene cada elemento.

Un *espacio de nombres* de XML es una colección de nombres de elementos y atributos. Al igual que los espacios de nombres en C#, los espacios de nombres en XML proporcionan los medios para que los autores de documentos hagan referencias sin ambigüedades (es decir, que eviten los conflictos) a los elementos que tengan el mismo nombre. Por ejemplo,

```
<tema>Matemáticas</tema>
```

y

```
<tema>Cardiología</tema>
```

utilizan el elemento `tema` para marcar datos. En el primer caso, el `tema` es algo que uno estudia en la escuela, mientras que en el segundo caso, el `tema` es un campo de la medicina. Los espacios de nombres pueden diferenciar estos dos elementos `tema`. Por ejemplo:

```
<escuela:tema>Matemáticas</escuela:tema>
```

y

```
<medicina:tema>Cardiología</medicina:tema>
```

Tanto `escuela` como `medicina` son *prefijos de espacios de nombres*. El autor de un documento coloca el prefijo de un espacio de nombres y un signo de dos puntos (:) antes del nombre de un elemento, para especificar el espacio de nombres al que ese elemento pertenece. Los autores de documentos pueden crear sus propios prefijos de espacios de nombres, usando prácticamente cualquier nombre, excepto el prefijo de espacio de nombres reservado `xml`. En las siguientes subsecciones, demostraremos cómo pueden asegurarse los autores de documentos que los espacios de nombres sean únicos.



Error común de programación 19.7

Tratar de crear un prefijo de espacio de nombres llamado `xml`, con cualquier combinación de letras mayúsculas y minúsculas, es un error de sintaxis; el prefijo de espacio de nombres `xml` está reservado para uso interno de XML.

Diferenciación de elementos mediante los espacios de nombres

La figura 19.7 demuestra el uso de los espacios de nombres. En este documento, los espacios de nombres diferencian dos elementos distintos; el elemento `archivo` relacionado con un archivo de texto, y el documento `archivo` relacionado con un archivo de imagen.

Las líneas 6 y 7 utilizan el atributo reservado `xmlns` del espacio de nombres XML para crear dos prefijos de espacio de nombres: `texto` e `imagen`. Cada prefijo de espacio de nombres está vinculado a una serie de caracteres que se conoce como *Identificador de recursos uniforme (URI)*, que identifica al espacio de nombres en forma única. Los autores de documentos crean sus propios prefijos de espacios de nombres y URIs. Un URI es una forma de identificar a un recurso, por lo general en Internet. Dos tipos populares de URI son el *Nombre de recurso uniforme (URN)* y el *Localizador de recursos uniforme (URL)*.

Para asegurar que los espacios de nombres sean únicos, los autores de documentos deben proporcionar URIs únicos. En este ejemplo, utilizamos el texto `urn:deitel:textoInfo` y `urn:deitel:imagenInfo` como URIs. Estos URIs emplean el esquema de URN que se utiliza con frecuencia para identificar espacios


```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.7: namespace.xml -->
3  <!-- Demostración de los espacios de nombres -->
4
5  <text:directory
6      xmlns:text = "urn:deitel:textInfo"
7      xmlns:imagen = "urn:deitel:imageInfo">
8
9      <text:file filename = "book.xml">
10         <text:description>A book list</text:description>
11     </text:file>
12
13     <image:file filename = "funny.jpg">
14         <image:description>A funny picture</image:description>
15         <image:size width = "200" height = "100" />
16     </image:file>
17 </text:directory>

```

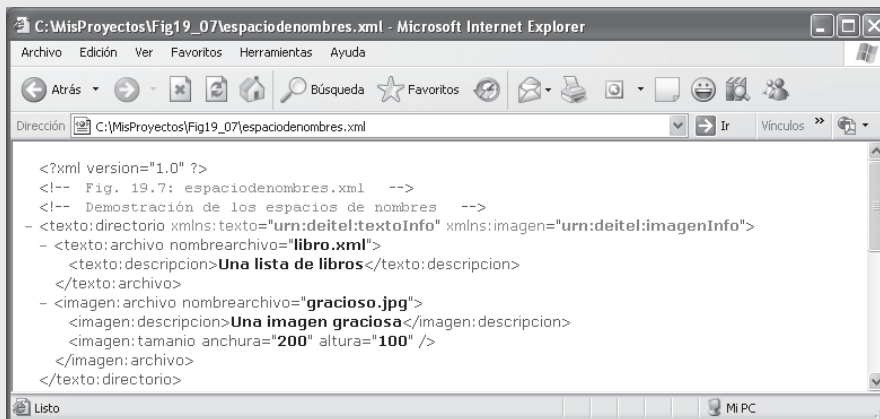


Figura 19.7 | Demostración de los espacios de nombres de XML.

de nombres. Bajo este esquema de nomenclatura, un URI empieza con “urn:”, seguido de una serie única de nombres adicionales, separados por signos de dos puntos.

Otra de las prácticas comunes es utilizar URLs, que especifican la ubicación de un archivo o recurso en Internet. Por ejemplo, www.deitel.com es el URL que identifica a la página inicial del sitio Web de Deitel & Associates. Al utilizar URLs se garantiza que los espacios de nombres sean únicos, ya que se garantiza que los nombres de dominio son únicos. Por ejemplo, las líneas 5-7 podrían volver a escribirse así:

```

<text:directorío
    xmlns:text = "http://www.deitel.com/xmlns-texto"
    xmlns:imagen = "http://www.deitel.com/xmlns-imagen">

```

en donde los URLs relacionados con el nombre de dominio de Deitel & Associates, Inc. sirven como URIs para identificar a los nombres de espacio `texto` e `imagen`. El analizador no visita estos URLs, ni éstos tienen que hacer referencia a páginas Web reales. Simplemente representan una serie única de caracteres utilizados para diferenciar los nombres de los URIs. De hecho, cualquier cadena puede representar a un espacio de nombres. Por ejemplo, nuestro URI del espacio de nombres `imagen` podría ser `hgjfkdl1sa4556`, en cuyo caso la asignación de nuestro prefijo sería

```
xmlns:imagen = "hgjfkdl1sa4556"
```

Las líneas 9-11 utilizan el prefijo de espacio de nombres `text` para los elementos `archivo` y `descripcion`. Observe que las etiquetas finales también deben especificar el texto del prefijo de espacio de nombres. Las líneas 13-16 aplican el prefijo de espacio de nombres `imagen` a los elementos `archivo`, `descripcion` y `tamaño`. Observe que los atributos no requieren prefijos de espacios de nombres (aunque pueden tenerlos), puesto que cada atributo ya forma parte de un elemento que especifica el prefijo del espacio de nombres. Por ejemplo, el atributo `nombrearchivo` (línea 9) forma parte implícitamente del espacio de nombres `texto`, ya que su elemento (es decir, `archivo`) especifica el prefijo de espacio de nombres `texto`.

Especificación de un espacio de nombres predeterminado

Para eliminar la necesidad de colocar los prefijos de espacios de nombres en cada elemento, los autores de documentos pueden especificar un *espacio de nombres predeterminado* para un elemento y sus hijos. La figura 19.8 demuestra el uso de un espacio de nombres predeterminado (`urn:deitel:textoInfo`) para el elemento `directorio`.

La línea 5 define un espacio de nombres predeterminado, usando el atributo `xmlns` con un URI como su valor. Una vez que definimos este espacio de nombres predeterminado, los elementos hijos que pertenecen al espacio de nombres no necesitan calificarse mediante un prefijo de espacio de nombres. Por ende, el elemento `archivo` (líneas 8-10) está en el espacio de nombres predeterminado `urn:deitel:textoInfo`. Compare esto con las líneas 8-10 de la figura 19.7, en donde teníamos que agregar el prefijo del espacio de nombres `texto` a los nombres de los elementos `archivo` y `descripcion`.

El espacio de nombres predeterminado se aplica al elemento `directorio` y a todos los elementos que no están calificados con un prefijo de espacio de nombres. No obstante, podemos usar un prefijo de espacio de

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.8: espaciodenombrespredeterminado.xml -->
3  <!-- Uso de espacios de nombres predeterminados -->
4
5  <directorio xmlns = "urn:deitel:textoInfo"
6    xmlns:imagen = "urn:deitel:imagenInfo">
7
8    <archivo nombrearchivo = "libro.xml">
9      <descripcion>Una lista de libros</descripcion>
10    </archivo>
11
12    <imagen:archivo nombrearchivo = "gracioso.jpg">
13      <imagen:descripcion>Una imagen graciosa</imagen:descripcion>
14      <imagen:tamano anchura = "200" altura = "100"/>
15    </imagen:archivo>
16  </directorio>

```

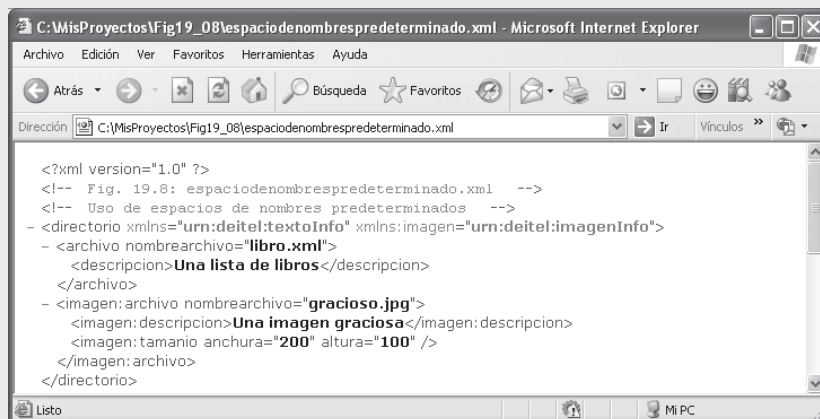


Figura 19.8 | Demostración de un espacio de nombres predeterminado.

nombres para especificar un espacio de nombres distinto para elementos específicos. Por ejemplo, el elemento `archivo` en las líneas 12-15 incluye el prefijo de espacio de nombres `imagen`, lo cual indica que este elemento está en el espacio de nombres `urn:deitel:imagenInfo`, y no en el espacio de nombres predeterminado.

Espacios de nombres en vocabularios de XML

Los lenguajes basados en XML, tales como XML Schema (sección 19.6), el Lenguaje de hojas de estilo extensible (XSL) (sección 19.7) y BizTalk (www.microsoft.com/biztalk), a menudo utilizan espacios de nombres para identificar a sus elementos. Cada uno de estos vocabularios define elementos de propósito especial que se agrupan en espacios de nombres. Estos espacios de nombres ayudan a evitar conflictos de nombres entre los elementos predefinidos y los elementos definidos por el usuario.

19.5 Definiciones de tipo de documento (DTDs)

Las Definiciones de tipo de documento (DTDs) son uno de dos tipos principales de documentos que podemos usar para especificar la estructura de los documentos de XML. La sección 19.6 presenta los documentos de esquemas XML del W3C, que proporcionan un método mejorado para especificar la estructura de los documentos de XML.



Observación de ingeniería de software 19.2

Los documentos de XML pueden tener muchas estructuras diferentes, y por esta razón una aplicación no puede estar segura si un documento específico que recibe está completo, ordenado en forma apropiada, o si le faltan datos. Las DTDs y los esquemas (sección 19.6) resuelven este problema, al proporcionar una manera extensible de describir la estructura de un documento de XML. Las aplicaciones deben utilizar DTDs o esquemas para confirmar si los documentos de XML son válidos.



Observación de ingeniería de software 19.3

Muchas organizaciones e individuos están creando DTDs y esquemas para un amplio rango de aplicaciones. Estas colecciones (conocidas como repositorios) están disponibles para descargarse sin costo de varios sitios Web (por ejemplo, www.xml.org, www.oasis-open.org).

Creación de una Definición de tipo de documento

La figura 19.4 presenta una carta de negocios simple, marcada con XML. Recuerde que la línea 5 de `carta.xml` hace referencia a una DTD: `carta.dtd` (figura 19.9). Esta DTD especifica los tipos de los elementos y atributos de la carta de negocios, y sus relaciones entre ellos.

Una DTD describe la estructura de un documento de XML y permite que un analizador de XML verifique si un documento de XML es válido (es decir, si sus elementos contienen los atributos apropiados y aparecen en la secuencia apropiada). Las DTDs permiten a los usuarios comprobar la estructura de un documento e intercambiar datos en un formato estandarizado. Una DTD expresa el conjunto de reglas para la estructura de un documento mediante el uso de una gramática EBNF (Forma de Backus-Naur extendida). [Nota: por lo general, las gramáticas EBNF se utilizan para definir lenguajes de programación. Para obtener más información acerca de las gramáticas EBNF, visite los sitios en wikipedia.org/wiki/EBNF o www.garshol.priv.no/download/text/bnf.html.]

```

1  <!-- Fig. 19.9: carta.dtd -->
2  <!-- Documento DTD para carta.xml -->
3
4  <!ELEMENT carta ( contacto+, saludo, parrafo+,
5     cierre, firma )>
6
7  <!ELEMENT contacto ( nombre, direccion1, direccion2, ciudad, estado,
8     cp, telefono, bandera )>
9  <!ATTLIST contacto tipo CDATA #IMPLIED>

```

Figura 19.9 | Definición de tipo de documento (DTD) para una carta de negocios. (Parte 1 de 2).

```

10
11 <!ELEMENT nombre ( #PCDATA )>
12 <!ELEMENT direccion1 ( #PCDATA )>
13 <!ELEMENT direccion2 ( #PCDATA )>
14 <!ELEMENT ciudad ( #PCDATA )>
15 <!ELEMENT estado ( #PCDATA )>
16 <!ELEMENT cp ( #PCDATA )>
17 <!ELEMENT telefono ( #PCDATA )>
18 <!ELEMENT bandera EMPTY>
19 <!ATTLIST bandera genero (M | F) "M">
20
21 <!ELEMENT saludo ( #PCDATA )>
22 <!ELEMENT cierre ( #PCDATA )>
23 <!ELEMENT parrafo ( #PCDATA )>
24 <!ELEMENT firma ( #PCDATA )>

```

Figura 19.9 | Definición de tipo de documento (DTD) para una carta de negocios. (Parte 2 de 2).



Error común de programación 19.8

Para los documentos validados con DTDs, cualquier documento que utilice elementos, atributos o relaciones que no estén definidos explícitamente mediante una DTD es un documento inválido.

Definición de elementos en una DTD

La *declaración de tipo de elemento* **ELEMENT** en las líneas 4-5 define las reglas para el elemento carta. En este caso, carta contiene uno o más elementos contacto, un elemento saludo, uno o más elementos parrafo, un elemento cierre y un elemento firma, en esa secuencia. El *indicador de ocurrencia de signo positivo* (+) especifica que la DTD permite una o más ocurrencias de un elemento. Otros indicadores de ocurrencia son el *asterisco* (*), que indica un elemento opcional que puede ocurrir cero o más veces, y el *signo de interrogación* (?), que indica que un elemento opcional puede ocurrir cuando menos una vez (es decir, cero ocurrencias o una). Si un elemento no tiene un indicador de ocurrencia, la DTD permite exactamente una ocurrencia de este elemento.

La declaración del tipo del elemento contacto (líneas 7-8) especifica que un elemento contacto contiene los elementos hijos nombre, direccion1, direccion2, ciudad, estado, cp, telefono y bandera, en ese orden. La DTD requiere sólo una ocurrencia de cada uno de estos elementos.

Definición de atributos en una DTD

La línea 9 utiliza la *declaración de lista de atributos* **ATTLIST** para definir un atributo llamado tipo, para el elemento contacto. La palabra clave **#IMPLIED** especifica que, si el analizador encuentra un elemento contacto sin un atributo tipo, puede elegir un valor arbitrario para el atributo, o puede ignorarlo. De cualquier forma, el documento aún sería válido (si el resto del documento es válido); si se omite un atributo tipo no se invalidará el documento. Otras palabras clave que pueden usarse en lugar de **#IMPLIED** en una declaración **ATTLIST** son **#REQUIRED** y **#FIXED**. La palabra clave **#REQUIRED** especifica que el atributo debe estar presente en el elemento, y la palabra clave **#FIXED** especifica que el atributo (si está presente) debe tener el valor fijo dado. Por ejemplo,

```
<!ATTLIST direccion cp CDATA #FIXED "01757">
```

indica que el atributo cp (si está presente en el elemento direccion) debe tener el valor 01757 para que el documento sea válido. Si el atributo no está presente, entonces el analizador, de manera predeterminada, usará el valor fijo que especifique la declaración **ATTLIST**.

Comparación entre datos tipo carácter y datos tipo carácter analizados

La palabra clave **CDATA** (línea 9) especifica que el atributo type contiene *datos de carácter* (es decir, una cadena). Un analizador pasará tales datos a una aplicación sin modificarlos.



Observación de ingeniería de software 19.4

La sintaxis de las DTDs no cuenta con un mecanismo para describir el tipo de datos de un elemento (o atributo). Por ejemplo, una DTD no puede especificar que un elemento o atributo específico puede contener sólo datos enteros.

La palabra clave **#PCDATA** (línea 11) especifica que un elemento (por ejemplo, nombre) puede contener **datos tipo carácter analizados** (es decir, datos procesados por un analizador de XML). Los elementos con datos tipo carácter analizados no pueden contener caracteres de marcado, como los signos menor que (<), mayor que (>) o &. El autor del documento debe sustituir cualquier carácter de marcado en un elemento **#PCDATA** con la correspondiente **referencia a entidad de carácter** de ese carácter. Por ejemplo, la referencia a entidad de carácter < debe utilizarse en lugar del símbolo menor que (<), y la referencia a entidad de carácter > debe utilizarse en lugar del símbolo mayor que (>). El autor de un documento que desee utilizar un símbolo & literal, debe utilizar en su defecto la referencia a entidad &. Los datos tipo carácter analizados pueden contener símbolos &, sólo para insertar entidades. En el apéndice H podrá ver una lista de otras referencias a entidades de carácter.



Error común de programación 19.9

El uso de caracteres de marcado (por ejemplo: <, > y &) en datos tipo carácter analizados es un error. Use en su defecto referencias a entidades de carácter (por ejemplo: <, > y &).

Definición de elementos vacíos en una DTD

La línea 18 define un elemento vacío llamado bandera. La palabra clave **EMPTY** especifica que el elemento no contiene datos entre sus etiquetas inicial y final. Por lo general, los elementos vacíos describen datos mediante atributos. Por ejemplo, los datos de bandera aparecen en su atributo **genero** (línea 19). La línea 19 especifica que el valor del atributo **genero** debe ser uno de los valores enumerados (M o F) encerrados entre paréntesis y delimitados por una barra vertical (|) que significa “o”. Observe que la línea 19 también indica que **genero** tiene un valor predeterminado de M.

Comparación entre documentos bien formados y documentos válidos

En la sección 19.3 demostramos cómo usar el Microsoft XML Validator para validar un documento de XML, comparándolo con su DTD especificada. La validación reveló que el documento de XML *carta.xml* (figura 19.4) está bien formado y es válido; se conforma a *carta.dtd* (figura 19.9). Recuerde que un documento bien formado es sintácticamente correcto (es decir, cada etiqueta inicial tiene su correspondiente etiqueta final, el documento sólo contiene un elemento raíz, etc.), y un documento válido contiene los elementos apropiados con los atributos apropiados, en la secuencia apropiada. Un documento de XML no puede ser válido a menos que esté bien formado.

Cuando un documento no se conforma a una DTD o a un esquema, Microsoft XML Validator muestra un mensaje de error. Por ejemplo, la DTD en la figura 19.10 indica que un elemento **contacto** debe contener el elemento hijo **nombre**. Un documento que omita a este elemento hijo sigue estando bien formado, pero no es válido. En dicho escenario, Microsoft XML Validator muestra el mensaje de error que aparece en la figura 19.10.

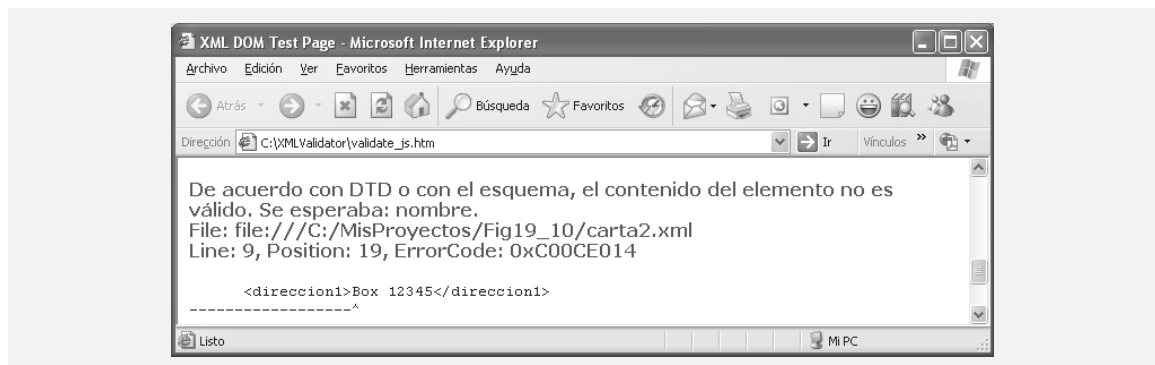


Figura 19.10 | XML Validator muestra un mensaje de error.

19.6 Documentos de esquemas XML del W3C

En esta sección presentaremos los esquemas para especificar la estructura de un documento de XML y validar documentos de XML. Muchos desarrolladores en la comunidad de XML creen que las DTDs no son lo bastante flexibles como para satisfacer las necesidades de programación actuales. Por ejemplo, las DTDs carecen de un medio para indicar qué tipo específico de datos (por ejemplo: numérico, texto) puede contener un elemento, y las DTDs no son documentos de XML en sí. Éstas y otras limitaciones han conducido al desarrollo de los esquemas.

A diferencia de las DTDs, los esquemas no utilizan gramática EBNF. En vez de ello, utilizan sintaxis de XML y, de hecho, son documentos de XML que los programas pueden manipular. Al igual que las DTDs, los analizadores de validación utilizan los esquemas para validar documentos.

En esta sección, nos concentraremos en el vocabulario *XML Schema* del W3C (observe la letra “S” mayúscula en “Schema”). Utilizaremos el término XML Schema durante el resto del capítulo, cada vez que hagamos referencia al vocabulario XML Schema del W3C. Para obtener la información más reciente acerca de XML Schema, visite el sitio www.w3.org/XML/Schema. Si desea ver tutoriales sobre los conceptos de XML Schema que estén más allá de lo que presentamos aquí, visite el sitio www.w3schools.com/schema/default.asp.

Una DTD describe la estructura de un documento de XML, no el contenido de sus elementos. Por ejemplo,

```
<cantidad>5</cantidad>
```

contiene datos tipo carácter. Si el documento que contiene el elemento `cantidad` hace referencia a una DTD, un analizador de XML puede validar el documento para confirmar que, ciertamente, el contenido de este elemento es PCDATA. Sin embargo, el analizador no puede validar que el contenido sea numérico; las DTDs no proporcionan esta capacidad. Así que, desafortunadamente, el analizador también considera que

```
<cantidad>hola</cantidad>
```

es válido. Una aplicación que utilice el documento de XML que contenga este marcado debe comprobar que los datos en el elemento `cantidad` sean numéricos, y debe realizar las acciones apropiadas en caso de que no sea así.

XML Schema permite a los autores de esquemas especificar que los datos del elemento `cantidad` deben ser numéricos o, más específicamente, de tipo entero. Un analizador que valide el documento de XML y lo compare con este esquema puede determinar que el 5 está en conformidad, y que `hola` no lo está. Un documento de XML que se conforma a un documento de esquema es *válido para el esquema*, y un documento que no se conforma es *inválido para el esquema*. Los esquemas son documentos de XML y, por lo tanto, también deben ser válidos.

Validación con base en un documento de esquema de XML

La figura 19.11 muestra un documento de XML válido para el esquema, llamado `libro.xml`, y la figura 19.12 muestra el documento de XML Schema pertinente (`libro.xsd`) que define la estructura para `libro.xml`. Por convención, los esquemas utilizan la extensión **.xsd**. Nosotros usamos un validador de esquemas XSD proporcionado por Microsoft en la dirección

apps.getdotnet.com/xmltools/xsdvalidator

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.11: libro.xml -->
3  <!-- Lista de libros marcada como XML -->
4
5  <deitel:libros xmlns:deitel = "http://www.deitel.com/listalibros">
6    <libro>
7      <titulo>Visual Basic 2005 Cómo programar, 3/e</titulo>
8    </libro>
```

Figura 19.11 | Documento de XML válido para el esquema, que describe una lista de libros. (Parte 1 de 2).

```

9
10 <libro>
11   <titulo>Visual C# 2005 Cómo programar</titulo>
12 </libro>
13
14 <libro>
15   <titulo>Java Cómo programar, 6/e</titulo>
16 </libro>
17
18 <libro>
19   <titulo>C++ Cómo programar, 5/e</titulo>
20 </libro>
21
22 <libro>
23   <titulo>Internet y World Wide Web Cómo programar, 3/e</titulo>
24 </libro>
25 </deitel:libros>

```

Figura 19.11 | Documento de XML válido para el esquema, que describe una lista de libros. (Parte 2 de 2).

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 19.12: libro.xsd -->
3 <!-- Documento simple de XML Schema del W3C -->
4
5 <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6   xmlns:deitel = "http://www.deitel.com/listalibros"
7   targetNamespace = "http://www.deitel.com/listalibros">
8
9   <element name = "libros" type = "deitel:TipoLibros"/>
10
11   <complexType name = "TipoLibros">
12     <sequence>
13       <element name = "libro" type = "deitel:TipoLibroSencillo"
14         minOccurs = "1" maxOccurs = "unbounded"/>
15     </sequence>
16   </complexType>
17
18   <complexType name = "TipoLibroSencillo">
19     <sequence>
20       <element name = "title" type = "string"/>
21     </sequence>
22   </complexType>
23 </schema>

```

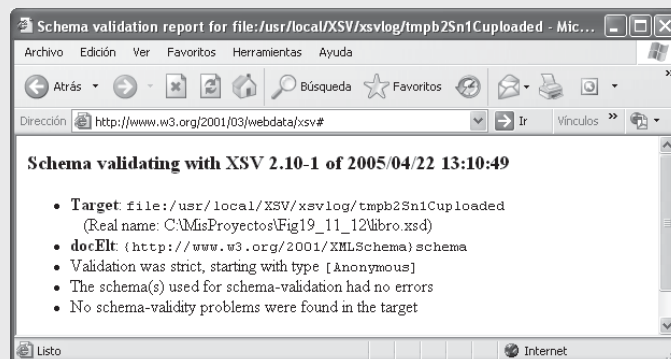


Figura 19.12 | Documento de XML Schema para libro.xml.

para asegurarnos que el documento de XML de la figura 19.11 se conforme al esquema de la figura 19.12. Para validar el documento del esquema en sí (es decir, `libro.xsd`) y producir la salida que se muestra en la figura 19.12, utilizamos un XSV en línea (Validador de XML Schema) proporcionado por el consorcio W3C en la dirección

www.w3.org/2001/03/webdata/xsv

Estas herramientas son gratuitas y cumplen con las especificaciones del consorcio W3C, en relación con los esquemas de XML y la validación de los mismos. La sección 19.12 lista varios validadores en línea de XML Schema.

La figura 19.11 contiene marcado que describe varios libros de Deitel. El elemento `libros` (línea 5) tiene el prefijo de espacio de nombres `deitel`, que indica que el elemento `libros` forma parte del espacio de nombres <http://www.deitel.com/listalibros>. Observe que declaramos el prefijo de espacio de nombres `deitel` en la línea 5.

Creación de un documento de XML Schema

La figura 19.12 presenta el documento de XML Schema que especifica la estructura de `libro.xml` (figura 19.11). Este documento define un lenguaje basado en XML (es decir, un vocabulario) para escribir documentos de XML acerca de colecciones de libros. El esquema define los elementos, atributos y relaciones padre-hijo que un documento de este tipo puede (o debe) incluir. Este esquema también especifica el tipo de datos que pueden contener estos elementos y atributos.

El elemento raíz **`schema`** (figura 19.12, líneas 5-23) contiene elementos que definen la estructura de un documento de XML como `libro.xml`. La línea 5 especifica como espacio de nombres predeterminado el URI del espacio de nombres de XML Schema del W3C: **`http://www.w3.org/2001/XMLSchema`**. Este espacio de nombres contiene elementos predefinidos (por ejemplo: el elemento raíz `schema`) que conforman el vocabulario XML Schema: el lenguaje utilizado para escribir un documento de XML Schema.



Tip de portabilidad 19.3

Los autores de esquemas de XML del W3C especifican el URI `http://www.w3.org/2001/XMLSchema` cuando hacen referencia al espacio de nombres XML Schema. Este espacio de nombres contiene elementos predefinidos que conforman el vocabulario XML Schema. Al especificar este URI, se asegura que las herramientas de validación identifiquen en forma correcta los elementos de XML Schema, y que no los confundan con los elementos definidos por los autores de documentos.

La línea 6 enlaza el URI <http://www.deitel.com/listalibros> al prefijo de espacio de nombres `deitel`. Como veremos en unos momentos, el esquema utiliza este espacio de nombres para diferenciar los nombres que nosotros creamos, de los nombres que forman parte del espacio de nombres XML Schema. La línea 7 también especifica a <http://www.deitel.com/listalibros> como el espacio de nombres de destino (**`targetNamespace`**) del esquema. Este atributo identifica el espacio de nombres del vocabulario de XML que define este esquema. Observe que el valor para `targetNamespace` de `libro.xsd` es el mismo que el espacio de nombres referenciado en la línea 5 de `libro.xml` (figura 19.11). Esto es lo que “conecta” al documento de XML con el esquema que define su estructura. Cuando un validador de esquemas de XML analice los archivos `libro.xml` y a `libro.xsd`, reconocerá que `libro.xml` utiliza elementos y atributos del espacio de nombres <http://www.deitel.com/listalibros>. El validador también reconocerá que este espacio de nombres es el que está definido en `libro.xsd` (es decir, el elemento `targetNamespace` del esquema). Por ende, el validador sabe en dónde buscar las reglas estructurales para los elementos y atributos utilizados en `libro.xml`.

Definición de un elemento en XML Schema

En XML Schema, la etiqueta **`element`** (línea 9) define a un elemento que se incluirá en un documento de XML que se conforme a ese esquema. En otras palabras, **`element`** especifica los *elementos* actuales que pueden usarse para marcar datos. La línea 9 define el elemento `libros`, que utilizamos como elemento raíz en `libro.xml` (figura 19.11). Los atributos **`nombre`** y **`tipo`** especifican el nombre y tipo de datos del elemento, respectivamente. El tipo de datos de un elemento indica los datos que puede contener. Los posibles tipos de datos incluyen los tipos definidos por XML Schema (por ejemplo: `string`, `double`) y los tipos definidos por el usuario (por ejemplo:

TipoLibros, que se define en las líneas 11-16). La figura 19.13 lista varios de los muchos tipos integrados de XML Schema. Para obtener una lista completa de tipos integrados, consulte la sección 3 de la especificación en www.w3.org/TR/xmlschema-2.

En este ejemplo, libros se define como un elemento con el tipo de datos deitel:TipoLibros (línea 9). TipoLibros es un tipo definido por el usuario (líneas 11-16) en el espacio de nombres <http://www.deitel.com/listalibros> y, por lo tanto, debe tener el prefijo de espacio de nombres deitel. No es un tipo de datos existente de XML Schema.

Tipo(s) de datos de XML Schema	Descripción	Rangos o estructuras	Ejemplos
string	Una cadena de caracteres.		"ho1a"
boolean	Verdadero o falso.	true, false	true
decimal	Un número decimal.	$i * (10^n)$, en donde i es un entero y n es un entero menor o igual a cero.	5, -12, -45.78
float	Un número de punto flotante.	$m * (2^e)$, en donde m es un entero cuyo valor absoluto es menor que 2^{24} y e es un entero en el rango de -149 a 104. Más tres números adicionales: infinito positivo, infinito negativo y "no es un número" (NaN).	0, 12, -109.375, NaN
double	Un número de punto flotante.	$m * (2^e)$, en donde m es un entero cuyo valor absoluto es menor que 2^{53} y e es un entero en el rango de -1075 a 970. Más tres números adicionales: infinito positivo, infinito negativo y "no es un número" (NaN).	0, 12, -109.375, NaN
long	Un número entero.	-9223372036854775808 a 9223372036854775807, inclusive	1234567890, -1234567890
int	Un número entero.	-2147483648 a 2147483647, inclusive	1234567890, -1234567890
short	Un número entero.	-32768 a 32767, inclusive	12, -345
date	Una fecha que consiste en el año, mes y día.	aaaa-mm con un valor dd opcional y una zona horaria opcional, en donde aaaa es de cuatro dígitos; mm y dd son de dos dígitos.	2005-05-10
time	Una hora que consiste en horas, minutos y segundos.	hh:mm:ss con una zona horaria opcional, en donde hh, mm y ss son de dos dígitos.	16:30:25-05:00

Figura 19.13 | Algunos tipos de datos de XML Schema.

En XML Schema existen dos categorías de tipos de datos: *tipos simples* y *tipos complejos*. La única diferencia entre estos dos tipos es que los tipos simples no pueden contener atributos o elementos hijos, y los tipos complejos sí.

Un tipo definido por el usuario que contiene atributos o elementos hijos debe definirse como tipo complejo. Las líneas 11-16 utilizan el elemento **complexType** para definir a TipoLibros como un tipo complejo que tiene un elemento hijo llamado libro. El elemento sequence (líneas 12-15) nos permite especificar el orden secuencial en el que deben aparecer los elementos hijos. El elemento element (líneas 13-14) anidado dentro del elemento complexType indica que un elemento TipoLibros (por ejemplo, libros) puede contener elementos hijos llamados libro de tipo deitel:TipoLibroSencillo (definido en las líneas 18-22). El atributo **minOccurs** (línea 14), que tiene el valor 1, especifica que los elementos de tipo TipoLibros deben contener como mínimo un elemento libro. El atributo **maxOccurs** (línea 14), con el valor **unbounded**, especifica que los elementos de tipo TipoLibros pueden tener cualquier número de elementos hijos libro.

Las líneas 18-22 definen el tipo complejo TipoLibroSencillo. Un elemento de este tipo contiene un elemento hijo llamado titulo. La línea 20 define el elemento title como del tipo simple string. Recuerde que los elementos de un tipo simple no pueden contener atributos o elementos hijos. La etiqueta final schema (</schema>, línea 23) declara el fin del documento de XML Schema.

Análisis detallado de los tipos en XML Schema

Cada elemento en XML Schema tiene un tipo. Los tipos pueden ser los tipos integrados proporcionados por XML Schema (figura 19.13), o los tipos definidos por el usuario (por ejemplo, TipoLibroSencillo en la figura 19.12).

Cada tipo simple define una *restricción* sobre un tipo definido por XML Schema, o una restricción sobre un tipo definido por el usuario. Las restricciones limitan los posibles valores que puede guardar un elemento.

Los tipos complejos se dividen en dos grupos: los que tienen *contenido simple* y los que tienen *contenido complejo*. Ambos pueden contener atributos, pero sólo el contenido complejo puede contener elementos hijos. Los tipos complejos con contenido simple deben extender o restringir a algún otro tipo existente. Los tipos complejos con contenido complejo no tienen esta limitación. En el siguiente ejemplo demostraremos los tipos complejos con cada tipo de contenido.

El documento de esquema en la figura 19.14 crea tanto tipos simples como tipos complejos. El documento de XML en la figura 19.15 (laptop.xml) sigue la estructura definida en la figura 19.14 para describir partes de una computadora laptop. Un documento como laptop.xml que se conforma a un esquema, se conoce como *documento de instancia XML*: el documento es una instancia (es decir, ejemplo) del esquema.

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.14: computadora.xsd -->
3  <!-- Documento de XML Schema del W3C -->
4
5  <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6    xmlns:computadora = "http://www.deitel.com/computadora"
7    targetNamespace = "http://www.deitel.com/computadora">
8
9    <simpleType name = "gigahertz">
10      <restriction base = "decimal">
11        <minInclusive value = "2.1"/>
12      </restriction>
13    </simpleType>
14
15    <complexType name = "CPU">
16      <simpleContent>
17        <extension base = "string">
18          <attribute name = "modelo" type = "string"/>
19        </extension>
20      </simpleContent>
21    </complexType>

```

Figura 19.14 | Documento de XML Schema que define tipos simples y complejos. (Parte 1 de 2).

```

22
23     <complexType name = "portatil">
24         <all>
25             <element name = "procesador" type = "computadora:CPU"/>
26             <element name = "monitor" type = "int"/>
27             <element name = "VelocidadCPU" type = "computadora:gigahertz"/>
28             <element name = "RAM" type = "int"/>
29         </all>
30         <attribute name = "fabricante" type = "string"/>
31     </complexType>
32
33     <element name = "laptop" type = "computadora:portatil"/>
34 </schema>

```

Figura 19.14 | Documento de XML Schema que define tipos simples y complejos. (Parte 2 de 2).

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.15: laptop.xml -->
3  <!-- Componentes de una laptop marcados como XML -->
4
5  <computadora:laptop xmlns:computadora = "http://www.deitel.com/computadora"
6      fabricante = "IBM">
7
8      <procesador modelo = "Centrino">Intel</procesador>
9      <monitor>17</monitor>
10     <VelocidadCPU>2.4</VelocidadCPU>
11     <RAM>256</RAM>
12 </computadora:laptop>

```

Figura 19.15 | Documento de XML que usa el elemento `laptop` definido en `computadora.xsd`.

La línea 5 declara el espacio de nombres predeterminado para que sea el espacio de nombres estándar de XML Schema; se asumirá que cualquier elemento sin prefijo se encuentra en el espacio de nombres de XML Schema. La línea 6 enlaza el prefijo de espacio de nombres `computadora` al espacio de nombres `http://www.deitel.com/computadora`. La línea 7 identifica a este espacio de nombres como el elemento `targetNamespace`: el espacio de nombres que está siendo definido por el documento actual de XML Schema.

Para diseñar los elementos de XML para describir computadoras `laptop`, primero creamos un tipo simple en las líneas 9-13 usando el elemento ***simpleType***. Nombramos a este tipo simple `gigahertz` debido a que se utilizará para describir la velocidad del reloj del procesador en gigahertz. Los tipos simples son restricciones de un tipo, que por lo general se denomina ***tipo base***. Para este tipo simple, la línea 10 declara el tipo base como `decimal`, y restringimos el valor para que sea cuando menos 2.1, mediante el uso del elemento ***minInclusive*** en la línea 11.

A continuación declaramos un tipo complejo llamado `CPU`, que tiene contenido simple (***simpleContent***) (líneas 16-20). Recuerde que un tipo complejo con contenido simple puede tener atributos, pero no elementos hijos. Además recuerde que los tipos complejos con contenido simple deben extender o restringir algún tipo de XML Schema o algún tipo definido por el usuario. El elemento ***extension*** con el atributo ***base*** (línea 17) establece el tipo base a `string`. En este tipo complejo, extendemos el tipo base `string` con un atributo. El elemento ***attribute*** (línea 18) proporciona al tipo complejo un atributo de tipo `string` llamado `modelo`. Por ende, un elemento de tipo `CPU` debe contener texto `string` (debido a que el tipo base es `string`) y puede contener un atributo `modelo` que también es de tipo `string`.

Por último, definimos el tipo `portatil`, que es un tipo complejo con contenido complejo (líneas 23-31). Dichos tipos pueden tener elementos hijos y atributos. El elemento ***all*** (líneas 24-29) encierra elementos que deben incluirse una vez cada uno en el correspondiente documento de instancia de XML. Estos elementos pueden incluirse en cualquier orden. Este tipo complejo guarda cuatro elementos: `procesador`, `monitor`, `VelocidadCPU`

y RAM. Estos elementos reciben los tipos CPU, int, gigahertz e int, respectivamente. Al utilizar los tipos CPU y gigahertz, debemos incluir el prefijo de espacio de nombres computadora, ya que estos tipos definidos por el usuario son parte del espacio de nombres computadora (<http://www.deitel.com/computadora>): el espacio de nombres definido en el documento actual (línea 7). Además, portatil contiene un atributo definido en la línea 30. El elemento attribute indica que los elementos de tipo portatil contienen un atributo de tipo string, llamado fabricante.

La línea 33 declara el elemento actual que utiliza los tres tipos definidos en el esquema. El elemento se llama laptop y es de tipo portatil. Debemos utilizar el prefijo de espacio de nombres computadora en frente de portatil.

Ahora hemos creado un elemento llamado laptop que contiene los elementos hijos procesador, monitor, VelocidadCPU y RAM, y un atributo llamado fabricante. La figura 19.15 utiliza el elemento laptop definido en el esquema computadora.xsd. Una vez más, utilizamos un validador de esquemas XSD en línea para asegurar que este documento de instancia de XML se adhiera a las reglas estructurales del esquema.

La línea 5 declara el prefijo de espacio de nombres computadora. El elemento laptop requiere este prefijo, ya que forma parte del espacio de nombres <http://www.deitel.com/computadora>. La línea 6 establece el atributo fabricante de la laptop, y las líneas 8-11 utilizan los elementos definidos en el esquema para describir las características de la laptop.

En esta sección presentamos los documentos del XML Schema del W3C para definir la estructura de documentos de XML, y validamos los documentos de instancia de XML, comparándolos con esquemas mediante el uso de un validador de esquemas XSD en línea. La sección 19.9 demuestra cómo validar los documentos de XML mediante la programación, comparándolos con esquemas mediante el uso de clases del .NET Framework. Esto le permite asegurar que un programa en C# manipule sólo documentos válidos; manipular un documento inválido al que le falten piezas requeridas de datos podría provocar errores en el programa.

19.7 (Opcional) Lenguaje de hojas de estilos extensible y transformaciones XSL

Los documentos en *Lenguaje de hojas de estilos extensible (XSL)* especifican la forma en que los programas deben representar los datos de un documento de XML. XSL es un grupo de tres tecnologías: *XSL-FO (Objetos de formato XSL)*, *XPath (Lenguaje de rutas XML)* y *XSLT (Transformaciones XSL)*. XSL-FO es un vocabulario para especificar el formato, y XPath es un lenguaje basado en cadenas, de expresiones utilizadas por XML y muchas de sus tecnologías relacionadas, para localizar en forma efectiva y eficiente las estructuras y los datos (como los elementos y atributos específicos) en documentos de XML.

La tercera porción de XSL, las Transformaciones XSL (XSLT), es una tecnología para transformar documentos de XML en otros documentos; es decir, transformar la estructura de los datos del documento de XML en otra estructura. XSLT proporciona elementos que definen reglas para transformar un documento de XML y producir un documento de XML distinto. Esto es útil cuando queremos utilizar datos en varias aplicaciones o plataformas, cada una de las cuales puede estar diseñada para trabajar con documentos escritos en un vocabulario específico. Por ejemplo, XSLT nos permite convertir un documento de XML simple en un documento *XHTML (Lenguaje de marcado de hipertexto extensible)* que presente los datos del documento de XML (o un subconjunto de los datos) en un formato para visualizarlo en un explorador Web. (En la figura 19.16 encontrará una vista de ejemplo tipo “antes” y “después” de dicha transformación.) XHTML es la recomendación técnica que sustituye a HTML para marcar el contenido Web. Para obtener más información sobre XHTML, consulte el Apéndice F, Introducción al lenguaje XHTML: parte 1 y el Apéndice G, Introducción a XHTML: parte 2, y visite el sitio www.w3.org.

Para transformar un documento de XML mediante el uso de XSLT se requieren dos estructuras tipo árbol: el *árbol de origen* (es decir, el documento de XML que se va a transformar) y el *árbol de resultados* (es decir, el documento de XML que se va a crear). XPath se utiliza para localizar partes del documento del árbol de origen que coinciden con *plantillas* definidas en una *hoja de estilos XSL*. Cuando ocurre una coincidencia (es decir, cuando un nodo coincide con una plantilla), se ejecuta la plantilla coincidente y agrega su resultado al árbol de resultados. Cuando ya no hay coincidencias, XSLT ha transformado el árbol de origen en el árbol de resultados. XSLT no analiza cada nodo del árbol de origen; navega en forma selectiva por el árbol de origen, utilizando los atributos select y match de XPath. Para que XSLT funcione, el árbol de origen debe estar estructurado en forma apropiada. Los esquemas, las DTDs y los analizadores de validación pueden validar la estructura de un documento antes de utilizar XPath y XSLTs.

Un ejemplo simple de XSL

La figura 19.16 lista un documento de XML que describe varios deportes. La salida muestra el resultado de la transformación (especificada en la plantilla XSLT de la figura 19.17), representado por Internet Explorer 6.

Para realizar transformaciones, se requiere un procesador de XSLT. Algunos de los procesadores de XSLT populares son: MSXML de Microsoft y *Xalan 2* de la Fundación de software Apache (xml.apache.org). El documento de XML que se muestra en la figura 19.16 se transforma en un documento XHTML mediante MSXML, cuando el documento se carga en Internet Explorer. MSXML es tanto un analizador de XML como un procesador de XSLT.

La línea 2 (figura 19.16) es una *instrucción de procesamiento (PI)* que hace referencia a la hoja de estilos XSL `deportes.xsl` (figura 19.17). Una instrucción de procesamiento se incrusta en un documento de XML y proporciona información específica de la aplicación a cualquier procesador de XML que utilice esa aplicación. En este caso específico, la instrucción de procesamiento especifica la ubicación de un documento XSLT con el que se va a

```

1  <?xml version = "1.0"?>
2  <?xml:stylesheet type = "text/xsl" href = "deportes.xsl"?>
3
4  <!-- Fig. 19.16: deportes.xml -->
5  <!-- Base de datos de deportes -->
6
7  <deportes>
8      <juego id = "783">
9          <nombre>Cricket</nombre>
10
11          <parrafo>
12              Más popular entre las comunidades de naciones.
13          </parrafo>
14      </juego>
15
16      <juego id = "239">
17          <nombre>Béisbol</nombre>
18
19          <parrafo>
20              Más popular en América.
21          </parrafo>
22      </juego>
23
24      <juego id = "418">
25          <nombre>Soccer (Fútbol)</nombre>
26
27          <parrafo>
28              El deporte más popular en el mundo.
29          </parrafo>
30      </juego>
31  </deportes>

```

ID	Deporte	Información
783	Cricket	Más popular entre las comunidades de naciones.
239	Béisbol	Más popular en América.
418	Soccer (Fútbol)	El deporte más popular en el mundo.

Figura 19.16 | Documento de XML que describe varios deportes.

transformar el documento de XML. Los caracteres `<?y ?>` (línea 2, figura 19.16) delimitan una instrucción de procesamiento, la cual consiste en un *destino de la PI* (por ejemplo, `xml:stylesheet`) y un *valor de la PI* (por ejemplo, `type = "text/xsl" href = "deportes.xsl"`). El atributo `type` del valor de la PI especifica que `deportes.xsl` es un archivo tipo `text/xsl` (es decir, un archivo de texto con contenido XSL). El atributo `href` especifica el nombre y la ubicación de la hoja de estilo que se va a aplicar; en este caso, `deportes.xsl` en el directorio actual.



Observación de ingeniería de software 19.5

XSL permite a los autores de documentos separar la presentación de los datos (especificada en documentos de XSL) de su descripción (especificada en documentos de XML).

La figura 19.17 muestra el documento de XSL para transformar los datos estructurados del documento de XML de la figura 19.16 en un documento de XHTML para presentarlo en pantalla. Por convención, los documentos XSL tienen la extensión de archivo `.xsl`.

Las líneas 6-7 empiezan la hoja de estilos XSL con la etiqueta inicial **stylesheet**. El atributo **version** especifica la versión de XSLT a la cual se conforma este documento. La línea 7 enlaza el prefijo de espacio de nombres **xsl** al URI de XSLT del W3C (es decir, `http://www.w3.org/1999/XSL/Transform`).

Las líneas 9-12 usan el elemento **xsl:output** para escribir una declaración de tipo de documento XHTML (DOCTYPE) en el árbol de resultados (es decir, el documento de XML que se va a crear). DOCTYPE identifica a XHTML como el tipo del documento resultante. Al atributo `method` se le asigna `"xml"`, lo cual indica que se va a enviar XML al árbol de resultados. (Recuerde que XHTML es un tipo de XML.) El atributo **omit-xml-declaration** especifica si la transformación debe escribir la declaración XML en el árbol de resultados. En este caso, no queremos omitir la declaración XML, por lo que asignamos a este atributo el valor `"no"`. Los atributos `doctype-system` y `doctype-public` escriben la información DOCTYPE de la DTD en el árbol de resultados.

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.17: deportes.xsl -->
3  <!-- Una transformación XSLT simple -->
4
5  <!-- URI de la hoja de estilo XSL de referencia -->
6  <xsl:stylesheet version = "1.0"
7    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9    <xsl:output method = "xml" omit-xml-declaration = "no"
10      doctype-system =
11        "http://www.w3c.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
12      doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14    <xsl:template match = "/"> <!-- coincidencia con el elemento raíz -->
15
16      <html xmlns = "http://www.w3.org/1999/xhtml">
17        <head>
18          <title>Deportes</title>
19        </head>
20
21        <body>
22          <table border = "1" bgcolor = "wheat">
23            <thead>
24              <tr>
25                <th>ID</th>
26                <th>Deporte</th>
27                <th>Información</th>
28              </tr>
29            </thead>
30

```

Figura 19.17 | XSLT que crea elementos y atributos en un documento de XHTML. (Parte I de 2).


```

31      <!-- inserta cada valor de elemento nombre y párrafo -->
32      <!-- en la fila de una tabla. -->
33      <xsl:for-each select = "/deportes/juego">
34          <tr>
35              <td><xsl:value-of select = "@id"/></td>
36              <td><xsl:value-of select = "nombre"/></td>
37              <td><xsl:value-of select = "parrafo"/></td>
38          </tr>
39      </xsl:for-each>
40  </table>
41 </body>
42 </html>
43
44 </xsl:template>
45 </xsl:stylesheet>

```

Figura 19.17 | XSLT que crea elementos y atributos en un documento de XHTML. (Parte 2 de 2).

XSLT utiliza *plantillas* (es decir, elementos **xsl:template**) para describir cómo transformar nodos específicos del árbol de origen al árbol de resultados. Se aplica una plantilla a los nodos que se especifican en el atributo **match** requerido. La línea 14 utiliza el atributo **match** para seleccionar la *raíz del documento* (es decir, la parte conceptual del documento que contiene el elemento raíz y todo lo que está debajo de éste) de XML de origen (es decir, `deportes.xml`). El carácter / (una barra diagonal) de XPath siempre selecciona la raíz del documento. Recuerde que XPath es un lenguaje basado en cadenas, que se utiliza para localizar partes de un documento de XML con facilidad. En XPath, una barra diagonal específica que estamos usando *direccionamiento absoluto* (es decir, empezamos desde la raíz y definimos rutas hacia abajo por el árbol de origen). En el documento de XML de la figura 19.16, los nodos hijos de la raíz del documento son los dos nodos de instrucción de procesamiento (líneas 1-2), los dos nodos de comentario (líneas 4-5) y el nodo del elemento `deportes` (líneas 7-31). La plantilla en la figura 19.17, línea 14, coincide con un nodo (es decir, el nodo raíz), por lo que ahora se agrega el contenido de la plantilla al árbol de resultados.

El procesador MSXML escribe el XHTML en las líneas 16-29 (figura 19.17) en el árbol de resultados, exactamente como aparece en el documento de XSL. Ahora el árbol de resultados consiste en la definición DOCTYPE y el código XHTML de las líneas 16-29. Las líneas 33-39 utilizan los elementos **xsl:for-each** para iterar a través del documento XML de origen, buscando elementos `juego`. El elemento **xsl:for-each** es similar a la instrucción `foreach` de C#. El atributo **select** es una expresión XPath que especifica los nodos (se le conoce como *conjunto de nodos*) en los que opera el elemento **xsl:for-each**. De nuevo, la primera barra diagonal significa que estamos usando direccionamiento absoluto. La barra diagonal entre `deportes` y `juego` indica que `juego` es un nodo hijo de `deportes`. Por ende, el elemento **xsl:for-each** busca nodos `juego` que son hijos del nodo `deportes`. El documento `deportes.xml` contiene sólo un nodo `deportes`, que también es el nodo raíz del documento. Después de buscar los elementos que coincidan con el criterio de búsqueda, el elemento **xsl:for-each** procesa cada elemento con el código en las líneas 34-38 (estas líneas producen una fila en una tabla, cada vez que se ejecutan) y coloca el resultado de las líneas 34-38 en el árbol de resultados.

La línea 35 utiliza el elemento **value-of** para recuperar el valor del atributo `id` y colocarlo en un elemento `td` en el árbol de resultados. El símbolo `@` de XPath especifica que `id` es un nodo de atributo del nodo de contexto `juego`. Las líneas 36-37 colocan los valores de los elementos `nombre` y `parrafo` en elementos `td` y los insertan en el árbol de resultados. Cuando una expresión de XPath no tiene barra diagonal al principio, utiliza *direccionamiento relativo*. Al omitir la barra diagonal al principio se indica a las instrucciones **xsl:value-of select** que deben buscar elementos `nombre` y `parrafo` que sean hijos del nodo de contexto, no del nodo raíz. Debido a la última selección de la expresión de XPath, el nodo de contexto actual es `juego`, el cual sin duda tiene un atributo `id`, un elemento hijo `nombre` y un elemento hijo `parrafo`.

Uso de XSLT para ordenar y dar formato a los datos

La figura 19.18 presenta un documento de XML (`ordenamiento.xml`) que marca información acerca de un libro. Observe que varios elementos del marcado que describen al libro aparecen fuera de orden (por ejemplo, el

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.18: ordenamiento.xml -->
3  <!-- Documento de XML que contiene información sobre un libro -->
4
5  <?xml:stylesheet type = "text/xsl" href = "ordenamiento.xsl"?>
6
7  <libro isbn = "999-99999-9-X">
8      <titulo>Deitel&apos;s XML Primer</titulo>
9
10     <autor>
11         <primerNombre>Jane</primerNombre>
12         <apellidoPaterno>Blue</apellidoPaterno>
13     </autor>
14
15     <capitulos>
16         <paginasLiminares>
17             <prefacio paginas = "2" />
18             <contenido paginas = "5" />
19             <ilustraciones paginas = "4" />
20         </paginasLiminares>
21
22         <capitulo numero = "3" paginas = "44">Advanced XML</capitulo>
23         <capitulo numero = "2" paginas = "35">Intermediate XML</capitulo>
24         <apendice numero = "B" paginas = "26">Parsers and Tools</apendice>
25         <apendice numero = "A" paginas = "7">Entities</apendice>
26         <capitulo numero = "1" paginas = "28">XML Fundamentals</capitulo>
27     </capitulos>
28
29     <medio tipo = "CD" />
30 </libro>

```

Figura 19.18 | Documento de XML que contiene información sobre un libro.

elemento que describe al capítulo 3 aparece antes del elemento que describe al capítulo 2). Los ordenamos de esta forma a propósito, para demostrar que la hoja de estilos XSL a la que se hace referencia en la línea 5 (`ordenamiento.xsl`) puede ordenar los datos del archivo XML para fines de presentación.

La figura 19.19 presenta un documento de XSL (`ordenamiento.xsl`) para transformar a `ordenamiento.xml` (figura 19.18) en XHTML. Recuerde que un documento de XSL navega por un árbol de origen y crea un árbol de resultados. En este ejemplo, el árbol de origen es XML, y el árbol de salida es XHTML. La línea 14 de la figura 19.19 coincide con el elemento raíz del documento en la figura 19.18. La línea 15 envía una etiqueta inicial `html` al árbol de resultados. El elemento `<xsl:apply-templates/>` (línea 16) especifica que el procesador de XSLT debe aplicar los elementos `xsl:template` definidos en este documento XSL a los hijos del nodo actual (es decir, la raíz del documento). El contenido que resulta al aplicar las plantillas se envía en el elemento `html` que termina en la línea 17. Las líneas 21-84 especifican una plantilla que coincide con el elemento `libro`. La plantilla indica cómo dar formato de XHTML a la información contenida en los elementos `libro` de `ordenamiento.xml` (figura 19.18).

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.19: ordenamiento.xsl -->
3  <!-- Transformación de la información del libro en XHTML -->
4
5  <xsl:stylesheet version = "1.0"
6      xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
7

```

Figura 19.19 | Documento de XSL que transforma a `ordenamiento.xml` en XHTML. (Parte I de 3).


```

8      <!-- escribe la declaración XML y la información DOCTYPE de la DTD -->
9      <xsl:output method = "xml" omit-xml-declaration = "no"
10         doctype-system = "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"
11         doctype-public = "-//W3C//DTD XHTML 1.1//EN"/>
12
13      <!-- coincidencia con la raíz del documento -->
14      <xsl:template match = "/">
15          <html xmlns = "http://www.w3.org/1999/xhtml">
16              <xsl:apply-templates/>
17          </html>
18      </xsl:template>
19
20      <!-- coincidencia con libro -->
21      <xsl:template match = "libro">
22          <head>
23              <title>ISBN <xsl:value-of select = "@isbn"/> -
24                  <xsl:value-of select = "titulo"/></title>
25          </head>
26
27          <body>
28              <h1 style = "color: blue"><xsl:value-of select = "titulo"/></h1>
29              <h2 style = "color: blue">por
30                  <xsl:value-of select = "autor/apellidoPaterno"/>,
31                  <xsl:value-of select = "autor/primerNombre"/></h2>
32
33              <table style = "border-style: groove; background-color: wheat">
34
35                  <xsl:for-each select = "capitulos/paginasLiminares/*">
36                      <tr>
37                          <td style = "text-align: right">
38                              <xsl:value-of select = "name()"/>
39                          </td>
40
41                          <td>
42                              ( <xsl:value-of select = "@paginas"/> páginas )
43                          </td>
44                      </tr>
45                  </xsl:for-each>
46
47                  <xsl:for-each select = "capitulos/capitulo">
48                      <xsl:sort select = "@numero" data-type = "number"
49                          order = "ascending"/>
50                      <tr>
51                          <td style = "text-align: right">
52                              Capitulo <xsl:value-of select = "@numero"/>
53                          </td>
54
55                          <td>
56                              <xsl:value-of select = "text()"/>
57                              ( <xsl:value-of select = "@paginas"/> páginas )
58                          </td>
59                      </tr>
60                  </xsl:for-each>
61
62                  <xsl:for-each select = "capitulos/apendice">
63                      <xsl:sort select = "@numero" data-type = "text"
64                          order = "ascending"/>
65                      <tr>
66                          <td style = "text-align: right">

```

Figura 19.19 | Documento de XSL que transforma a ordenamiento.xml en XHTML. (Parte 2 de 3).

```

67         Apéndice <xsl:value-of select = "@numero"/>
68     </td>
69
70     <td>
71         <xsl:value-of select = "text()"/>
72         ( <xsl:value-of select = "@paginas"/> páginas )
73     </td>
74 </tr>
75 </xsl:for-each>
76 </table>
77
78 <br /><p style = "color: blue">Páginas:
79 <xsl:variable name = "cuentapaginas"
80     select = "sum(capitulos//*[@paginas])"/>
81 <xsl:value-of select = "$cuentapaginas"/>
82 <br />Tipo de medio: <xsl:value-of select = "medio/@tipo"/></p>
83 </body>
84 </xsl:template>
85 </xsl:stylesheet>

```

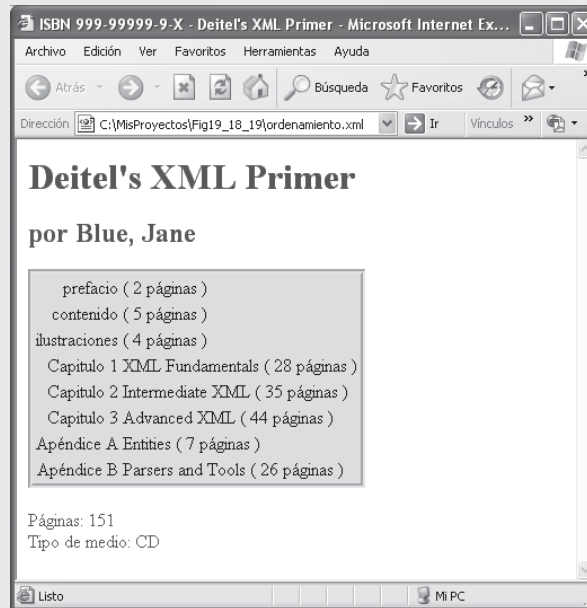


Figura 19.19 | Documento de XSL que transforma a ordenamiento.xml en XHTML. (Parte 3 de 3).

Las líneas 23-24 crean el título para el documento XHTML. Utilizamos el ISBN del libro (del atributo `isbn`) y el contenido del elemento `título` para crear la cadena de texto que aparece en la barra de título de la ventana del explorador (ISBN 999-99999-9-X – Deitel's XML Primer).

La línea 28 crea un elemento de encabezado que contiene el título del libro. Las líneas 29-31 crean un elemento de encabezado que contiene el autor del libro. Debido a que el nodo de contexto (es decir, el nodo actual que se está procesando) es `libro`, la expresión `autor/apellidoPaterno` de XPath selecciona el apellido paterno del autor, y la expresión `autor/primerNombre` selecciona el primer nombre del autor.

La línea 35 selecciona a cada elemento (indicado mediante un asterisco) que sea hijo del elemento `paginas-Liminales`. La línea 38 llama a la *función de establecimiento de nodo* **name** para recuperar el nombre del elemento del nodo actual (por ejemplo, `prefacio`). El nodo actual es el nodo de contexto que se especifica en `xsl:for-each` (línea 35). La línea 42 recupera el valor del atributo `paginas` del nodo actual.

La línea 47 selecciona a cada uno de los elementos capítulo. Las líneas 48-49 utilizan el elemento `<xsl:sort>` para ordenar los capítulos por número, en orden ascendente. El atributo `select` selecciona el valor del atributo `numero` en el nodo de contexto capítulo. El atributo `data-type` con el valor "number" especifica un orden numérico, y el atributo `order` con el valor "ascending" especifica el orden ascendente. El atributo `data-type` también acepta el valor "text" (línea 63), y el atributo `order` también acepta el valor "descending". La línea 56 usa la *función de establecimiento de nodo text* para obtener el texto entre las etiquetas capítulo iniciales y finales (es decir, el nombre del capítulo). La línea 57 recupera el valor del atributo `paginas` del nodo actual. Las líneas 62-75 realizan tareas similares para cada apéndice.

Las líneas 79-80 utilizan una *variable de XSL* para guardar el valor de la cuenta total de páginas del libro, y envían la cuenta de páginas al árbol de resultados. El atributo `name` especifica el nombre de la variable (es decir, `cuentapaginas`) y el atributo `select` asigna un valor a esa variable. La función `sum` (línea 80) calcula el total de los valores para todos los atributos `pagina`. Las dos barras diagonales entre capítulos y * indican un *descenso recursivo*; el procesador MSXML buscará elementos que contengan un atributo llamado `paginas` en todos los nodos descendientes de capítulos. La expresión XPath

```
//*
```

selecciona a todos los nodos en un documento de XML. La línea 81 recupera el valor de la recién creada variable de XSL llamada `cuentapaginas`, colocando un signo de dólar en frente de su nombre.

Resumen de los elementos de las hojas de estilos XSL

Los ejemplos de esta sección utilizaron varios elementos predefinidos de XSL para realizar varias operaciones. La figura 19.20 lista estos elementos y algunos otros que se utilizan con frecuencia. Para obtener más información acerca de estos elementos y de XSL en general, visite el sitio Web www.w3.org/Style/XSL.

Elemento	Descripción
<xsl:apply-templates>	Aplica las plantillas del documento de XSL a los hijos del nodo actual.
<xsl:apply-templates match = "expresión">	Aplica las plantillas del documento de XSL a los hijos de <i>expresión</i> . El valor del atributo <code>match</code> (es decir, <i>expresión</i>) debe ser una expresión de XPath que especifique elementos.
<xsl:template>	Contiene reglas que se aplican cuando coincide un nodo especificado.
<xsl:value-of select = "expresión">	Selecciona el valor de un elemento de XML y lo agrega al árbol de resultados de la transformación. El atributo <code>select</code> requerido contiene una expresión de XPath.
<xsl:for-each select = "expresión">	Aplica una plantilla a cada nodo seleccionado por la expresión de XPath especificada por el atributo <code>select</code> .
<xsl:sort select = "expresión">	Se utiliza como elemento hijo de un elemento <xsl:apply-templates> o <xsl:for-each>. Ordena los nodos seleccionados por el elemento <xsl:apply-templates> o <xsl:for-each>, de manera que los nodos se procesen en orden.
<xsl:output>	Tiene varios atributos para definir el formato (por ejemplo, XML, XHTML), la versión (por ejemplo, 1.0, 2.0), el tipo de documento y el tipo de medio del documento de salida. Esta etiqueta es un elemento de nivel superior; sólo puede usarse como elemento hijo de <code><xsl:stylesheet></code> .
<xsl:copy>	Agrega el nodo actual al árbol de resultados.

Figura 19.20 | Elementos de las hojas de estilos XSL.

En esta sección presentamos el Lenguaje de hojas de estilos extensible (XSL) y mostramos cómo crear transformaciones XSL, para convertir documentos XML de un formato a otro. Le mostramos también cómo transformar documentos XML en documentos XHTML, para visualizarlos en un explorador Web. Recuerde que estas transformaciones las realiza MSXML, el analizador de XML y procesador de XSLT integrado en Internet Explorer. En la mayoría de las aplicaciones empresariales, los documentos de XML se transfieren entre los socios de negocios y se transforman en otros vocabularios de XML mediante la programación. En la sección 19.10 demostraremos cómo realizar transformaciones XSL utilizando la clase `XslCompiledTransform` que proporciona el .NET Framework.

19.8 (Opcional) Modelo de objetos de documento (DOM)

Aunque un documento de XML es un archivo de texto, no es práctico ni eficiente recuperar datos del documento mediante el uso de las técnicas tradicionales de procesamiento de archivos secuenciales, en especial para agregar y eliminar elementos en forma dinámica.

Después de analizar con éxito un documento, algunos analizadores de XML almacenan los datos del documento como estructuras tipo árbol en memoria. La figura 19.21 ilustra la estructura de árbol para el elemento raíz del documento `artículo.xml` que vimos en la figura 19.2. Esta estructura de árbol jerárquica se conoce como *árbol del Modelo de objetos de documento (DOM)*, y a un analizador de XML que crea este tipo de estructura se le conoce como *analizador DOM*. El nombre de cada elemento (por ejemplo: `artículo`, `fecha`, `primerNombre`) se representa mediante un nodo. A un nodo que contiene otros nodos (llamados *nodos hijos* o hijos) se le conoce como *nodo padre* (por ejemplo, `autor`). Un nodo padre puede tener muchos hijos, pero un nodo hijo sólo puede tener un nodo padre. Los nodos que son iguales (por ejemplo, `primerNombre` y `apellidoPaterno`) se llaman *nodos hermanos*. Los *nodos descendientes* de un nodo incluyen a sus hijos, a los hijos de sus hijos, y así en lo sucesivo. Los *nodos ancestros* de un nodo incluyen a su padre, al padre de su padre, y así en lo sucesivo.

El árbol DOM tiene un solo *nodo raíz*, que contiene a todos los demás nodos en el documento. Por ejemplo, el nodo raíz del árbol DOM que representa a `artículo.xml` (figura 19.2) contiene un nodo para la declaración XML (línea 1), dos nodos para los comentarios (líneas 2-3) y un nodo para el elemento raíz `artículo` del documento de XML (línea 5).

Las clases para crear, leer y manipular documentos de XML se encuentran en el espacio de nombres **System.Xml** de C#. Este espacio de nombres también contiene espacios de nombres adicionales, que proporcionan otras operaciones relacionadas con XML.

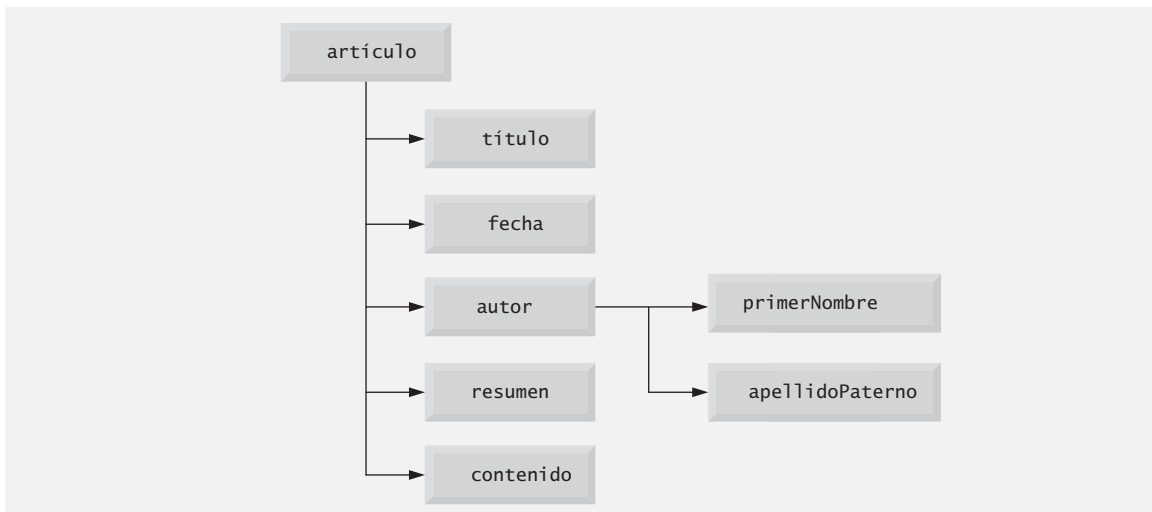


Figura 19.21 | Estructura tipo árbol para el documento `artículo.xml` de la figura 19.2.

Lectura de un documento de XML con un objeto XmlReader

En esta sección presentamos varios ejemplos que utilizan árboles DOM. Nuestro primer ejemplo, el programa en la figura 19.22, carga el documento de XML presentado en la figura 19.2 y muestra sus datos en un cuadro de texto. Este ejemplo utiliza la clase **XmlReader** para iterar a través de cada nodo en el documento de XML.

La línea 5 es una declaración `using` para el espacio de nombres `System.Xml`, el cual contiene las clases de XML que se utilizan en este ejemplo. `XmlReader` es una clase abstracta que define la interfaz para leer documentos de XML. No podemos crear un objeto `XmlReader` en forma directa, sino que debemos invocar al método `static Create` de `XmlReader` para obtener una referencia `XmlReader` (línea 21). Sin embargo, antes de hacerlo debemos preparar un objeto `XmlReaderSettings` que especifique cómo queremos que se comporte el objeto `XmlReader` (línea 20). En este ejemplo, utilizamos la configuración predeterminada de las propiedades de un objeto `XmlReaderSettings`. Más adelante aprenderá a establecer ciertas propiedades de la clase `XmlReaderSettings` para indicar al objeto `XmlReader` que realice la validación, lo cual no hace de manera predeterminada. El método `static Create` recibe como argumentos el nombre del documento de XML a leer y un objeto `XmlReaderSettings`. En este ejemplo, el documento de XML `articulo.xml` (figura 19.2) se abre cuando se invoca el método `Create` en la línea 21. Una vez que se crea el objeto `XmlReader`, se puede leer el contenido del documento de XML mediante la programación.

```

1  // Fig. 19.22: PruebaXmlReader.cs
2  // Lectura de un documento de XML.
3  using System;
4  using System.Windows.Forms;
5  using System.Xml;
6
7  namespace PruebaXmlReader
8  {
9      public partial class PruebaXmlReaderForm : Form
10     {
11         public PruebaXmlReaderForm()
12         {
13             InitializeComponent();
14         } // fin del constructor
15
16         // lee el documento de XML y muestra su contenido
17         private void PruebaXmlReaderForm_Load( object sender, EventArgs e )
18         {
19             // crea el objeto XmlReader
20             XmlReaderSettings configuraciones = new XmlReaderSettings();
21             XmlReader lector = XmlReader.Create( "articulo.xml", configuraciones );
22
23             int profundidad = -1; // la profundidad del árbol es -1, sin sangría
24
25             while ( lector.Read() ) // muestra el contenido de cada nodo
26             {
27                 switch ( lector.NodeType )
28                 {
29                     case XmlNodeType.Element: // Elemento de XML, muestra su nombre
30                         profundidad++; // aumenta la profundidad de los tabuladores
31                         ImprimirTab( profundidad ); // inserta tabuladores
32                         txtSalida.Text += "<" + lector.Name + ">\r\n";
33
34                         // si el elemento está vacío, reduce la profundidad
35                         if ( lector.IsEmptyElement )
36                             profundidad--;
37                         break;
38                     case XmlNodeType.Comment: // Comentario de XML, lo muestra

```

Figura 19.22 | Un objeto `XmlReader` iterando a través de un documento de XML. (Parte I de 2).

```

39      ImprimirTab( profundidad ); // inserta tabuladores
40      txtSalida.Text += "<!--" + lector.Value + "-->\r\n";
41      break;
42      case XmlNodeType.Text: // Texto de XML, lo muestra
43      ImprimirTab( profundidad ); // inserta tabuladores
44      txtSalida.Text += "\t" + lector.Value + "\r\n";
45      break;
46
47      // Declaracion XML la muestra
48      case XmlNodeType.XmlDeclaration:
49      ImprimirTab( profundidad ); // inserta tabuladores
50      txtSalida.Text += "<?" + lector.Name + " " +
51      lector.Value + ">\r\n";
52      break;
53      case XmlNodeType.EndElement: // EndElement de XML, lo muestra
54      ImprimirTab( profundidad ); // inserta tabuladores
55      txtSalida.Text += "</" + lector.Name + ">\r\n";
56      profundidad--; // reduce la profundidad
57      break;
58  } // fin de switch
59  } // fin de while
60  } // fin del método XmlReaderTextForm_Load
61
62  // inserta tabuladores
63  private void ImprimirTab( int numero )
64  {
65      for ( int i = 0; i < numero; i++ )
66          txtSalida.Text += "\t";
67  } // fin del método ImprimirTab
68  } // fin de la clase PruebaXmlReaderForm
69  } // fin del espacio de nombres PruebaXmlReader

```

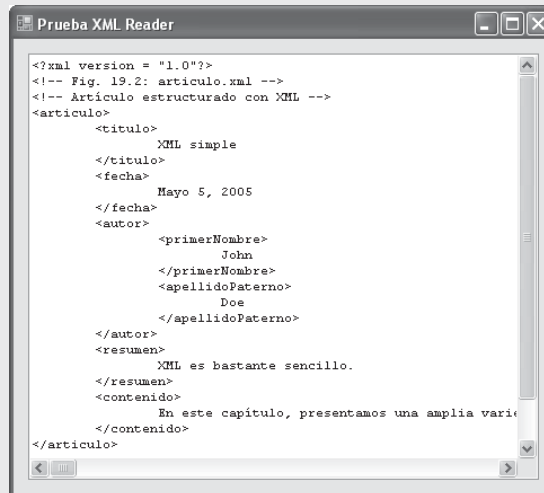


Figura 19.22 | Un objeto XmlReader iterando a través de un documento de XML. (Parte 2 de 2).

El método **Read** de XmlReader lee un nodo del árbol DOM. Al llamar a este método en la condición del ciclo while (línea 27), lector lee todos los nodos del documento. La instrucción switch (líneas 27-58) procesa cada nodo. Se da formato a la propiedad Name (líneas 32, 50 y 55), que contiene el nombre del nodo, o a la propiedad Value (líneas 40 y 44), que contiene los datos del nodo, y se concatena al objeto string asignado a la propiedad Text del control TextBox. La propiedad NodeType del objeto XmlReader especifica si el nodo es un elemento, comentario,

texto, declaración XML o elemento final. Observe que cada caso especifica un tipo de nodo mediante el uso de constantes de la enumeración **XmlNodeType**. Por ejemplo, `XmlNodeType.Element` (línea 31) indica la etiqueta inicial de un elemento.

Los resultados que se muestran en pantalla enfatizan la estructura del documento de XML. La variable profundidad (línea 23) mantiene el número de caracteres de tabulación para aplicar sangría a cada elemento. La profundidad se incrementa cada vez que el programa encuentra un tipo de nodo `Element`, y se decrementa cada vez que el programa encuentra un tipo de nodo `EndElement` o un elemento vacío. Utilizamos una técnica similar en el siguiente ejemplo, para enfatizar la estructura de árbol del documento de XML que se mostrará en pantalla.

Mostrar el gráfico de un árbol DOM en un control `TreeView`

Los objetos `XmlReader` no cuentan con herramientas para mostrar su contenido en forma gráfica. En este ejemplo, mostramos el contenido de un documento de XML mediante el uso de un control **`TreeView`**. Utilizamos la clase **`TreeNode`** para representar a cada nodo en el árbol. Las clases `TreeView` y `TreeNode` forman parte del espacio de nombres `System.Windows.Forms`. Los objetos `TreeNode` se agregan al control `TreeView` para enfatizar la estructura del documento de XML.

El programa en C# de la figura 19.23 demuestra cómo manipular un árbol DOM mediante la programación, para mostrar su gráfico en un control `TreeView`. La GUI para esta aplicación contiene un control `TreeView` llamado `xmlTreeView` (declarado en `XmlDom.Designer.cs`). La aplicación carga el archivo `carta.xml` (figura 19.24) en `XmlReader` (línea 27) y después muestra la estructura del árbol del documento en el control `TreeView`.

```

1  // Fig. 19.23: XmlDom.cs
2  // Demuestra la manipulación de un árbol DOM.
3  using System;
4  using System.Windows.Forms;
5  using System.Xml;
6
7  namespace XmlDom
8  {
9      public partial class XmlDomForm : Form
10     {
11         public XmlDomForm()
12         {
13             InitializeComponent();
14         } // fin del constructor
15
16         private TreeNode arbol; // referencia TreeNode
17
18         // inicializa las variables de instancia
19         private void XmlDomForm_Load( object sender, EventArgs e )
20         {
21             // crea el objeto de Xml ReaderSettings y
22             // establece la propiedad IgnoreWhitespace
23             XmlReaderSettings configuraciones = new XmlReaderSettings();
24             configuraciones.IgnoreWhitespace = true;
25
26             // crea el objeto XmlReader
27             XmlReader lector = XmlReader.Create( "carta.xml", configuraciones );
28
29             arbol = new TreeNode(); // crea instancia de TreeNode
30
31             arbol.Text = "carta.xml"; // asigna un nombre a TreeNode
32             treeXml.Nodes.Add( arbol ); // agrega TreeNode al control TreeView
33             CrearArbol( lector, arbol ); // crea la jerarquía de nodo y árbol

```

Figura 19.23 | Estructura DOM de un documento XML, visualizada en un control `TreeView`. (Parte I de 3).

```

34     } // fin del método XmlDomForm_Load
35
36     // construye TreeView con base en el árbol DOM
37     private void CrearArbol( XmlReader lector, TreeNode nodoArbol )
38     {
39         // nodoArbol que se va a agregar al árbol existente
40         TreeNode nuevoNode = new TreeNode();
41
42         while ( lector.Read() )
43         {
44             // crea árbol con base en el tipo de nodo
45             switch ( lector.NodeType )
46             {
47                 // si es nodo de texto, agrega su valor al árbol
48                 case XmlNodeType.Text:
49                     nuevoNode.Text = lector.Value;
50                     nodoArbol.Nodes.Add( nuevoNode );
51                     break;
52                 case XmlNodeType.EndElement: // si es EndElement, se mueve hacia arriba
                    del árbol
53                     nodoArbol = nodoArbol.Parent;
54                     break;
55
56                 // si es nuevo elemento, agrega el nombre y recorre el árbol
57                 case XmlNodeType.Element:
58
59                     // determina si el elemento tiene contenido
60                     if ( !lector.IsEmptyElement )
61                     {
62                         // asigna texto al nodo, agrega nuevoNode como hijo
63                         nuevoNode.Text = lector.Name;
64                         nodoArbol.Nodes.Add( nuevoNode );
65
66                         // establece nodoArbol al último hijo
67                         nodoArbol = nuevoNode;
68                     } // fin de if
69                     else // no recorre elementos vacíos
70                     {
71                         // asigna cadena de NodeType a nuevoNode
72                         // y lo agrega al árbol
73                         nuevoNode.Text = lector.NodeType.ToString();
74                         nodoArbol.Nodes.Add( nuevoNode );
75                     } // fin de else
76                     break;
77                 default: // para todos los demás tipos, muestra el tipo de nodo
78                     nuevoNode.Text = lector.NodeType.ToString();
79                     nodoArbol.Nodes.Add( nuevoNode );
80                     break;
81             } // fin de switch
82
83             nuevoNode = new TreeNode();
84         } // fin de while
85
86         // actualiza el control TreeView
87         treeXml.ExpandAll(); // expande los nodos del árbol en el control TreeView
88         treeXml.Refresh(); // obliga al control TreeView a actualizarse
89     } // fin del método CrearArbol
90 } // fin de la clase XmlDomForm
91 } // fin del espacio de nombres XmlDom

```

Figura 19.23 | Estructura DOM de un documento XML, visualizada en un control TreeView. (Parte 2 de 3).

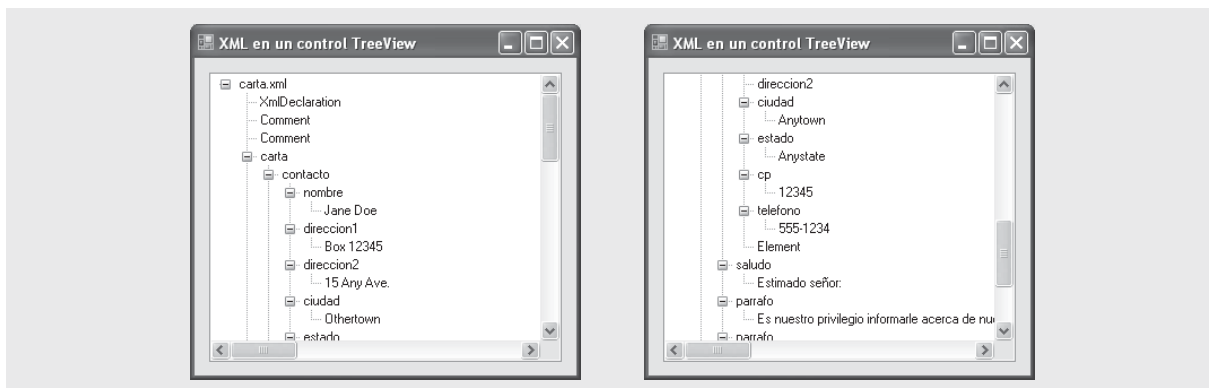


Figura 19.23 | Estructura DOM de un documento XML, visualizada en un control TreeView. (Parte 3 de 3).

[Nota: la versión de `carta.xml` en la figura 19.24 es casi idéntica a la de la figura 19.4, sólo que la figura 19.24 no hace referencia a una DTD, como en la línea 5 de la figura 19.4.]

En el manejador del evento Load de `XmlDomForm` (líneas 19-34), las líneas 23-24 crean un objeto `XmlReaderSettings` y establecen su propiedad **IgnoreWhitespace** a `true`, de manera que se ignoren los espacios en blanco insignificantes en el documento de XML. Después, la línea 27 invoca al método `static Create` de `XmlReader` para analizar y cargar el archivo `carta.xml`.

La línea 29 crea el objeto `TreeNode` llamado `arbol` (que se declara en la línea 16). Este objeto `TreeNode` se utiliza como una representación gráfica de un nodo de árbol DOM en el control `TreeView`. La línea 31 asigna el nombre del documento de XML (es decir, `carta.xml`) a la propiedad `Text` de `arbol`. La línea 32 llama al método `Add` para agregar el nuevo objeto `TreeNode` a la colección `Nodes` de `TreeView`. La línea 33 llama al método `BuildTree` para actualizar el control `TreeView`, de manera que muestre el árbol DOM de origen completo.

El método `CrearArbol` (líneas 37-89) recibe un objeto `XmlReader` para leer el documento de XML, y un objeto `TreeNode` que hace referencia a la ubicación actual en el árbol (es decir, el objeto `TreeNode` que se agregó más recientemente al control `TreeView`). La línea 40 declara la referencia `TreeNode` llamada `nuevoNodo`, que se utilizará para agregar nuevos nodos al control `TreeView`. Las líneas 42-84 iteran a través de cada nodo en el árbol DOM del documento de XML.

La instrucción `switch` en las líneas 45-81 agregan un nodo al control `TreeView`, con base en el nodo actual de `XmlReader`. Cuando se encuentra un nodo de texto, a la propiedad `Text` del nuevo objeto `TreeNode` (`nuevoNodo`) se le asigna el valor del nodo actual (línea 49). La línea 50 agrega este objeto `TreeNode` a la lista de nodos de `nodoArbol` (es decir, agrega el nodo al control `TreeView`).

La línea 52 coincide con un tipo de nodo `EndElement`. Esta instrucción case se mueve hacia arriba del árbol, hasta el padre del nodo, ya que se encontró el final de un elemento. La línea 53 accede a la propiedad **Parent** de `nodoArbol` para recuperar el padre actual del nodo.

La línea 57 coincide con los tipos de nodos `Element`. Cada tipo de nodo `Element` que no esté vacío (línea 60) incrementa la profundidad del árbol; por ende, asignamos el nombre (`Name`) del lector actual a la propiedad `Text` del `nuevoNodo` y agregamos ese `nuevoNodo` a la lista de nodos de `nodoArbol` (líneas 63-64). La línea 67 asigna la referencia de `nuevoNodo` a `nodoArbol`, para asegurar que `nodoArbol` haga referencia al último `NodoArbol` hijo en la lista de nodos. Si el nodo `Element` actual es un elemento vacío (línea 69), asignamos a la propiedad `Texto` de `nuevoNodo` la representación de cadena del `TipoNodo` (línea 73). A continuación, el `nuevoNodo` se agrega a la lista de nodos de `nodoArbol` (línea 74). La instrucción case predeterminada (líneas 77-80) asigna la representación de cadena del tipo de nodo a la propiedad `Text` de `nuevoNodo`, y después agrega el `nuevoNodo` a la lista de nodos `NodoArbol`.

Una vez que se procesa el árbol DOM completo, la lista de nodos `NodoArbol` se muestra en el control `TreeView` (líneas 87-88). El método **ExpandAll** de `TreeView` hace que se muestren todos los nodos del árbol. El método **Refresh** de `TreeView` actualiza la pantalla para mostrar los objetos `NodoArbol` recién agregados. Observe que, mientras la aplicación se ejecuta, si hace clic en los nodos (es decir, en los cuadros `+` o `-`) en el control `TreeView`, éstos se expanden o se contraen.

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.24: carta.xml -->
3  <!-- Carta de negocios con formato de XML -->
4
5  <carta>
6      <contacto tipo = "remitente">
7          <nombre>Jane Doe</nombre>
8          <direccion1>Box 12345</direccion1>
9          <direccion2>15 Any Ave.</direccion2>
10         <ciudad>Othertown</ciudad>
11         <estado>Otherstate</estado>
12         <cp>67890</cp>
13         <telefono>555-4321</telefono>
14         <bandera genero = "F" />
15     </contacto>
16
17     <contacto tipo = "destinatario">
18         <nombre>John Doe</nombre>
19         <direccion1>123 Main St.</direccion1>
20         <direccion2></direccion2>
21         <ciudad>Anytown</ciudad>
22         <estado>Anystate</estado>
23         <cp>12345</cp>
24         <telefono>555-1234</telefono>
25         <bandera genero = "M" />
26     </contacto>
27
28     <saludo>Estimado señor:</saludo>
29
30     <parrafo>Es nuestro privilegio informarle acerca de nuestra nueva base de datos
31         administrada con XML. Este nuevo sistema le permite reducir la carga en su
32         servidor de lista de inventario, al hacer que la máquina cliente realice
33         el trabajo de ordenar y filtrar los datos.
34     </parrafo>
35
36     <parrafo>Por favor visite nuestro sitio Web para ver la disponibilidad
37         y los precios.
38     </parrafo>
39
40     <cierre>Sinceramente,</cierre>
41     <firma>Ms. Jane Doe</firma>
42 </carta>

```

Figura 19.24 | Carta de negocios marcada como XML.

Localización de datos en documentos XML mediante XPath

Aunque la clase `XmlReader` incluye métodos para leer y modificar los valores de los nodos, no es el medio más eficiente para localizar datos en un árbol DOM. La Biblioteca de clases del .NET Framework cuenta con la clase **`XPathNavigator`** en el espacio de nombres **`System.Xml.XPath`** para iterar a través de listas de nodos que coincidan con los criterios de búsqueda, que se escriben como expresiones de XPath. Recuerde que XPath (Lenguaje de rutas de XML) proporciona una sintaxis para localizar nodos específicos en los documentos de XML, en forma efectiva y eficiente. XPath es un lenguaje basado en cadenas de expresiones utilizadas por XML, y muchas de sus tecnologías relacionadas (como XSLT, que vimos en la sección 19.7).

La figura 19.25 utiliza un objeto `XPathNavigator` para navegar a través de un documento de XML y utiliza un control `TreeView`, junto con objetos `TreeNode`, para mostrar la estructura del documento de XML. En este ejemplo, la lista de nodos `NodoArbol` se actualiza cada vez que el objeto `XPathNavigator` se posiciona en un nuevo nodo, en vez de mostrar todo el árbol DOM completo de una sola vez. Los nodos se agregan y se eliminan del control `TreeView` para reflejar la ubicación del objeto `XPathNavigator` en el árbol DOM. La figura 19.26 muestra

```

1  // Fig. 19.25: NavegadorRutas.cs
2  // Demuestra la clase XPathNavigator.
3  using System;
4  using System.Windows.Forms;
5  using System.Xml.XPath; // contiene a XPathNavigator
6
7  namespace NavegadorRutas
8  {
9      public partial class NavegadorRutasForm : Form
10     {
11         public NavegadorRutasForm()
12         {
13             InitializeComponent();
14         } // fin del constructor
15
16         private XPathNavigator xPath; // navegador para recorrer el documento
17
18         // referencia al documento para uso de XPathNavigator
19         private XPathDocument documento;
20
21         // referencia a la lista TreeNode para uso del control TreeView
22         private TreeNode arbol;
23
24         // inicializa las variables y el control TreeView
25         private void NavegadorRutasForm_Load( object sender, EventArgs e )
26         {
27             // carga el documento de XML
28             documento = new XPathDocument( "deportes.xml" );
29             xPath = documento.CreateNavigator(); // crea el navegador
30             arbol = new TreeNode(); // crea el nodo raíz para los objetos TreeNode
31
32             arbol.Text = xPath.NodeType.ToString(); // #raíz
33             rutaArbol.Nodes.Add( arbol ); // agrega árbol
34
35             // actualiza el control TreeView
36             rutaArbol.ExpandAll(); // expande el nodo del árbol en el control TreeView
37             rutaArbol.Refresh(); // fuerza la actualización del control TreeView
38             rutaArbol.SelectedNode = arbol; // resalta la raíz
39         } // fin del método NavegadorRutasForm_Load
40
41         // procesa el evento btnSeleccionar_Click
42         private void btnSeleccionar_Click( object sender, EventArgs e )
43         {
44             XPathNodeIterator iterador; // permite iterar por los nodos
45
46             try // obtiene el nodo especificado del control ComboBox
47             {
48                 // selecciona el nodo especificado
49                 iterador = xPath.Select( cboSeleccion.Text );
50                 MostrarIterador( iterador ); // imprime la selección
51             } // fin de try
52             // atrapa expresiones inválidas
53             catch ( System.Xml.XPath.XPathException argumentException )
54             {
55                 MessageBox.Show( argumentException.Message, "Error",
56                                 MessageBoxButtons.OK, MessageBoxIcon.Error );
57             } // fin de catch
58         } // fin del método btnSeleccionar_Click
59

```

Figura 19.25 | Un objeto XPathNavigator, navegando por los nodos seleccionados. (Parte I de 4).

```

60 // se desplaza al primer hijo al ocurrir el evento btnPrimerHijo_Click
61 private void btnPrimerHijo_Click( object sender, EventArgs e )
62 {
63     TreeNode nuevoNodoArbol;
64
65     // se desplaza al primer hijo
66     if ( XPath.MoveToFirstChild() )
67     {
68         nuevoNodoArbol = new TreeNode(); // crea nuevo nodo
69
70         // establece la propiedad Text del nodo
71         // con el nombre o el valor del navegador
72         DeterminarTipo( nuevoNodoArbol, XPath );
73
74         // agrega nodos a la lista de nodos TreeNode
75         arbol.Nodes.Add( nuevoNodoArbol );
76         arbol = nuevoNodoArbol; // asigna nuevoNodoArbol al arbol
77
78         // actualiza el control TreeView
79         rutaArbol.ExpandAll(); // expande el nodo en el control TreeView
80         rutaArbol.Refresh(); // obliga a que el control TreeView se actualice
81         rutaArbol.SelectedNode = arbol; // resalta la raíz
82     } // fin de if
83     else // el nodo no tiene hijos
84         MessageBox.Show( "El nodo actual no tiene hijos.",
85             "", MessageBoxButtons.OK, MessageBoxIcon.Information );
86 } // fin del método btnPrimerHijo_Click
87
88 // se desplaza al padre del nodo al ocurrir el evento btnPadre_Click
89 private void btnPadre_Click( object sender, EventArgs e )
90 {
91     // se desplaza al padre
92     if ( XPath.MoveToParent() )
93     {
94         arbol = arbol.Parent;
95
96         // obtiene el número de nodos hijos, sin incluir subárboles
97         int cuenta = arbol.GetNodeCount( false );
98
99         // elimina a todos los hijos
100         for ( int i = 0; i < cuenta; i++ )
101             arbol.Nodes.Remove( arbol.FirstNode ); // elimina nodo hijo
102
103         // actualiza el control TreeView
104         rutaArbol.ExpandAll(); // expande el nodo en el control TreeView
105         rutaArbol.Refresh(); // obliga a que el control TreeView se actualice
106         rutaArbol.SelectedNode = arbol; // resalta la raíz
107     } // fin de if
108     else // si el nodo no tiene padre (nodo raíz)
109         MessageBox.Show( "El nodo actual no tiene padre.", "",
110             MessageBoxButtons.OK, MessageBoxIcon.Information );
111 } // fin del método btnPadre_Click
112
113 // busca el siguiente hermano al ocurrir el evento btnSiguiente_Click
114 private void btnSiguiente_Click( object sender, EventArgs e )
115 {
116     // declara e inicializa dos objetos TreeNode
117     TreeNode nuevoNodoArbol = null;
118     TreeNode nuevoNodo = null;

```

Figura 19.25 | Un objeto XPathNavigator, navegando por los nodos seleccionados. (Parte 2 de 4).

```

119
120 // se desplaza al siguiente hermano
121 if ( XPath.MoveNext() )
122 {
123     nuevoNodoArbol = arbol.Parent; // obtiene el nodo padre
124
125     nuevoNodo = new TreeNode(); // crea nuevo nodo
126
127     // decide si va a mostrar el nodo actual
128     DeterminarTipo( nuevoNodo, XPath );
129     nuevoNodoArbol.Nodes.Add( nuevoNodo ); // lo agrega al nodo padre
130
131     arbol = nuevoNodo; // establece la posición actual de visualización
132
133     // actualiza el control TreeView
134     rutaArbol.ExpandAll(); // expande el nodo en el control TreeView
135     rutaArbol.Refresh(); // obliga a que el control TreeView se actualice
136     rutaArbol.SelectedNode = arbol; // resalta la raíz
137 } // fin de if
138 else // el nodo no tiene más hermanos
139     MessageBox.Show( "El nodo actual es el último hermano.", "",
140         MessageBoxButtons.OK, MessageBoxIcon.Information );
141 } // fin del método btnSiguiente_Click
142
143 // obtiene al hermano anterior al ocurrir el evento btnAnterior_Click
144 private void btnAnterior_Click( object sender, EventArgs e )
145 {
146     TreeNode nodoArbolPadre = null;
147
148     // se desplaza al hermano anterior
149     if ( XPath.MoveToPrevious() )
150     {
151         nodoArbolPadre = arbol.Parent; // obtiene el nodo padre
152         nodoArbolPadre.Nodes.Remove( arbol ); // elimina el nodo actual
153         arbol = nodoArbolPadre.LastNode; // se desplaza al nodo anterior
154
155         // actualiza el control TreeView
156         rutaArbol.ExpandAll(); // expande el nodo del árbol en el control TreeView
157         rutaArbol.Refresh(); // obliga a que el control TreeView se actualice
158         rutaArbol.SelectedNode = arbol; // resalta la raíz
159     } // fin de if
160     else // si el nodo actual no tiene hermanos anteriores
161         MessageBox.Show( "El nodo actual es el primer hermano.", "",
162             MessageBoxButtons.OK, MessageBoxIcon.Information );
163 } // fin del método btnAnterior_Click
164
165 // imprime los valores para XPathNodeIterator
166 private void MostrarIterador( XPathNodeIterator iterador )
167 {
168     txtSeleccion.Clear();
169
170     // imprime los valores del nodo seleccionado
171     while ( iterador.MoveNext() )
172         txtSeleccion.Text += iterador.Current.Value.Trim() + "\r\n";
173 } // fin del método MostrarIterador
174
175 // determina si TreeNode debe mostrar el nombre o valor del nodo actual
176 private void DeterminarTipo( TreeNode nodo, XPathNavigator XPath )
177 {

```

Figura 19.25 | Un objeto XPathNavigator, navegando por los nodos seleccionados. (Parte 3 de 4).

```

178     switch ( XPath.NodeType ) // determina el valor de NodeType
179     {
180         case XPathNodeType.Element: // si es Element, obtiene su nombre
181             // obtiene el nombre del nodo actual y elimina los espacios en blanco
182             nodo.Text = XPath.Name.Trim();
183             break;
184         default: // obtiene los valores del nodo
185             // obtiene el valor del nodo actual y elimina los espacios en blanco
186             nodo.Text = XPath.Value.Trim();
187             break;
188     } // fin de switch
189     } // fin del método DeterminarTipo
190 } // fin de la clase NavegadorRutasForm
191 } // fin del espacio de nombres NavegadorRutas

```

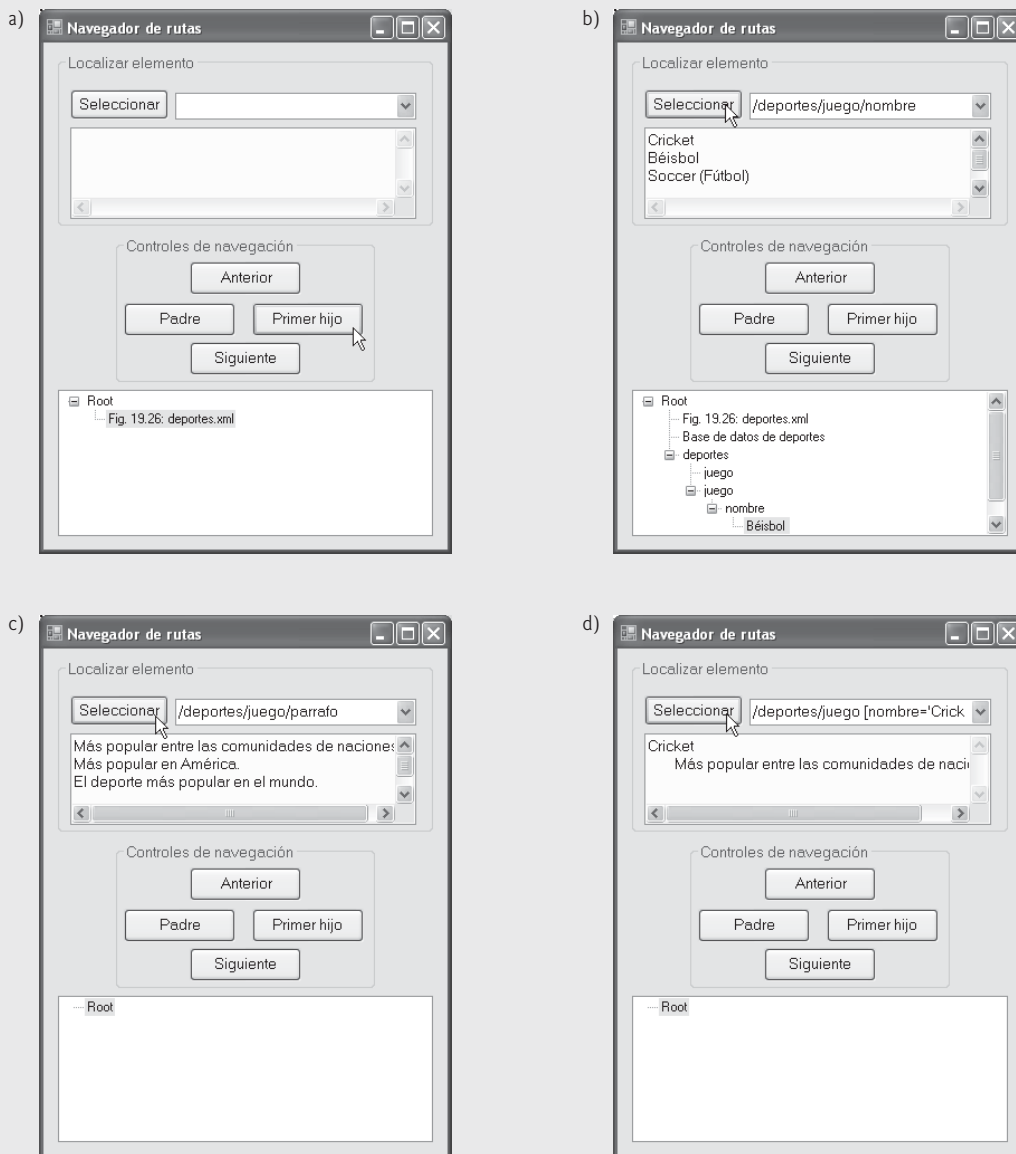


Figura 19.25 | Un objeto XPathNavigator, navegando por los nodos seleccionados. (Parte 4 de 4).

el documento de XML `deportes.xml` que utilizamos en este ejemplo. [Nota: Las versiones de `deportes.xml` que presentamos en las figuras 19.26 y 19.16 son casi idénticas. En el ejemplo actual no queremos aplicar una XSLT, por lo que omitimos la instrucción de procesamiento que se encuentra en la línea 2 de la figura 19.16.]

El programa de la figura 19.25 carga el documento de XML `deportes.xml` (figura 19.26) en un objeto **XPathDocument**; para ello le pasa el nombre de archivo del documento al constructor de **XPathDocument** (línea 28). El método **CreateNavigator** (línea 29) crea y devuelve una referencia **XPathNavigator** a la estructura del árbol de **XPathDocument**.

Los métodos de navegación de **XPathNavigator** son **MoveToFirstChild** (línea 66), **MoveToParent** (línea 92), **MoveToNext** (línea 121) y **MoveToPrevious** (línea 149). Cada método realiza la acción que su nombre implica. El método **MoveToFirstChild** se desplaza al primer hijo del nodo al que el objeto **XPathNavigator** hace referencia, **MoveToParent** se desplaza al nodo padre del nodo referenciado por el objeto **XPathNavigator**, **MoveToNext** se desplaza al siguiente hermano del nodo referenciado por el objeto **XPathNavigator**, y **MoveToPrevious** se desplaza al hermano anterior del nodo al que el objeto **XPathNavigator** hace referencia. Cada método devuelve un valor `bool`, indicando si el desplazamiento tuvo éxito. En este ejemplo, mostramos una advertencia en un cuadro **MessageBox** cada vez que falla una operación de desplazamiento. Lo que es más, se hace una llamada a cada método en el manejador de eventos del botón que coincide con su nombre (por ejemplo, al hacer clic en el botón **Primer hijo** en la figura 19.25(a) se activa `primerHijoButton_Click`, que hace una llamada a **MoveToFirstChild**).

Cada vez que avanzamos usando **XPathNavigator**, como con **MoveToFirstChild** y **MoveToNext**, se agregan nodos a la lista de nodos **TreeNode**. El método `private DeterminarTipo` (líneas 176-189) determina si se va a asignar la propiedad **Name** o la propiedad **Value** del nodo al objeto **TreeNode** (líneas 182 y 186). Cada vez que se hace una llamada a **MoveToParent**, todos los hijos del nodo padre se eliminan de la pantalla. De manera similar, una llamada a **MoveToPrevious** elimina el nodo hermano actual. Observe que los nodos se eliminan sólo del control **TreeView**, no de la representación de árbol del documento.

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.26: deportes.xml -->
3  <!-- Base de datos de deportes -->
4
5  <deportes>
6      <juego id = "783">
7          <nombre>Cricket</nombre>
8
9          <parrafo>
10             Más popular entre las comunidades de naciones.
11          </parrafo>
12      </juego>
13
14      <juego id = "239">
15          <nombre>Béisbol</nombre>
16
17          <parrafo>
18             Más popular en América.
19          </parrafo>
20      </juego>
21
22      <juego id = "418">
23          <nombre>Soccer (Fútbol)</nombre>
24
25          <parrafo>
26             El deporte más popular en el mundo.
27          </parrafo>
28      </juego>
29  </deportes>

```

Figura 19.26 | Documento de XML que describe varios deportes.

Expresión de XPath	Descripción
<code>/deportes</code>	Coincide con todos los nodos <code>deportes</code> que sean nodos hijos del nodo raíz del documento.
<code>/deportes/juego</code>	Coincide con todos los nodos <code>juego</code> que sean nodos hijos de <code>deportes</code> , que es un hijo de la raíz del documento.
<code>/deportes/juego/nombre</code>	Coincide con todos los nodos <code>nombre</code> que sean nodos hijos de <code>juego</code> . El nodo <code>juego</code> es un hijo de <code>deportes</code> , el cual es hijo de la raíz del documento.
<code>/deportes/juego/parrafo</code>	Coincide con todos los nodos <code>parrafo</code> que sean nodos hijos de <code>juego</code> . El nodo <code>juego</code> es un hijo de <code>deportes</code> , que es hijo de la raíz del documento.
<code>/deportes/juego [nombre='Cricket']</code>	Coincide con todos los nodos <code>juego</code> que contengan un elemento hijo, cuyo nombre sea <code>Cricket</code> . El nodo <code>juego</code> es un hijo de <code>deportes</code> , que es un hijo de la raíz del documento.

Figura 19.27 | Expresiones de XPath y sus descripciones.

El manejador de eventos `seleccionarButton_Click` (líneas 42-58) corresponde al botón **Seleccionar**. El método `Select` de `XPathNavigator` (línea 49) recibe los criterios de búsqueda en forma de una expresión ***XPath-Expression***, o de un objeto `string` que representa a la expresión XPath, y devuelve cualquier objeto que coincide con los criterios de búsqueda en forma de un objeto `XNodeIterator`. La figura 19.27 muestra un resumen de las expresiones de XPath que proporciona el control `ComboBox` de este programa. En la figura 19.25(b)-(d) mostramos el resultado de algunas de estas expresiones.

El método `MostrarIterador` (definido en las líneas 166-173) adjunta al control `seleccionTextBox` los valores de los nodos del objeto `XPathNodeIterator` dado. Observe que llamamos al método `string Trim` para eliminar el espacio en blanco innecesario. El método ***MoveNext*** (línea 171) avanza al siguiente nodo, al que la propiedad ***Current*** (línea 172) puede acceder.

19.9 (Opcional) Validación de esquemas con la clase `XmlReader`

En la sección 19.6 vimos que los esquemas proporcionan los medios para especificar la estructura de un documento de XML, y para validar los documentos de XML. Dicha validación ayuda a una aplicación a determinar si un documento específico que recibe está completo, ordenado en forma apropiada y si no le faltan datos. En la sección 19.6 utilizamos un validador de esquemas XSD en línea, para verificar que un documento XML se conforme a un esquema de XML. En esta sección le mostraremos cómo realizar el mismo tipo de validación mediante la programación, usando las clases que proporciona el .NET Framework.

Validación de un documento de XML mediante la programación

La clase `XmlReader` puede validar un documento de XML a medida que lo lee y analiza. En este ejemplo, demostraremos cómo activar dicha validación. El programa en la figura 19.28 valida un documento de XML seleccionado por el usuario; ya sea `libro.xml` (figura 19.11) o `falla.xml` (figura 19.29), y lo compara con el documento de XML Schema `libro.xsd` (figura 19.12).

La línea 17 crea la variable ***XmlSchemaSet*** llamada `esquemas`. Un objeto de la clase `XmlSchemaSet` almacena una colección de esquemas, para validarlos con un objeto `XmlReader`. La línea 23 asigna un nuevo objeto `XmlSchemaSet` a la variable `esquemas`, y la línea 26 llama al método ***Add*** de este objeto para agregar un esquema a la colección. El método `Add` recibe como argumentos un URI de espacio de nombres que identifica al esquema (<http://www.deitel.com/listalibros>), junto con el nombre y la ubicación del archivo de esquema (`libro.xsd` en el directorio actual).

Las líneas 29-32 crean y establecen las propiedades de un objeto `XmlReaderSettings`. La línea 30 establece la **propiedad *ValidationType*** del objeto `XmlReaderSettings` al valor ***ValidationType.Schema***, indicando que deseamos que el objeto `XmlReader` realice la validación con un esquema, a medida que vaya leyendo un

documento de XML. La línea 31 establece la *propiedad Schemas* del objeto `XmlReaderSettings` a esquemas. Esta propiedad establece el (los) esquema(s) utilizado(s) para validar el documento que lee el objeto `XmlReader`.

La línea 32 registra el método `ValidationError` con la *propiedad ValidationEventHandler* del objeto configuraciones. El método `ErrorValidacion` (líneas 52-56) se llama si el documento que se está leyendo es inválido, o si ocurre un error (por ejemplo, si el documento no puede encontrarse). Si no se registra un método con `ValidationEventHandler`, se lanza una excepción (`XmlException`) cuando el documento XML resulta ser inválido o no se encuentra.

Después de establecer las propiedades `ValidationType`, `Schemas` y `ValidationEventHandler` del objeto `XmlReaderSettings`, estamos listos para crear un objeto `XmlReader` de validación. Las líneas 35-36 crean un objeto `XmlReader` que lee el archivo que seleccionó el usuario del control `ComboBox cboArchivos`, y lo valida con el esquema `libro.xsd`.

La validación se realiza nodo por nodo, mediante llamadas al método `Read` del objeto `XmlReader` (línea 39). Debido a que establecimos la propiedad `ValidationType` de `XmlReaderSettings` a `ValidationType.Schema`, cada llamada a `Read` valida el siguiente nodo en el documento. El ciclo termina cuando se validan todos los nodos, o cuando falla la validación en uno de los nodos.

Detección de un documento de XML inválido

El programa en la figura 19.28 valida el documento de XML `libro.xml` (figura 19.12) con el esquema `libro.xsd` (figura 19.11) con éxito. No obstante, cuando el usuario selecciona el documento de XML de la figura 19.29,

```

1  // Fig. 19.28: PruebaValidacion.cs
2  // Validación de documentos XML con esquemas.
3  using System;
4  using System.Windows.Forms;
5  using System.Xml;
6  using System.Xml.Schema; // contiene la clase XmlSchemaSet
7
8  namespace PruebaValidacion
9  {
10     public partial class PruebaValidacionForm : Form
11     {
12         public PruebaValidacionForm()
13         {
14             InitializeComponent();
15         } // fin del constructor
16
17         private XmlSchemaSet esquemas; // esquemas con los que se va a validar
18         private bool valido = true; // resultado de la validación
19
20         // maneja el evento Click del botón Validar
21         private void btnValidar_Click( object sender, EventArgs e )
22         {
23             esquemas = new XmlSchemaSet(); // crea la clase XmlSchemaSet
24
25             // agrega el esquema a la colección
26             esquemas.Add( "http://www.deitel.com/listalibros", "libro.xsd" );
27
28             // establece las configuraciones de validación
29             XmlReaderSettings configuraciones = new XmlReaderSettings();
30             configuraciones.ValidationType = ValidationType.Schema;
31             configuraciones.Schemas = esquemas;
32             configuraciones.ValidationEventHandler += ErrorValidacion;
33
34             // crea el objeto XmlReader

```

Figura 19.28 | Ejemplo de validación de esquemas. (Parte I de 2).

```

35 XmlReader lector =
36     XmlReader.Create( cboArchivos.Text, configuraciones );
37
38 // analiza el archivo
39 while ( lector.Read() ); // cuerpo vacío
40
41 if ( valido ) // comprueba el resultado de la validación
42 {
43     lblConsola.Text = "El documento es válido";
44 } // fin de if
45
46 valido = true; // restablece la variable
47 lector.Close(); // cierra el flujo del lector
48 } // fin del método btnValidar_Click
49
50 // manejador de eventos para el error de validación
51 private void ErrorValidacion( object sender ,
52     ValidationEventArgs arguments )
53 {
54     lblConsola.Text = arguments.Message;
55     valido = false; // falló la validación
56 } // fin del método ErrorValidacion
57 } // fin de la clase PruebaValidacionForm
58 } // fin del espacio de nombres PruebaValidacion

```

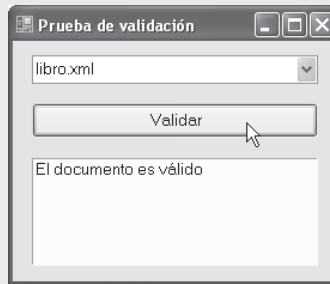


Figura 19.28 | Ejemplo de validación de esquemas. (Parte 2 de 2).

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 19.29: falla.xml -->
3  <!-- Archivo XML que no se conforma al esquema libro.xsd -->
4
5  <deitel:libros xmlns:deitel = "http://www.deitel.com/listalibros">
6      <libro>
7          <titulo>Visual Basic 2005 How to Program, 3/e</titulo>
8      </libro>
9
10     <libro>
11         <titulo>Visual C# 2005 How to Program</titulo>
12     </libro>
13
14     <libro>
15         <titulo>Java How to Program, 6/e</titulo>
16     </libro>
17
18     <libro>

```

Figura 19.29 | Archivo de XML que no se conforma al documento de XML Schema en la figura 19.12. (Parte 1 de 2).

```

19      <titulo>C++ How to Program, 5/e</titulo>
20      <titulo>Internet and World Wide Web How to Program, 3/e</titulo>
21    </libro>
22
23    <libro>
24      <titulo>XML How to Program</titulo>
25    </libro>
26  </deitel:libros>

```

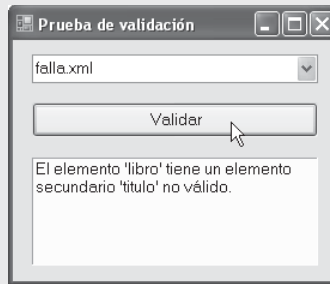


Figura 19.29 | Archivo de XML que no se conforma al documento de XML Schema en la figura 19.12. (Parte 2 de 2).

la validación falla; el elemento `libro` en las líneas 18-21 contiene más de un elemento `titulo`. Cuando el programa encuentra el nodo inválido, se hace una llamada al método `ErrorValidacion` (líneas 51-56 de la figura 19.28), con lo cual se muestra un mensaje explicando por qué falló la validación.

19.10 (Opcional) XSLT con la clase `XslCompiledTransform`

En la sección 19.7 vimos que XSLT proporciona elementos que definen las reglas para convertir un tipo de documento de XML en otro tipo de documento de XML. Mostramos cómo transformar documentos de XML en documentos de XHTML, y los visualizamos en Internet Explorer. El procesador XSLT incluido en Internet Explorer (es decir, MSXML) realizó las transformaciones.

Realización de una transformación XSL en C#, usando el .NET Framework

La figura 19.30 aplica la hoja de estilos `deportes.xsl` (figura 19.17) a `deportes.xml` (figura 19.26) mediante programación. El resultado de la transformación se escribe en un archivo de XHTML en el disco, y se muestra en un cuadro de texto. La figura 19.30(c) muestra el documento de XHTML resultante (`deportes.html`), visto en Internet Explorer.

La línea 5 es una declaración `using` para el espacio de nombres **`System.Xml.Xsl`**, el cual contiene la clase **`XslCompiledTransform`** para aplicar hojas de estilos XSLT a documentos de XML. La línea 17 declara la referencia `XslCompiledTransform` llamada `transformador`. Un objeto de este tipo sirve como un procesador XSLT (como MSXML en los ejemplos anteriores) para transformar datos XML de un formato a otro.

```

1  // Fig. 19.30: PruebaTransformacion.cs
2  // Aplicación de una hoja de estilos XSLT a un documento de XML.
3  using System;
4  using System.Windows.Forms;
5  using System.Xml.Xsl; // contiene la clase XslCompiledTransform
6
7  namespace PruebaTransformacion
8  {
9      public partial class PruebaTransformacionForm : Form

```

Figura 19.30 | Hoja de estilos XSLT aplicada a un documento de XML. (Parte 1 de 2).

```

10 {
11     public PruebaTransformacionForm()
12     {
13         InitializeComponent();
14     } // fin del constructor
15
16     // aplica la transformación
17     private XslCompiledTransform transformador;
18
19     // inicializa las variables
20     private void PruebaTransformacionForm_Load( object sender, EventArgs e )
21     {
22         transformador = new XslCompiledTransform(); // crea el transformador
23
24         // carga y compila la hoja de estilos
25         transformador.Load( "deportes.xsl" );
26     } // fin del método PruebaTransformacionForm_Load
27
28     // transforma los datos XML al ocurrir el evento Click del botón Transformar XML
29     private void btnTransformar_Click( object sender, EventArgs e )
30     {
31         // realiza la transformación y almacena el resultado en un nuevo archivo
32         transformador.Transform( "deportes.xml", "deportes.html" );
33
34         // lee y muestra el texto del documento de XHTML en un control Textbox
35         txtConsola.Text =
36             System.IO.File.ReadAllText( "deportes.html" );
37     } // fin del método btnTransformar_Click
38 } // fin de la clase PruebaTransformacionForm
39 } // fin del espacio de nombres PruebaTransformacion

```

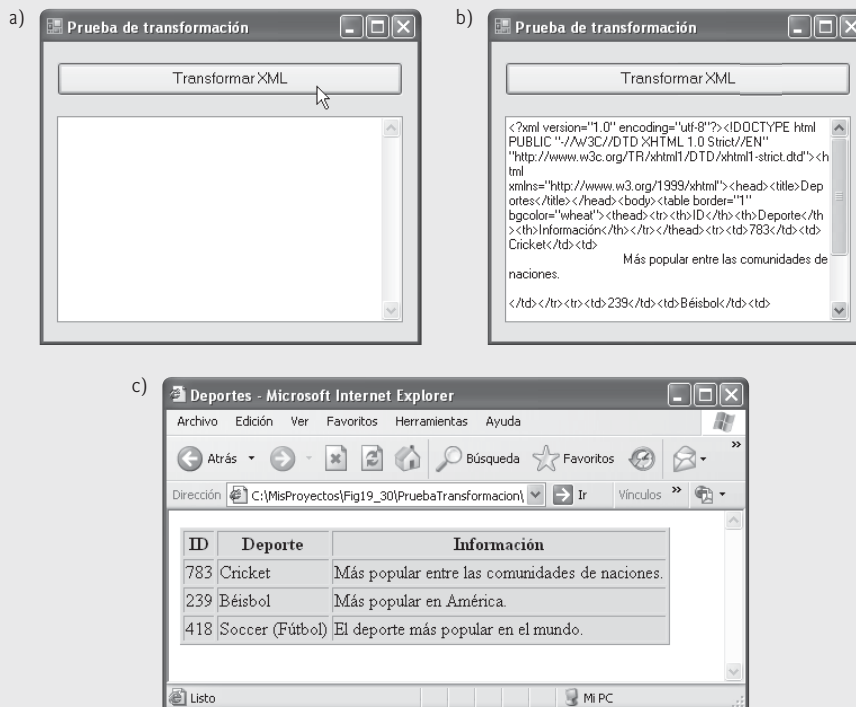


Figura 19.30 | Hoja de estilos XSLT aplicada a un documento de XML. (Parte 2 de 2).

En el manejador de eventos `PruebaTransformacionForm_Load` (líneas 20-26), la línea 22 crea un nuevo objeto `XslCompiledTransform`. Después la línea 25 llama al método **Load** del objeto `XslCompiledTransform`, que analiza y carga la hoja de estilos que utiliza esta aplicación. Este método recibe un argumento que especifica el nombre y la ubicación de la hoja de estilos, `deportes.xsl` (figura 19.17), que se encuentra en el directorio actual.

El manejador de eventos `transformarButton_Click` (líneas 29-37) llama al método **Transform** de la clase `XslCompiledTransform` para aplicar la hoja de estilos (`deportes.xsl`) a `deportes.xml` (línea 32). Este método recibe dos argumentos `string`; el primero especifica el archivo XML al que debe aplicarse la hoja de estilos, y el segundo especifica el archivo en el que debe almacenarse el resultado de la transformación en el disco. Por ende, la llamada al método `Transform` en la línea 32 transforma a `deportes.xml` en XHTML y escribe el resultado en disco como el archivo `deportes.html`. La figura 19.30(c) muestra el nuevo documento de XHTML representado en Internet Explorer. Observe que la salida es idéntica a la de la figura 19.16; no obstante, en el ejemplo actual, el XHTML se almacena en disco, en vez de que MSXML lo genere en forma dinámica.

Después de aplicar la transformación, el programa muestra el contenido del archivo `deportes.html` recién creado en el control `TextBox` llamado `txtConsole`, como se muestra en la figura 19.30(b). Las líneas 35-36 obtienen el texto del archivo, pasando su nombre al método `ReadAllText` de la clase `System.IO.File` que proporciona la FCL, para simplificar las tareas de procesamiento de archivos en el sistema local.

19.11 Conclusión

En este capítulo estudiamos el Lenguaje de marcado extensible y varias de sus tecnologías relacionadas. Empezamos hablando sobre cierta terminología básica de XML, y presentamos los conceptos del marcado, los vocabularios de XML y los analizadores de XML (de validación y no validadores). Después demostramos cómo describir y estructurar datos en XML, ilustrando estos puntos con ejemplos, marcando un artículo y una carta de negocios.

En este capítulo se describió también el concepto de un espacio de nombres de XML. Usted aprendió que un espacio de nombres tiene un nombre único, que proporciona los medios para que los autores de documentos se refieran sin ambigüedades a los elementos que tienen el mismo nombre (es decir, evitando el conflicto de nombres). Presentamos ejemplos de cómo definir dos espacios de nombres en el mismo documento, así como de establecer el espacio de nombres predeterminado para un documento.

También vimos cómo crear DTDs y esquemas para especificar y validar la estructura de un documento de XML. Mostramos cómo utilizar varias herramientas para confirmar si los documentos de XML son válidos (es decir, si se conforman a una DTD o a un esquema).

En este capítulo se demostró además cómo crear y utilizar documentos de XML, para especificar las reglas de conversión de documentos de XML en varios formatos. En específico, aprendió a dar formato y ordenar datos XML como XHTML, para mostrarlos en un explorador Web.

Las secciones finales del capítulo presentaron usos más avanzados de XML en aplicaciones en C#. Demostramos cómo recuperar y mostrar datos de un documento de XML, usando varias clases del .NET Framework. Ilustramos cómo un árbol del Modelo de objetos de documento (DOM) representa a cada uno de los elementos de un documento de XML como un nodo en el árbol. También se demostró cómo leer datos de un documento de XML mediante el uso de la clase `XmlReader`, cómo mostrar árboles DOM en forma gráfica y cómo localizar datos en documentos de XML con XPath. Por último, mostramos cómo utilizar las clases `XmlReader` y `XslCompiledTransform` del .NET Framework para realizar la validación de esquemas y las transformaciones XSL, respectivamente.

En el capítulo 20 empezaremos nuestra discusión sobre las bases de datos, que organizan datos de tal forma que éstos puedan seleccionarse y actualizarse con rapidez. Presentamos el Lenguaje de consultas estructurado (SQL) para escribir consultas simples de bases de datos (es decir, búsquedas) y ADO.NET para manipular la información en una base de datos a través de C#. En este capítulo aprenderá a crear documentos de XML con base en los datos contenidos en una base de datos.

19.12 Recursos Web

www.w3.org/XML

El W3C (Consorcio World Wide Web) facilita el desarrollo de protocolos comunes para asegurar la interoperabilidad en la Web. Su página XML incluye información acerca de los próximos eventos, publicaciones, software, grupos de discusión y los desarrollos más recientes en XML.

www.xml.org

xml.org es una referencia para XML, DTDs, esquemas y espacios de nombres.

www.w3.org/Style/XSL

Este sitio del W3C proporciona información acerca de XSL, incluyendo temas tales como el desarrollo con XSL, el aprendizaje de XSL, las herramientas compatibles con XSL; la especificación XSL, FAQs y antecedentes sobre XSL.

www.w3.org/TR

Éste es el sitio Web del W3C, acerca de los informes técnicos y las publicaciones. Contiene vínculos a anteproyectos funcionales, recomendaciones propuestas y otros recursos.

www.xmlbooks.com

Este sitio proporciona una lista de libros sobre XML recomendados por Charles Goldfarb, uno de los diseñadores originales de GML (Lenguaje de marcado general), del cual se derivan SGML, HTML y XML.

www.xml-zone.com

El sitio DevX XML Zone es un recurso completo de información sobre XML. Este sitio incluye preguntas frecuentes (FAQs), noticias, artículos y vínculos a otros sitios y grupos de noticias relacionados con XML.

wdvl.internet.com/Authoring/Languages/XML

El sitio Web Developer's Virtual Library XML incluye tutoriales, FAQs, las noticias más recientes y muchos vínculos a sitios de XML y descargas de software.

www.xml.com

Este sitio proporciona las noticias e información más reciente acerca de XML, listados de conferencias, vínculos a recursos Web de XML organizados por tema, herramientas y otros recursos.

msdn.microsoft.com/xml/default.aspx

El Centro de desarrollo de XML de MSDN contiene artículos sobre XML, sesiones de chat "Pregunte a los expertos", ejemplos, demos, grupos de noticias y demás información útil.

www.oasis-open.org/cover/xml.html

Este sitio incluye vínculos a FAQs, recursos en línea, iniciativas de la industria, demos, conferencias y tutoriales.

www-106.ibm.com/developerworks/xml

El sitio IBM developerWorks de XML es un gran recurso para desarrolladores, ya que ofrece noticias, herramientas, una biblioteca, casos de estudio e información acerca de eventos y estándares relacionados con XML.

www.devx.com/projectcool/door/7051

El sitio ProjectCool DevX incluye varios tutoriales que tratan acerca de temas de XML, desde introductorios hasta avanzados.

www.ucc.ie/xml

Este sitio proporciona una FAQ detallada sobre XML. Los desarrolladores pueden leer respuestas a ciertas preguntas populares, o enviar sus propias preguntas por medio del sitio.

www.w3.org/XML/Schema

Este sitio del W3C proporciona información acerca de XML Schema, incluyendo vínculos, herramientas, recursos y la especificación XML Schema.

tools.decisionsoft.com/schemaValidate.html

DecisionSoft proporciona un validador de XML Schema gratuito en línea.

www.sun.com/software/xml/developers/multischema

La página Web para descargar el Validador de XML Multi-Schema de Sun (MSV), el cual valida documentos de XML con varios tipos de esquemas.

en.wikipedia.org/wiki/EBNF

Este sitio proporciona información detallada acerca de la Forma de Backus-Naur extendida (EBNF).

www.garshol.priv.no/download/text/bnf.html

Este sitio introduce la Forma de Backus-Naur (BNF) y la EBNF, y habla sobre las diferencias entre estas dos notaciones.

www.w3schools.com/schema/default.asp

Este sitio proporciona un tutorial sobre XML Schema.

www.w3.org/2001/03/webdata/xsv

Ésta es una herramienta en línea para validar documentos de XML Schema.

www.w3.org/TR/xmlschema-2

Ésta es la parte 2 de la especificación XML Schema del W3C, la cual define los tipos de datos permitidos en un XML Schema.

20

Bases de datos, SQL y ADO.NET

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Acerca del modelo de bases de datos relacionales.
- Escribir consultas básicas de bases de datos en SQL.
- Agregar orígenes de datos a los proyectos.
- Utilizar las herramientas de “arrastrar y colocar” del IDE para mostrar tablas de bases de datos en aplicaciones.
- Utilizar las clases de los espacios de nombres `System.Data` y `System.Data.SqlClient` para manipular bases de datos.
- Utilizar el modelo de objetos sin conexión de ADO.NET para almacenar datos de una base de datos en la memoria local.
- Crear documentos de XML a partir de orígenes de datos.

*Es un error capital teorizar
antes de tener datos.*

—Arthur Conan Doyle

*Ahora ve,
escríbelo en una tablilla,
grábalo en un libro,
y que dure hasta el último día,
para testimonio hasta siempre.*

—Santa Biblia, Isaías 30:8

*Primero consiga los hechos,
y después podrá distorsionarlos
según le parezca.*

—Mark Twain

*Me gustan dos tipos de hombres:
domésticos y foráneos.*

—Mae West

- 20.1 Introducción
- 20.2 Bases de datos relacionales
- 20.3 Generalidades acerca de las bases de datos relacionales: la base de datos Libros
- 20.4 SQL
 - 20.4.1 Consulta SELECT básica
 - 20.4.2 Cláusula WHERE
 - 20.4.3 Cláusula ORDER BY
 - 20.4.4 Mezcla de datos de varias tablas: INNER JOIN
 - 20.4.5 Instrucción INSERT
 - 20.4.6 Instrucción UPDATE
 - 20.4.7 Instrucción DELETE
- 20.5 Modelo de objetos ADO.NET
- 20.6 Programación con ADO.NET: extraer información de una base de datos
 - 20.6.1 Mostrar una tabla de base de datos en un control DataGridView
 - 20.6.2 Cómo funciona el enlace de datos
- 20.7 Consulta de la base de datos Libros
- 20.8 Programación con ADO.NET: caso de estudio de libreta de direcciones
- 20.9 Uso de un objeto DataSet para leer y escribir XML
- 20.10 Conclusión
- 20.11 Recursos Web

20.1 Introducción

Una *base de datos* es una colección organizada de datos. Existen muchas estrategias para organizar datos, de manera que se facilite el acceso y la manipulación de los mismos. Un *sistema de administración de bases de datos* (*DBMS*) proporciona mecanismos para almacenar, organizar, recuperar y modificar datos para muchos usuarios. Los sistemas de administración de bases de datos permiten el acceso a los datos y su almacenamiento, de manera independiente a la representación interna de los datos.

Los sistemas de bases de datos más populares en la actualidad son las *bases de datos relacionales*. *SQL* es el lenguaje estándar internacional que se utiliza casi de manera universal con las bases de datos relacionales, para realizar *consultas* (es decir, para solicitar información que cumpla con ciertos criterios dados) y para manipular datos.

Algunos *sistemas de administración de bases de datos relacionales* (*RDBMS*) son: Microsoft SQL Server, Oracle, Sybase, IBM DB2 y PostgreSQL. En la Sección 20.11, Recursos Web, proporcionamos URLs para estos sistemas. MySQL (www.mysql.com) es un RDBMS de código fuente abierto que se está volviendo cada vez más popular, el cual puede descargarse y utilizarse libremente para fines no comerciales. Tal vez usted también esté familiarizado con Microsoft Access, un sistema de bases de datos relacionales que forma parte de Microsoft Office. En este capítulo, utilizamos *Microsoft SQL Server 2005 Express*: una versión gratuita de SQL Server 2005 que se instala al momento en que instalamos Visual C# 2005. Nos referiremos a SQL Server 2005 Express simplemente como SQL Server desde este punto en adelante.

Un lenguaje de programación se conecta e interactúa con una base de datos relacional a través de una *interfaz de base de datos*: un software que facilita la comunicación entre un sistema de administración de bases de datos y un programa. Los programas en C# se comunican con las bases de datos y manipulan sus datos a través de *ADO.NET*. La versión actual de ADO.NET es 2.0. La sección 20.5 presenta las generalidades acerca del modelo de objetos de ADO.NET, junto con los espacios de nombres y clases relevantes que le permitirán trabajar con las bases de datos en C#. Sin embargo, como aprenderá en las siguientes secciones, la mayor parte del trabajo requerido para comunicarse con una base de datos mediante el uso de ADO.NET 2.0 la realiza el mismo IDE. Usted trabajará principalmente con las herramientas y asistentes de programación visual del IDE, que simplifican

el proceso de conectarse a, y de manipular una base de datos. A lo largo de este capítulo, nos referiremos a ADO.NET 2.0 simplemente como ADO.NET.

En este capítulo se presentan los conceptos generales de las bases de datos relacionales y SQL, y después se explora ADO.NET y las herramientas del IDE para acceder a los orígenes de datos. Los ejemplos en las secciones 20.6-20.9 demuestran cómo crear aplicaciones que utilicen bases de datos para almacenar información. En los siguientes dos capítulos podrá ver otras aplicaciones prácticas de las bases de datos. El Capítulo 21, ASP.NET 2.0, formularios Web Forms y controles Web, presenta un caso de estudio de una librería basada en Web que recupera la información de usuarios y libros de una base de datos. El capítulo 22, Servicios Web, utiliza una base de datos para almacenar los datos de reservación de una aerolínea para un servicio Web (es decir, un componente de software al que se puede acceder en forma remota, a través de una red).

20.2 Bases de datos relacionales

Una *base de datos relacional* es una representación lógica de datos, que permite acceder a los mismos de manera independiente de su estructura física; además, organiza los datos en *tablas*. La figura 20.1 ilustra una tabla de ejemplo llamada *Empleados*, que podría usarse en un sistema de administración de personal. La tabla almacena los atributos de los empleados. Las tablas están compuestas de *filas* y *columnas*, en las que se almacenan los valores. Esta tabla consta de seis filas y cinco columnas. La columna *Numero* de cada fila en esta tabla es la *clave primaria*; una columna (o grupo de columnas), que requiere un valor único que no puede duplicarse en otras filas. Esto garantiza que pueda usarse un valor de clave primaria para identificar a una fila en forma única. Una clave primaria que está compuesta de dos o más columnas se conoce como *clave compuesta*. Algunos buenos ejemplos de columnas de clave primaria en otras aplicaciones son el número de identificación (ID) de un empleado en un sistema de nóminas, y el número de pieza en un sistema de inventarios; se garantiza que los valores en cada una de estas columnas serán únicos. Las filas en la figura 20.1 se muestran en orden por clave primaria. En este caso, las filas se listan en orden incremental (ascendente), pero también podrían listarse en orden decremental (descendente), o desordenadas. Como demostraremos en nuestro siguiente ejemplo, los programas pueden especificar criterios de ordenamiento al solicitar datos de una base de datos.

Cada columna representa un atributo de datos distinto. Por lo general, las filas son únicas (por clave primaria) dentro de una tabla, pero algunos valores de las columnas podrían duplicarse entre las filas. Por ejemplo, tres filas distintas en la columna *Departamento* de la tabla *Empleados* contienen el número 413, lo cual indica que estos empleados trabajan en el mismo departamento.

A menudo, los distintos usuarios de una base de datos se interesan en distintos datos y en distintas relaciones entre los datos. La mayoría de los usuarios sólo requiere subconjuntos de las filas y las columnas. Para obtener estos subconjuntos, los programas utilizan SQL para definir consultas que seleccionen subconjuntos de los datos de una tabla. Por ejemplo, un programa podría seleccionar datos de la tabla *Empleados* para crear el resultado de una consulta que muestre en dónde está ubicado cada departamento, en orden ascendente, por número de *Departamento* (figura 20.2). En la sección 20.4 hablaremos sobre las consultas de SQL. En la sección 20.7, aprenderá a utilizar el *Generador de consultas* del IDE para crear consultas de SQL.

Tabla Empleados					
	Numero	Nombre	Departamento	Salario	Ubicacion
Fila {	23603	Jones	413	1100	New Jersey
	24568	Kerwin	413	2000	New Jersey
	34589	Larson	642	1800	Los Angeles
	35761	Myers	611	1400	Orlando
	47132	Neumann	413	9000	New Jersey
	78321	Stephens	611	8500	Orlando
	Clave primaria		Columna		

Figura 20.1 | Datos de ejemplo de la tabla *Empleados*.

Departamento	Ubicacion
413	New Jersey
611	Orlando
642	Los Angeles

Figura 20.2 | Resultado de seleccionar distintos datos de Departamento y Ubicacion de la tabla Empleados.

20.3 Generalidades acerca de las bases de datos relacionales: la base de datos Libros

Ahora veremos las generalidades sobre las bases de datos relacionales, en el contexto de una base de datos simple llamada **Libros**. Esta base de datos almacena información acerca de algunas publicaciones recientes de Deitel. Primero veremos las tablas de la base de datos **Libros**. Después presentaremos los conceptos de las bases de datos, como la manera de usar SQL para recuperar información de la base de datos **Libros** y manipular esos datos. Le proporcionaremos el archivo de la base de datos (**Libros.mdf**) con los ejemplos para este capítulo (puede descargarlos de pearsoneducacion.net/deitel, o si prefiere la versión en inglés vaya a www.deitel.com/books/csharpforprogrammers2). Por lo general, los archivos de bases de datos de SQL Server terminan con la extensión de archivo **.mdf** (“archivo de datos maestro”). La sección 20.6 explica cómo utilizar este archivo en una aplicación.

Tabla Autores de la base de datos Libros

La base de datos consta de tres tablas: Autores, ISBNAutor y Titulos. La tabla Autores (descrita en la figura 20.3) cuenta con tres columnas que mantienen el número de ID único para cada autor, su primer nombre y su apellido paterno, respectivamente. La figura 20.4 contiene los datos de la tabla Autores. Listamos las filas en orden, con base en la clave primaria de la tabla: IDAutor. En la sección 20.4.3 aprenderá a ordenar los datos con base en otros criterios (por ejemplo, en orden alfabético por apellido paterno) mediante el uso de la cláusula **ORDER BY** de SQL.

Columna	Descripción
IDAutor	El número de ID del autor; en la base de datos Libros , esta columna de tipo entero se define como una columna de <i>identidad</i> , también conocida como columna <i>autoincremental</i> ; para cada fila que se inserta en la tabla, el valor de IDAutor se incrementa en 1 de manera automática, para asegurar que cada fila tenga un IDAutor único. Ésta es la clave primaria.
PrimerNombre	El primer nombre del autor (una cadena de caracteres).
ApellidoPaterno	El apellido paterno del autor (una cadena de caracteres).

Figura 20.3 | La tabla Autores de la base de datos Libros.

IDAutor	PrimerNombre	ApellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes

Figura 20.4 | Datos de la tabla Autores, de la base de datos Libros.

Tabla Títulos de la base de datos Libros

La tabla Títulos (descrita en la figura 20.5) tienen cuatro columnas que mantienen información acerca de cada libro en la base de datos, incluyendo el ISBN, título, número de edición y año de copyright. La figura 20.6 contiene los datos de la tabla Títulos.

Tabla ISBNAutor de la base de datos Libros

La tabla ISBNAutor (descrita en la figura 20.7) cuenta con dos columnas que mantienen números ISBN para cada libro y sus correspondientes números de ID del autor. Esta tabla asocia a los autores con sus libros. La columna IDAutor es una *clave externa*; una columna en esta tabla que coincide con la columna de clave primaria en otra tabla (es decir, IDAutor en la tabla Autores). La columna ISBN también es una clave externa; concuerda con la columna de clave primaria (es decir, ISBN) en la tabla Títulos. Tanto la columna IDAutor como ISBN, en conjunto, forman, en esta tabla, la clave primaria. Cada fila en esta tabla relaciona en forma única a un autor con el ISBN de un libro. La figura 20.8 contiene los datos de la tabla ISBNAutor, de la base de datos Libros.

Claves externas

Las claves externas pueden especificarse al momento de crear una tabla. Una clave externa ayuda a mantener la *Regla de integridad referencial*; todo valor de clave externa debe aparecer como valor de clave primaria de otra tabla. Esto permite a los DBMS determinar si el valor de IDAutor para una fila específica de la tabla ISBNAutor es válida. Las claves externas también permiten seleccionar datos relacionados en varias tablas de esas tablas; esto se conoce como *unir* los datos. (En la sección 20.4.4 aprenderá a unir datos mediante el uso del operador INNER JOIN de SQL.) Hay una *relación de uno a varios* entre una clave primaria y su correspondiente clave externa (por ejemplo, un autor puede escribir muchos libros). Esto significa que una clave externa puede aparecer muchas veces en su propia tabla, pero sólo una vez (como la clave primaria) en otra tabla. Por ejemplo, el ISBN 0131450913 puede aparecer en varias filas de ISBNAutor (ya que este libro tiene varios autores), pero sólo puede aparecer una vez en Títulos, en donde ISBN es la clave primaria.

Columna	Descripción
ISBN	El ISBN del libro (una cadena de caracteres); que es la clave primaria de la tabla. ISBN es una abreviación de “Número internacional estándar del libro”, un esquema de numeración que utilizan las editoriales en todo el mundo para dar a cada libro un número único de identificación.
Título	El título del libro (una cadena de caracteres).
NumeroEdicion	Número de edición del libro (un entero).
Copyright	Año de copyright del libro (una cadena de caracteres).

Figura 20.5 | La tabla Títulos de la base de datos Libros.

ISBN	Título	Numero-Edicion	Copy-right
0131426443	C How to Program	4	2004
0131450913	Internet & World Wide Web How to Program	3	2004
0131483986	Java How to Program	6	2005
0131525239	Visual C# 2005 How to Program	2	2006
0131828274	Operating Systems	3	2004
0131857576	C++ How to Program	5	2005
0131869000	Visual Basic 2005 How to Program	3	2006

Figura 20.6 | Datos de la tabla Títulos de la base de datos Libros.

Columna	Descripción
IDAutor	El número de ID del autor, una clave externa para la tabla Autores.
ISBN	El ISBN de un libro, una clave externa para la tabla Títulos.

Figura 20.7 | La tabla ISBNAutor de la base de datos Libros.

IDAutor	ISBN	IDAutor	ISBN
1	0131869000	2	0131450913
1	0131525239	2	0131426443
1	0131483986	2	0131857576
1	0131857576	2	0131483986
1	0131426443	2	0131525239
1	0131450913	2	0131869000
1	0131828274	3	0131450913
2	0131828274	4	0131828274

Figura 20.8 | Datos de la tabla ISBNAutor de Libros.

Diagrama de entidad relación para la base de datos Libros

La figura 20.9 es un *diagrama de entidad relación (ER)* para la base de datos Libros. Este diagrama muestra las tablas en la base de datos y las relaciones entre ellas. El primer compartimiento en cada cuadro contiene el nombre de la tabla. Los nombres en cursiva son claves primarias (por ejemplo, *IDAutor* en la tabla Autores). La clave primaria de una tabla identifica en forma única a cada fila de la misma. Cada fila debe tener un valor en la columna de clave primaria, y el valor de la clave debe ser único en la tabla. Esto se conoce como la *Regla de integridad de entidades*. Observe que los nombres IDAutor e ISBN en la tabla ISBNAutor están en cursiva; en conjunto, éstos forman una clave primaria compuesta para la tabla ISBNAutor.



Error común de programación 20.1

Si no se proporciona un valor para cada columna en una clave primaria, se rompe la Regla de integridad de las entidades y el DBMS reporta un error.



Error común de programación 20.2

Si se proporciona el mismo valor para la clave primaria en varias filas, se rompe la Regla de integridad de las entidades y el DBMS reporta un error.

Las líneas que conectan a las tablas en la figura 20.9 representan las relaciones entre las tablas. Considere la línea entre las tablas Autores e ISBNAutor. En el extremo de la línea correspondiente a Autores hay un 1, y en el extremo correspondiente a ISBNAutor hay un símbolo de infinito (∞). Esto indica una relación de uno a varios; para cada autor en la tabla Autores, puede haber un número arbitrario de números ISBN para los libros escritos por ese autor en la tabla ISBNAutor (es decir, un autor puede escribir cualquier número de libros). Observe que la línea de relación vincula a la columna IDAutor de la tabla Autores (en donde IDAutor es la clave primaria) con la columna IDAutor en la tabla ISBNAutor (en donde IDAutor es una clave externa); la línea entre las tablas vincula a la clave primaria con la correspondiente clave externa.



Error común de programación 20.3

Si se proporciona un valor de clave externa que no aparezca como valor de clave primaria en otra tabla, se rompe la Regla de integridad referencial y el DBMS reporta un error.

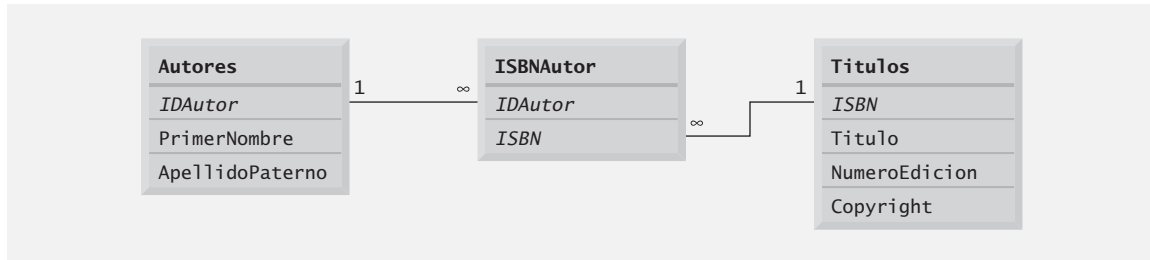


Figura 20.9 | Diagrama de entidad-relación para la base de datos Libros.

La línea entre las tablas *Titulos* e *ISBNAutor* ilustra una relación de uno a varios; varios autores pueden escribir un libro. Observe que la línea entre las tablas vincula a la clave primaria *ISBN* de la tabla *Titulos* con la correspondiente clave externa en la tabla *ISBNAutor*. Las relaciones en la figura 20.9 ilustran que el único propósito de la tabla *ISBNAutor* es proporcionar una relación de varios a varios entre las tablas *Autores* y *Titulos*; un autor puede escribir varios libros; y un libro puede tener varios autores.

20.4 SQL

Ahora veremos las generalidades acerca de SQL, en el contexto de la base de datos *Libros*. Más adelante en este capítulo creará aplicaciones en C# que ejecutan consultas de SQL y acceden a sus resultados mediante el uso de la tecnología ADO.NET. Aunque el IDE de Visual C# proporciona herramientas visuales que ocultan parte del SQL que se utiliza para manipular bases de datos, es sin duda importante comprender los fundamentos de SQL. Conocer los tipos de operaciones que usted puede realizar le ayudará a desarrollar aplicaciones más avanzadas, que hagan un uso extenso de las bases de datos.

La figura 20.10 lista algunas *palabras clave de SQL* comunes, que se utilizan para formar *instrucciones de SQL* completas; hablaremos sobre estas palabras clave en las siguientes subsecciones. Existen otras palabras clave de SQL, pero están más allá del alcance de este libro. Para obtener información adicional acerca de SQL, consulte los URLs listados en la Sección 20.11, Recursos Web.

20.4.1 Consulta SELECT básica

Vamos a considerar varias consultas de SQL que recuperan información de la base de datos *Libros*. Una *consulta* de SQL “selecciona” filas y columnas de una o más tablas en una base de datos. Las consultas realizan dichas selecciones con la palabra clave **SELECT**. La forma básica de una *consulta SELECT* es:

SELECT * FROM nombreTabla

Palabra clave de SQL	Descripción
SELECT	Recupera datos de una o más tablas.
FROM	Especifica las tablas involucradas en una consulta. Se requiere en todas las consultas.
WHERE	Especifica criterios opcionales para la selección, los cuales determinan las filas que se van a recuperar, eliminar o actualizar.
ORDER BY	Especifica criterios opcionales para ordenar filas (por ejemplo, en orden ascendente o descendente).
INNER JOIN	Especifica un operador opcional para mezclar filas de varias tablas.
INSERT	Inserta filas en una tabla especificada.
UPDATE	Actualiza filas en una tabla especificada.
DELETE	Elimina filas de una tabla especificada.

Figura 20.10 | Palabras clave comunes de SQL.

en donde el asterisco (*) indica que deben recuperarse todas las columnas de la tabla *nombreTabla*. Por ejemplo, para recuperar todos los datos en la tabla Autores, use:

```
SELECT * FROM Autores
```

Observe que no se garantiza que las filas de la tabla Autores se devuelvan en un orden específico. En la sección 20.4.3 aprenderá a especificar criterios para ordenar filas.

La mayoría de los programas no requieren todos los datos en una tabla. Para recuperar sólo columnas específicas de una tabla, sustituya el asterisco (*) con una lista separada por comas de los nombres de las columnas. Por ejemplo, para recuperar sólo las columnas IDAutor y ApellidoPaterno para todas las filas en la tabla Autores, use la consulta

```
SELECT IDAutor, ApellidoPaterno FROM Autores
```

Esta consulta devuelve sólo los datos que se listan en la figura 20.11.

20.4.2 Cláusula WHERE

Cuando los usuarios buscan en una base de datos filas que cumplan con ciertos *criterios de selección* (que formalmente se les conoce como *predicados*), sólo se seleccionan las filas que cumplen con esos criterios de selección. SQL utiliza la *cláusula WHERE* opcional en una consulta para especificar los criterios de selección de la consulta. La forma básica de una consulta con criterios de selección es:

```
SELECT nombreColumna1, nombreColumna2, ... FROM nombreTabla WHERE criterios
```

Por ejemplo, para seleccionar las columnas Título, NumeroEdicion y Copyright de la tabla Titulos, en donde la fecha de Copyright sea más reciente que 2004, use la consulta

```
SELECT Título, NumeroEdicion, Copyright  
FROM Titulos  
WHERE Copyright > '2004'
```

La figura 20.12 muestra el resultado de la consulta anterior.

IAutor	ApellidoPaterno
1	Deitel
2	Deitel
3	Goldberg
4	Choffnes

Figura 20.11 | Datos de IDAutor y ApellidoPaterno, de la tabla Autores.

Título	NumeroEdicion	Copyright
Java How to Program	6	2005
Visual C# 2005 How to Program	2	2006
C++ How to Program	5	2005
Visual Basic 2005 How to Program	3	2006

Figura 20.12 | Títulos con fechas de copyright después del 2004, de la tabla Titulos.

Los criterios de la cláusula **WHERE** pueden contener los operadores relacionales **<**, **>**, **<=**, **>=**, **=** (igualdad), **<>** (desigualdad) y **LIKE**, así como los operadores lógicos **AND**, **OR** y **NOT** (que veremos en la sección 20.4.6). El operador **LIKE** se utiliza para las *coincidencias de patrones* con los caracteres comodines *por ciento* (**%**) y *guión bajo* (**_**). Las coincidencias de patrones permiten a SQL buscar cadenas que coincidan con un patrón específico.

Un patrón que contiene un carácter de por ciento (**%**) busca cadenas que tengan cero o más caracteres en la posición del carácter de por ciento en el patrón. Por ejemplo, la siguiente consulta localiza las filas de todos los autores cuyos apellidos paternos empiecen con la letra D:

```
SELECT IDAutor, PrimerNombre, ApellidoPaterno
FROM Autores
WHERE ApellidoPaterno LIKE 'D%'
```

La consulta anterior selecciona las dos filas que se muestran en la figura 20.13, ya que dos de los cuatro autores en nuestra base de datos tienen un apellido paterno que empieza con la letra D (seguida de cero o más caracteres). El **%** en el patrón **LIKE** de la cláusula **WHERE** indica que puede aparecer cualquier número de caracteres después de la letra D en la columna **ApellidoPaterno**. Observe que la cadena del patrón va encerrada entre comillas sencillas.

Un guión bajo (**_**) en la cadena del patrón indica un solo carácter comodín en esa posición en el patrón. Por ejemplo, la siguiente consulta localiza las filas de todos los autores cuyo apellido paterno empieza con cualquier carácter (especificado por **_**), seguido de la letra h, seguido de cualquier número de caracteres adicionales (especificados por **%**):

```
SELECT IDAutor, PrimerNombre, ApellidoPaterno
FROM Autores
WHERE ApellidoPaterno LIKE '_h%'
```

La consulta anterior produce la fila que se muestra en la figura 20.14, ya que sólo hay un autor en nuestra base de datos que tenga un apellido paterno donde la segunda letra sea h.

20.4.3 Cláusula ORDER BY

Las filas en el resultado de una consulta pueden almacenarse en orden ascendente o descendente, mediante el uso de la *cláusula ORDER BY* opcional. La forma básica de una consulta con una cláusula **ORDER BY** es:

```
SELECT nombreColumna1, nombreColumna2, ... FROM nombreTabla ORDER BY columna ASC
SELECT nombreColumna1, nombreColumna2, ... FROM nombreTabla ORDER BY columna DESC
```

IAutor	PrimerNombre	ApellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel

Figura 20.13 | Autores de la tabla Autores cuyo apellido paterno empiece con D.

IAutor	PrimerNombre	ApellidoPaterno
4	David	Choffnes

Figura 20.14 | El único autor de la tabla Autores cuyo apellido paterno tiene una h como segunda letra.

en donde **ASC** especifica un orden ascendente (de menor a mayor), **DESC** especifica un orden descendente (de mayor a menor) y *columna* especifica la columna en la cual se basa el ordenamiento. Por ejemplo, para obtener la lista de autores en orden ascendente por apellido paterno (figura 20.15), utilizamos la siguiente consulta:

```
SELECT IDAutor, PrimerNombre, ApellidoPaterno
FROM Autores
ORDER BY ApellidoPaterno ASC
```

El orden predeterminado es ascendente, por lo que ASC es opcional en la consulta anterior.

Para obtener la misma lista de autores en orden descendente por apellido paterno (figura 20.16), utilizamos la siguiente consulta:

```
SELECT IDAutor, PrimerNombre, ApellidoPaterno
FROM Autores
ORDER BY ApellidoPaterno DESC
```

Pueden usarse varias columnas para ordenar datos con una cláusula ORDER BY de la forma:

```
ORDER BY columna1 tipoOrden, columna2 tipoOrden, ...
```

en donde *tipoOrden* puede ser ASC o DESC. Observe que *tipoOrden* no tiene que ser idéntico para cada columna. Por ejemplo, la consulta

```
SELECT Título, NumeroEdicion, Copyright
FROM Titulos
ORDER BY Copyright DESC, Título ASC
```

devuelve las filas de la tabla Titulos, ordenadas primero en orden descendente por fecha de copyright, después en orden ascendente por título (figura 20.17). Esto significa que las filas con valores mayores de Copyright se devuelven primero que las filas con valores menores de Copyright, y todas las filas que tenga los mismos valores de Copyright se ordenan en forma ascendente, por título.

Las cláusulas WHERE y ORDER BY pueden combinarse en una sola consulta. Por ejemplo, la consulta

```
SELECT ISBN, Título, NumeroEdicion, Copyright
FROM Titulos
WHERE Titulos LIKE '%How to Program'
ORDER BY Título ASC
```

devuelve el ISBN, Título, NumeroEdicion y Copyright de cada libro en la tabla Titulos que tenga un Título que termine con “How to Program”, y los ordena en forma ascendente por Título. Los resultados de la consulta se muestran en la figura 20.18.

20.4.4 Mezcla de datos de varias tablas: INNER JOIN

Por lo general, los diseñadores de las bases de datos *normalizan* las bases de datos; es decir, dividen los datos relacionados en tablas separadas, para asegurar que una base de datos no almacene datos redundantes. Por ejemplo,

IDAutor	PrimerNombre	ApellidoPaterno
4	David	Choffnes
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg

Figura 20.15 | Autores de la tabla Autores en orden ascendente por ApellidoPaterno.

IDAutor	PrimerNombre	ApellidoPaterno
3	Andrew	Goldberg
2	Harvey	Deitel
1	Paul	Deitel
4	David	Choffnes

Figura 20.16 | Autores de la tabla Autores en orden descendente por ApellidoPaterno.

Titulo	NumeroEdicion	Copyright
Visual Basic 2005 How to Program	3	2006
Visual C# 2005 How to Program	2	2006
C++ How to Program	5	2005
Java How to Program	6	2005
C How to Program	4	2004
Internet & World Wide Web How to Program	3	2004
Operating Systems	3	2004

Figura 20.17 | Datos de Titulos en orden descendente por Copyright, y en orden ascendente por Titulo.

ISBN	Titulo	NumeroEdicion	Copyright
0131426443	C How to Program	4	2004
0131857576	C++ How to Program	4	2005
0131450913	Internet & World Wide Web How to Program	4	2004
0131483986	Java How to Program	4	2005
0131869000	Visual Basic 2005 How to Program	4	2006
0131525239	Visual C# 2005 How to Program	4	2006

Figura 20.18 | Libros de la tabla Titulos, cuyos títulos terminan con How to Program en orden ascendente por Titulo.

la base de datos Libros tiene las tablas Autores y Titulos. Utilizamos una tabla ISBNAutor para almacenar “vínculos” entre los autores y los títulos. Si no separáramos esta información en tablas individuales, tendríamos que incluir la información del autor con cada entrada en la tabla Titulos. Esto significaría que la base de datos estaría almacenando información duplicada de los autores, para los autores que hayan escrito más de un libro.

A menudo es conveniente mezclar los datos provenientes de varias tablas en un solo resultado. A este proceso se le conoce como unir las tablas, y se especifica mediante un *operador INNER JOIN* en la consulta. Una operación

INNER JOIN mezcla filas de dos tablas, relacionando los valores en una columna que sea común para las tablas. La forma básica de una operación INNER JOIN es:

```
SELECT nombreColumna1, nombreColumna2, ...
FROM tabla1 INNER JOIN tabla2
ON tabla1.nombreColumna = tabla2.nombreColumna
```

La **cláusula ON** de INNER JOIN especifica las columnas que se van a comparar de cada tabla, para determinar cuáles filas se van a mezclar. Por ejemplo, la siguiente consulta produce una lista de autores, acompañada de los ISBNs para los libros escritos por cada autor:

```
SELECT PrimerNombre, ApellidoPaterno, ISBN
FROM Autores INNER JOIN ISBNAutor
ON Autores.IDAutor = ISBNAutor.IDAutor
ORDER BY ApellidoPaterno, PrimerNombre
```

La consulta combina las columnas PrimerNombre y ApellidoPaterno de la tabla Autores con la columna ISBN de la tabla ISBNAutor, y almacena los resultados en orden ascendente por ApellidoPaterno y PrimerNombre. Observe el uso de la sintaxis *nombreTabla.nombreColumna* en la cláusula ON. Esta sintaxis (conocida como *nombre calificado*) especifica las columnas de cada tabla que deben compararse para unir las tablas. La sintaxis “*nombreTabla*.” se requiere si las columnas tienen el mismo nombre en ambas tablas. Puede usarse la misma sintaxis en cualquier consulta, para diferenciar columnas que tengan el mismo nombre en distintas tablas.



Error común de programación 20.4

En una consulta de SQL, si no se califican los nombres para las columnas que tengan el mismo nombre en dos o más tablas, se produce un error.

Como siempre, la consulta puede contener una cláusula ORDER BY. La figura 20.19 ilustra los resultados de la consulta anterior, ordenados por PrimerNombre y ApellidoPaterno.

PrimerNombre	ApellidoPaterno	ISBN
David	Choffnes	0131828274
Harvey	Deitel	0131869000
Harvey	Deitel	0131525239
Harvey	Deitel	0131483986
Harvey	Deitel	0131857576
Harvey	Deitel	0131426443
Harvey	Deitel	0131450913
Harvey	Deitel	0131828274
Paul	Deitel	0131869000
Paul	Deitel	0131525239
Paul	Deitel	0131483986
Paul	Deitel	0131857576
Paul	Deitel	0131426443
Paul	Deitel	0131450913
Paul	Deitel	0131828274
Andrew	Goldberg	0131450913

Figura 20.19 | Autores y números ISBN para sus libros, en orden ascendente por ApellidoPaterno y PrimerNombre.

20.4.5 Instrucción INSERT

La *instrucción INSERT* inserta una fila en una tabla. La forma básica de esta instrucción es

```
INSERT INTO nombreTabla ( nombreColumna1, nombreColumna2, ..., nombreColumnaN )
VALUES ( valor1, valor2, ..., valorN )
```

en donde *nombreTabla* es la tabla en la que se va a insertar la fila. El *nombreTabla* va seguido de una lista separada por comas de los nombres de las columnas entre paréntesis (no se requiere esta lista si la operación INSERT especifica un valor para cada columna de la tabla, en el orden correcto). La lista de nombres de columnas va seguida de la palabra clave **VALUES** de SQL, y de una lista separada por comas de los valores entre paréntesis. Los valores que se especifican aquí deben coincidir con las columnas especificadas después del nombre de la tabla, tanto en orden como en tipo (por ejemplo, si se supone que *nombreColumna1* es la columna PrimerNombre, entonces *valor1* debe ser una cadena entre comillas sencillas, que represente el primer nombre). Siempre hay que listar en forma explícita las columnas al insertar filas; si cambia el orden de las columnas en la tabla y se utiliza sólo VALUES, se puede generar un error. La instrucción INSERT

```
INSERT INTO Autores ( PrimerNombre, ApellidoPaterno )
VALUES ( 'Sue', 'Smith' )
```

inserta una fila en la tabla Autores. Esta instrucción indica que los valores 'Sue' y 'Smith' se proporcionan para las columnas PrimerNombre y ApellidoPaterno, respectivamente.

No especificamos un IDAutor en este ejemplo, ya que es una columna de identidad en la tabla Autores (véase la figura 20.3). Para cada fila que se agrega a esta tabla, SQL Server asigna un valor único de IDAutor, que es el siguiente valor en una secuencia autoincremental (es decir, 1, 2, 3, etc.). En este caso, a Sue Smith se le asignaría el número 5 para IDAutor. La figura 20.20 muestra la tabla Autores después de la operación INSERT. No todos los DBMS soportan columnas de identidad o de autoincremento.



Error común de programación 20.5

Es un error especificar un valor para una columna de identidad.



Error común de programación 20.6

SQL utiliza el carácter de comilla sencilla (') para delimitar cadenas. Para especificar una cadena que contenga una comilla sencilla (por ejemplo, O'Malley) en una instrucción de SQL, debe haber dos comillas sencillas en la posición en donde aparece el carácter de comilla sencilla en la cadena (es decir, 'O' 'Ma1ley'). El primero de los dos caracteres de comilla sencilla actúa como carácter de escape para el segundo. Si no se escapan los caracteres de comilla sencilla en una cadena que forme parte de una instrucción de SQL, se produce un error de sintaxis.

20.4.6 Instrucción UPDATE

Una *instrucción UPDATE* modifica datos en una tabla. La forma básica de la instrucción UPDATE es

```
UPDATE nombreTabla
SET nombreColumna1 = valor1, nombreColumna2 = valor2, ..., nombreColumnaN = valorN
WHERE criterios
```

IDAutor	PrimerNombre	ApellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes
5	Sue	Smith

Figura 20.20 | La tabla Autores, después de una operación INSERT.

en donde *nombreTabla* es la tabla que se va a actualizar. El *nombreTabla* va seguido de la palabra clave **SET** y de una lista, separada por comas, de pares nombre de columna-valor, en el formato *nombreColumna* = *valor*. La cláusula **WHERE** opcional proporciona los criterios que determinan cuáles filas se van a actualizar. Aunque no se requiere, la cláusula **WHERE** se utiliza comúnmente, a menos que se requiera hacer una modificación en todas las filas. La instrucción **UPDATE**

```
UPDATE Autores
SET ApellidoPaterno = 'Jones'
WHERE ApellidoPaterno = 'Smith' AND PrimerNombre = 'Sue'
```

actualiza una fila en la tabla Autores. La palabra clave **AND** es un operador lógico que, al igual que el operador **&&** de C#, devuelve verdadero *sí, y sólo si* ambos operandos son verdaderos. Por ende, la instrucción anterior asigna el valor Jones a ApellidoPaterno para la fila en la que ApellidoPaterno es igual a Smith y PrimerNombre es igual a Sue. [Nota: si hay varias filas con el primer nombre “Sue” y el apellido paterno “Smith”, esta instrucción modifica todas esas filas para que tengan el apellido paterno “Jones”.] La figura 20.21 muestra la tabla Autores una vez que la operación **UPDATE** se haya realizado. SQL también proporciona otros operadores lógicos, como **OR** y **NOT**, que se comportan igual que sus contrapartes en C#.

20.4.7 Instrucción DELETE

Una *instrucción DELETE* elimina filas de una tabla. La forma básica de una instrucción **DELETE** es

```
DELETE FROM nombreTabla WHERE criterios
```

en donde *nombreTabla* es la tabla de la cual se van a eliminar datos. La cláusula **WHERE** opcional especifica los criterios utilizados para determinar cuáles filas se van a eliminar. La instrucción **DELETE**

```
DELETE FROM Autores
WHERE ApellidoPaterno = 'Jones' AND PrimerNombre = 'Sue'
```

elimina la fila para Sue Jones en la tabla Autores. Las instrucciones **DELETE** pueden eliminar varias filas, si éstas cumplen con los criterios en la cláusula **WHERE**. La figura 20.22 muestra la tabla Autores, una vez que se lleva a cabo la operación **DELETE**.

IDAutor	PrimerNombre	ApellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes
5	Sue	Jones

Figura 20.21 | La tabla Autores, después de una operación **UPDATE**.

IDAutor	PrimerNombre	ApellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes

Figura 20.22 | La tabla Autores, después de una operación **DELETE**.

Conclusión sobre SQL

Esto concluye nuestra introducción a SQL. Demostramos varias palabras clave de SQL que tienen uso común, formamos consultas de SQL para recuperar datos de las bases de datos y formamos otras instrucciones de SQL para manipular datos en una base de datos. Ahora vamos a presentar el modelo de objetos ADO.NET, el cual permite que las aplicaciones en C# interactúen con las bases de datos. Como veremos pronto, los objetos de ADO.NET manipulan bases de datos mediante instrucciones de SQL como las que presentamos aquí.

20.5 Modelo de objetos ADO.NET

El *modelo de objetos ADO.NET* proporciona una API para acceder a los sistemas de bases de datos mediante la programación. ADO.NET se creó para que el .NET Framework sustituyera a la tecnología ActiveX Data Objects™ (ADO) de Microsoft. Como veremos en la siguiente sección, el IDE cuenta con herramientas de programación visual que simplifican el proceso de utilizar una base de datos en sus proyectos. Aunque tal vez no necesite trabajar directamente con muchos objetos ADO.NET para desarrollar aplicaciones simples, es importante tener un conocimiento básico acerca de cómo funciona el modelo de objetos ADO.NET para comprender el acceso a los datos en C#.

Espacios de nombres *System.Data*, *System.Data.OleDb* y *System.Data.SqlClient*

El espacio de nombres **System.Data** es la raíz de la API de ADO.NET. Los otros espacios de nombres importantes de ADO.NET, **System.Data.OleDb** y **System.Data.SqlClient**, contienen clases que permiten a los programas conectarse con *orígenes de datos* y manipularlos; los orígenes de datos son ubicaciones que contienen datos, como una base de datos o un archivo XML. El espacio de nombres **System.Data.OleDb** contiene clases diseñadas para trabajar con cualquier origen de datos, mientras que **System.Data.SqlClient** cuenta con clases optimizadas para trabajar con bases de datos de Microsoft SQL Server. En este capítulo los ejemplos manipulan bases de datos de SQL Server 2005 Express, por lo que utilizamos las clases del espacio de nombres **System.Data.SqlClient**. SQL Server Express 2005 se incluye con Visual C# 2005 Express. También puede descargarlo de lab.msdn.microsoft.com/express/sql/default.aspx (en inglés), y de www.microsoft.com/spanish/msdn/vstudio/express/SQL/default.mspx (en español).

Un objeto de la clase **SqlConnection** (espacio de nombres **System.Data.SqlClient**) representa una conexión a un origen de datos; en específico, una base de datos de SQL Server. Un objeto **SqlConnection** lleva el registro de la ubicación del origen de datos y de cualquier configuración que especifique cómo se va a acceder a ese origen de datos. Una conexión puede estar *activa* (es decir, abierta y que permita presentar los datos a, y recuperarlos del origen de datos) o *cerrada*.

Un objeto de la clase **SqlCommand** (espacio de nombres **System.Data.SqlClient**) representa a un comando de SQL que un DBMS puede ejecutar en una base de datos. Un programa puede utilizar objetos **SqlCommand** para manipular un origen de datos a través de un objeto **SqlConnection**. El programa debe abrir la conexión con el origen de datos antes de poder ejecutar uno o más objetos **SqlCommand**, y debe cerrar la conexión una vez que ya no se requiera el acceso al origen de datos. Una conexión que permanece activa durante cierto tiempo para permitir múltiples operaciones de datos se conoce como *conexión persistente*.

La clase **DataTable** (espacio de nombres **System.Data**) representa a una tabla de datos. Un objeto **DataTable** contiene una colección de objetos **DataRow**, que representan a los datos de la tabla. Un objeto **DataTable** también tiene una colección de objetos **DataColumn**, los cuales describen las columnas en una tabla. Tanto **DataRow** como **DataColumn** se encuentran en el espacio de nombres **System.Data**. Un objeto de la clase **System.Data.DataSet**, que consta de un conjunto de objetos **DataTable** y de las relaciones entre ellos, representa a una *caché* de datos: los datos que un programa almacena en forma temporal en la memoria local. La estructura de un objeto **DataSet** imita a la estructura de una base de datos relacional.

Modelo sin conexión de ADO.NET

Una ventaja de utilizar la clase **DataSet** es que trabaja *sin conexión*; el programa no necesita una conexión persistente al origen de datos para trabajar con los datos en un objeto **DataSet**. En vez de ello, se conecta al origen de datos para *poblar el objeto DataSet* (es decir, llenar los objetos **DataTable** de **DataSet** con datos), pero se desconecta justo después de recuperar los datos deseados. Después, accede a, y potencialmente manipula, los datos almacenados en el objeto **DataSet**. El programa opera con esta caché local de datos, en vez de hacerlo con

los datos originales en el origen de datos. Si el programa realiza modificaciones en los datos en el objeto `DataSet` que necesiten guardarse de manera permanente en el origen de datos, vuelve a conectarse al origen de datos para realizar una actualización, y después se desconecta lo más pronto posible. Por ende, el programa no requiere de una conexión activa y persistente con el origen de datos.

Un objeto de la clase **`SqlDataAdapter`** (espacio de nombres `System.Data.SqlClient`) se conecta a un origen de datos de SQL Server y ejecuta instrucciones de SQL, tanto para poblar un objeto `DataSet` como para actualizar el origen de datos con base en el contenido actual de un objeto `DataSet`. Un objeto `SqlDataAdapter` mantiene un objeto `SqlConnection` que abre y cierra según sea necesario, para realizar estas operaciones mediante el uso de objetos `SqlCommand`. Más adelante en este capítulo mostraremos cómo poblar objetos `DataSet` y actualizar orígenes de datos.

20.6 Programación con ADO.NET: extraer información de una base de datos

En esta sección mostraremos cómo conectarnos a una base de datos, realizar una consulta y mostrar el resultado. En esta sección podrá observar que hay muy poco código. El IDE cuenta con herramientas de programación visual y asistentes que simplifican el acceso a los datos en sus proyectos. Estas herramientas establecen las conexiones a las bases de datos y crean los objetos ADO.NET necesarios para ver y manipular los datos a través de controles de la GUI. El ejemplo en esta sección se conecta a la base de datos `Libros` de SQL Server, de la cual hemos hablado a lo largo de este capítulo. Encontrará el archivo `Libros.mdf` que contiene la base de datos junto con los ejemplos de este capítulo en www.deitel.com/books/csharpforprogrammers2.

20.6.1 Mostrar una tabla de base de datos en un control `DataGridView`

Este ejemplo realiza una consulta simple en la base de datos `Libros`; recupera toda la tabla `Autores` y muestra los datos en un control **`DataGridView`** (un control del espacio de nombres `System.Windows.Forms` que puede mostrar un origen de datos en una GUI; en la figura 20.32 podrá ver la salida, más adelante en esta sección). Primero le mostraremos cómo conectarse a la base de datos `Libros` e incluirla como un origen de datos en su proyecto. Una vez establecida la base de datos `Libros` como un origen de datos, podrá mostrar los datos de la tabla `Autores` en un control `DataGridView`, con sólo arrastrar y colocar elementos en la vista de **Diseño** del proyecto.

Paso 1: crear el proyecto

Cree una nueva Aplicación para Windows llamada `MostrarTabla`. Cambie el nombre del formulario (Form) a `MostrarTablaForm` y cambie el nombre del archivo de código fuente a `MostrarTabla.cs`. Después establezca la propiedad `Text` del formulario a `Mostrar Tabla`.

Paso 2: agregar un origen de datos al proyecto

Para interactuar con un origen de datos (por ejemplo, una base de datos), debe agregarlo al proyecto usando la **ventana Orígenes de datos**, que lista los datos a los que su proyecto puede acceder. Para abrir la ventana **Orígenes de datos** (figura 20.23), seleccione **Datos > Mostrar orígenes de datos** o haga clic en la ficha que se encuentra a la derecha del **Explorador de soluciones**. En la ventana **Orígenes de datos**, haga clic en **Agregar nuevo origen de datos...** para abrir el **Asistente para la configuración de orígenes de datos** (figura 20.24). Este asistente lo guiará a través del proceso para conectarse a una base de datos y seleccionar a qué partes de la base de datos desee acceder en su proyecto.

Paso 3: seleccionar el tipo de origen de datos que se va a agregar al proyecto

La primera pantalla del **Asistente para la configuración de orígenes de datos** (figura 20.24) le pide que seleccione el tipo de origen de datos que desea incluir en el proyecto. Seleccione **Base de datos** y haga clic en **Siguiente >**.

Paso 4: agregar una nueva conexión de base de datos

Ahora debe seleccionar la conexión que utilizará para conectarse a la base de datos (es decir, el *origen* actual de los datos). Haga clic en **Nueva conexión...** para abrir el cuadro de diálogo **Agregar conexión** (figura 20.25). Si el **Origen de datos** no es **Archivo de base de datos de Microsoft SQL Server (SqlClient)**, haga clic en **Cambiar...**, seleccione **Archivo de base de datos de Microsoft SQL Server** y haga clic en **Aceptar**. En el cuadro de diálogo **Agregar conexión**,

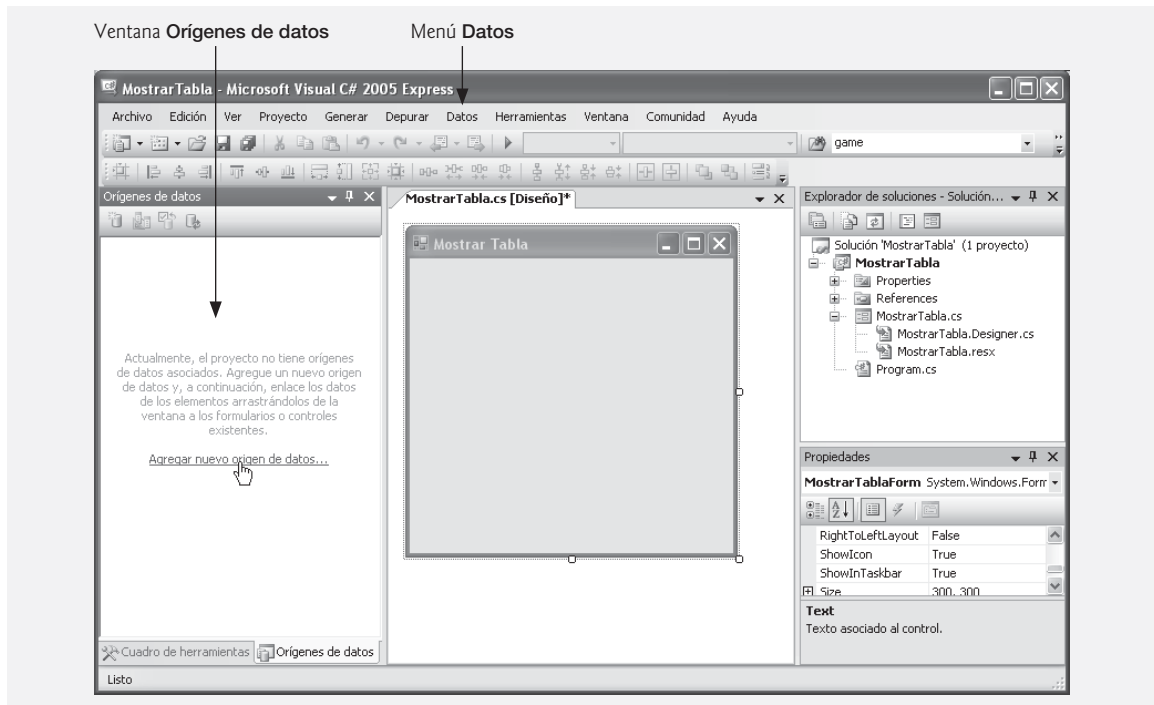


Figura 20.23 | Agregar un origen de datos a un proyecto.



Figura 20.24 | Seleccionar el tipo del origen de datos en el **Asistente para la configuración de orígenes de datos**.

haga clic en **Examinar...**, localice el archivo de base de datos **Libro.mdf** en su computadora, selecciónelo y haga clic en **Abrir**. Puede hacer clic en **Probar conexión** para verificar que el IDE pueda conectarse a la base de datos a través de SQL Server. Haga clic en **Aceptar** para crear la conexión.

Paso 5: seleccionar la conexión de datos *Libros.mdf*

Ahora que ha creado una conexión a la base de datos **Libros.mdf**, puede seleccionar y usar esta conexión para acceder a la base de datos. Haga clic en **Siguiente >** para establecer la conexión y después haga clic en **Sí** cuando se le pregunte si desea mover el archivo de base de datos a su proyecto (figura 20.26).

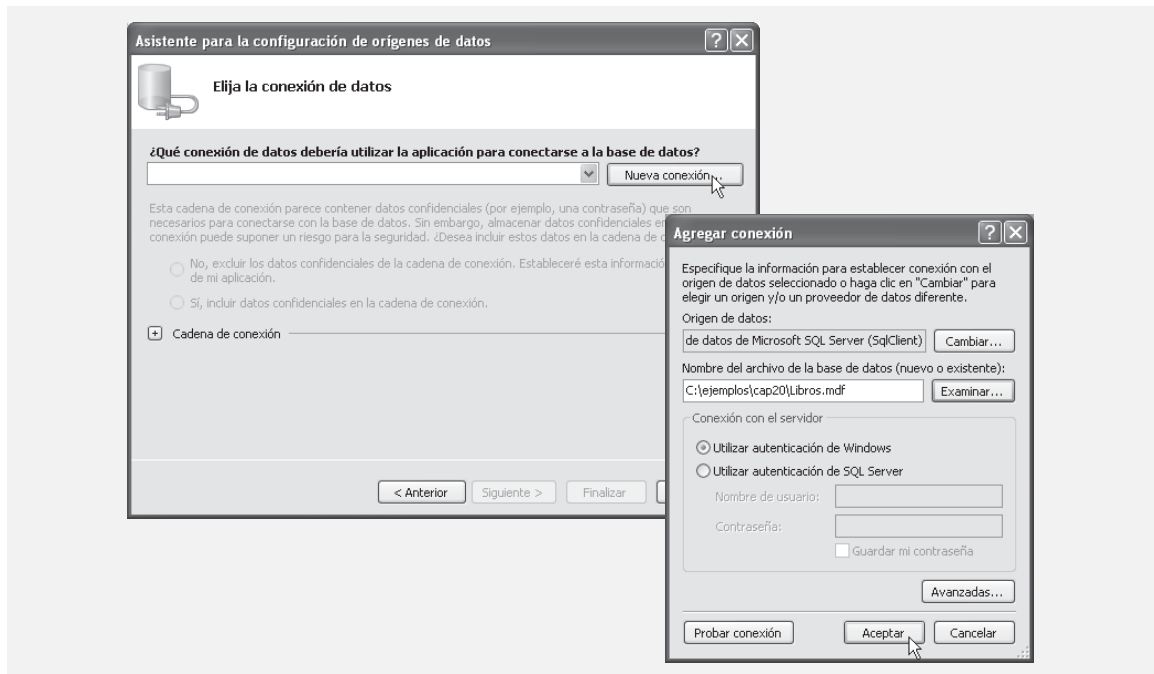


Figura 20.25 | Agregar una nueva conexión de datos.

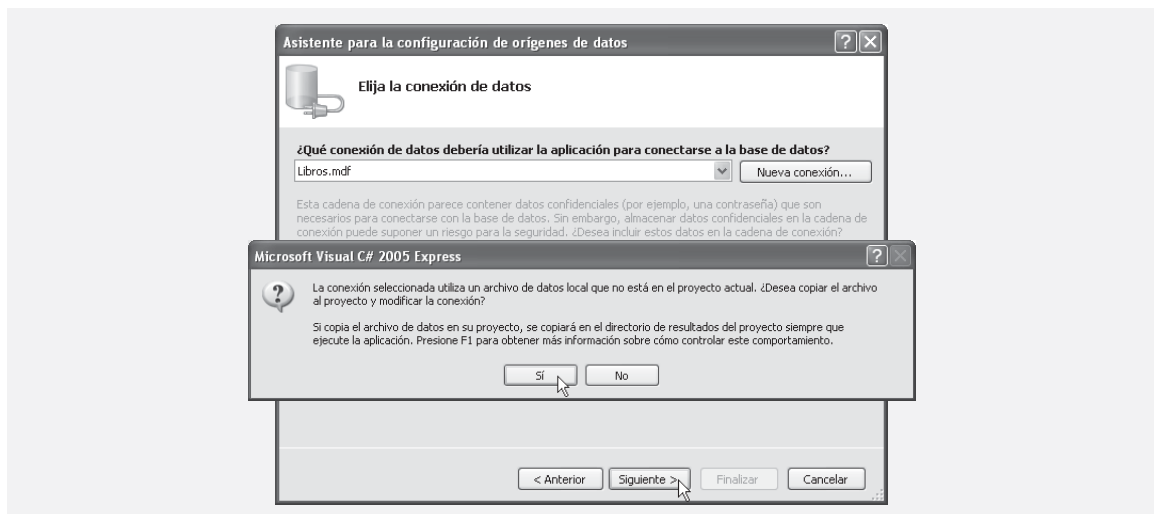


Figura 20.26 | Seleccionar la conexión de datos Libros.mdf.

Paso 6: guardar la cadena de conexión

La siguiente pantalla (figura 20.27) le pregunta si desea guardar la cadena de conexión en el archivo de configuración de la aplicación. Una *cadena de conexión* específica la ruta a un archivo de base de datos en el disco, así como algunas configuraciones adicionales que determinan cómo acceder a la base de datos. Si guarda la cadena de conexión en un archivo de configuración, será más sencillo cambiar las configuraciones de conexión más adelante. Deje las selecciones predeterminadas y haga clic en **Siguiente >** para proceder.

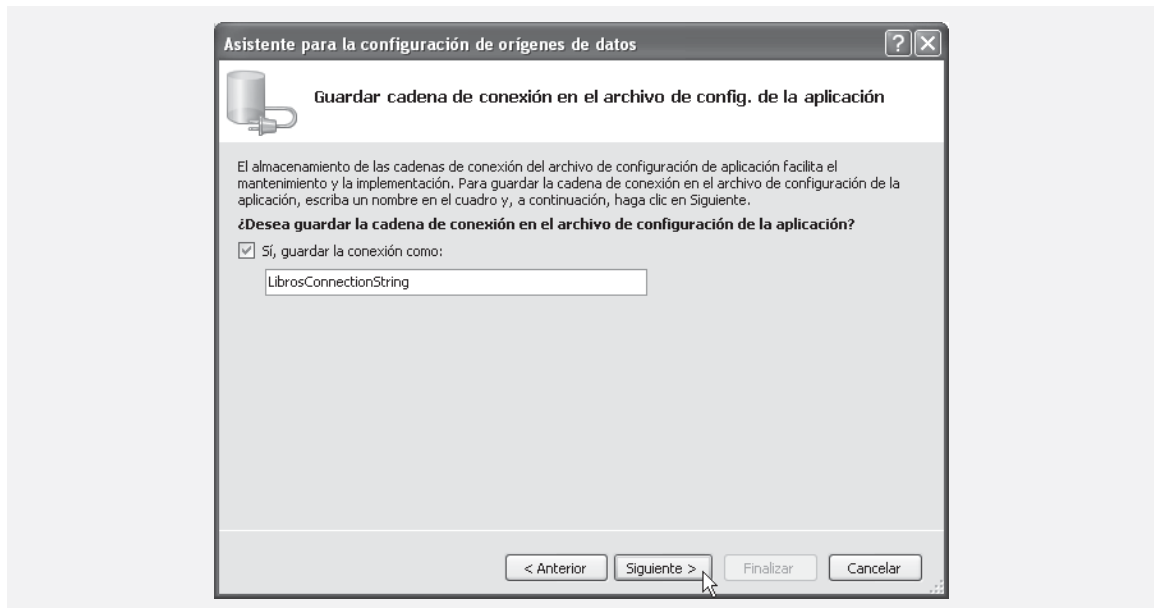


Figura 20.27 | Guardar la cadena de conexión en el archivo de configuración de la aplicación.

Paso 7: elegir los objetos de base de datos que va a incluir en su conjunto de datos (DataSet)

El IDE recupera información acerca de la base de datos que usted seleccionó y le pide que elija los objetos de base de datos (es decir, las partes de la base de datos) a los que desea que su proyecto pueda acceder (figura 20.28). Recuerde que, por lo general, los programas acceden al contenido de una base de datos a través de una caché de datos, la cual se almacena en un objeto DataSet. En respuesta a lo que usted elija en esta pantalla, el IDE generará una clase derivada de System.Data.DataSet que está diseñada específicamente para almacenar datos de la

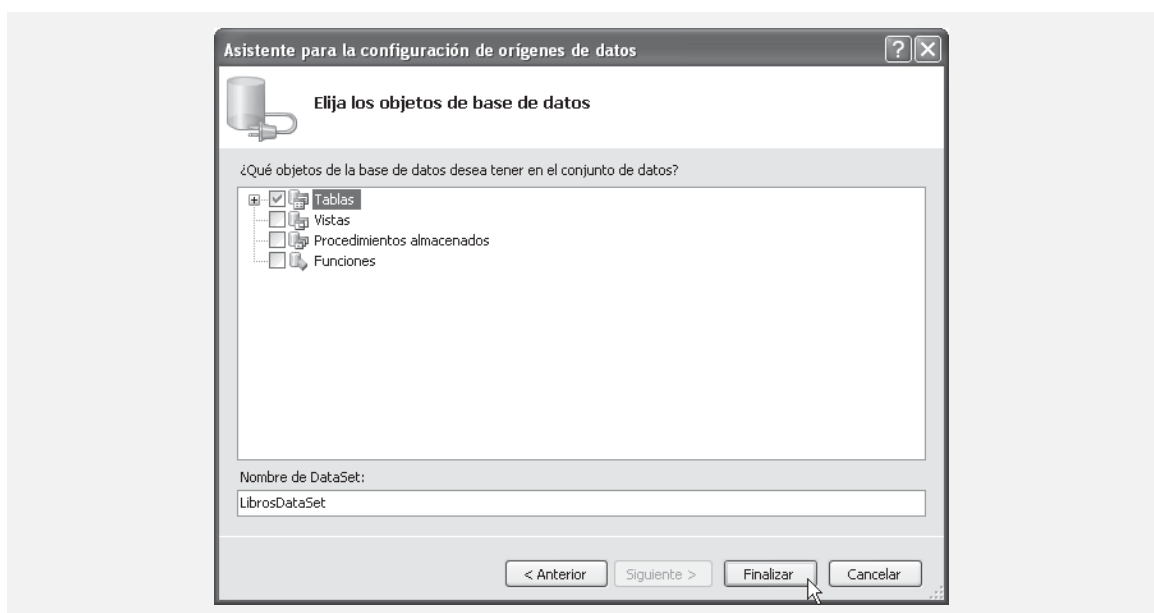


Figura 20.28 | Elección de los objetos de base de datos a incluir en el DataSet.

base de datos **Libros**. Haga clic en la casilla de verificación a la izquierda de **Tablas** para indicar que el objeto DataSet personalizado debe almacenar en la caché (es decir, almacenar localmente) los datos de todas las tablas en la base de datos **Libros**: **Autores**, **ISBNAutor** y **Titulos**. [Nota: También puede expandir el nodo **Tablas** para seleccionar tablas específicas. Los demás objetos de base de datos que se listan no contienen datos en nuestra base de datos **Libros**, y están más allá del alcance del libro.] De manera predeterminada, el IDE nombra al objeto DataSet **LibrosDataSet**, aunque es posible especificar un nombre distinto en esta pantalla. Por último, haga clic en **Finalizar** para completar el proceso de agregar un origen de datos al proyecto.

Paso 8: ver el origen de datos en la ventana Orígenes de datos

Observe que ahora aparece un nodo **LibrosDataSet** en la ventana **Orígenes de datos** (figura 20.29), con nodos hijos para cada tabla en la base de datos **Libros**; estos nodos representan los objetos DataTable del conjunto de datos **LibrosDataSet**. Expanda el nodo **Autores** y podrá ver las columnas de la tabla; la estructura del DataSet imita la de la verdadera base de datos **Libros**.

Paso 9: ver la base de datos en el Explorador de soluciones

Libros.mdf se lista ahora como un nodo en el **Explorador de soluciones** (figura 20.30), lo cual indica que ahora la base de datos forma parte de este proyecto. Además, el **Explorador de soluciones** lista ahora un nuevo nodo llamado **LibrosDataSet.xsd**. En el capítulo 19 aprendió que un archivo con la extensión **.xsd** es un documento de XML Schema, el cual especifica la estructura de un conjunto de documentos de XML. El IDE utiliza un documento de XML Schema para representar la estructura de un objeto DataSet, incluyendo las tablas que conforman el objeto DataSet y las relaciones entre ellos. Cuando agregó la base de datos **Libros** como un origen de datos, el IDE creó el archivo **LibrosDataSet.xsd** con base en la estructura de la base de datos **Libros**. Después, el IDE generó la clase **LibrosDataSet** del esquema (es decir, la estructura) descrito por el archivo **.xsd**.

Mostrar la tabla Autores

Ahora que agregó la base de datos **Libros** como un origen de datos, puede mostrar los datos de la tabla **Autores** de la base de datos en su programa. El IDE cuenta con herramientas de diseño que le permiten mostrar datos de un

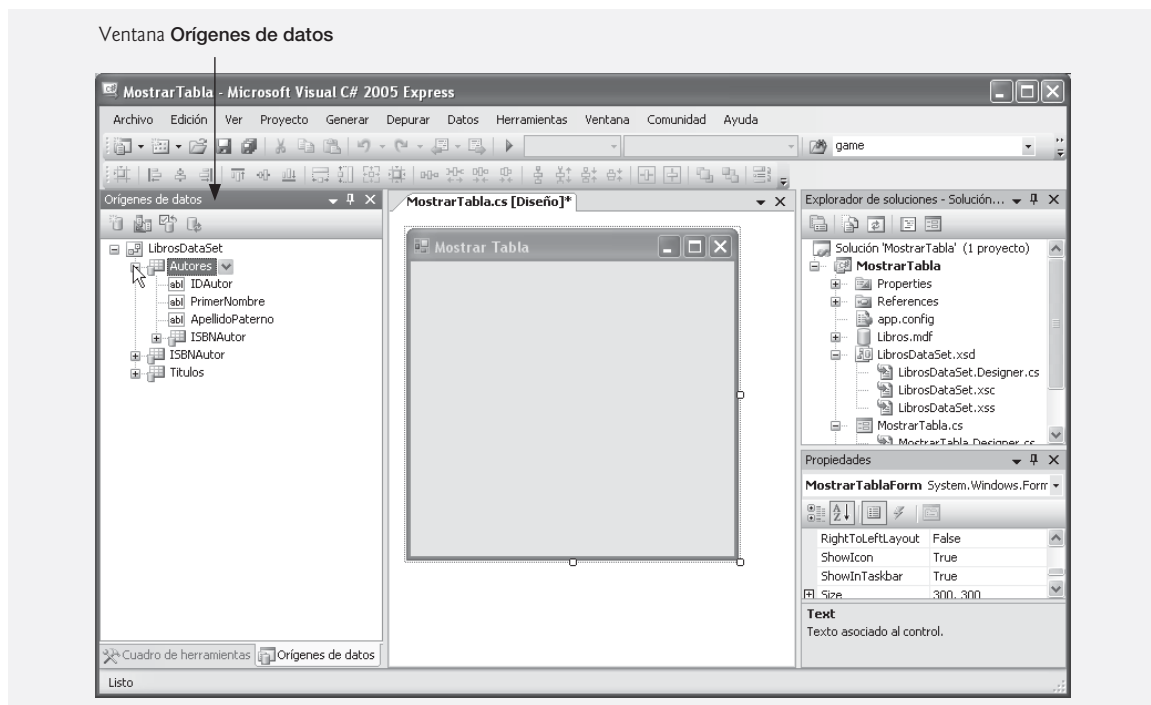


Figura 20.29 | Ver un origen de datos listado en la ventana **Orígenes de datos**.

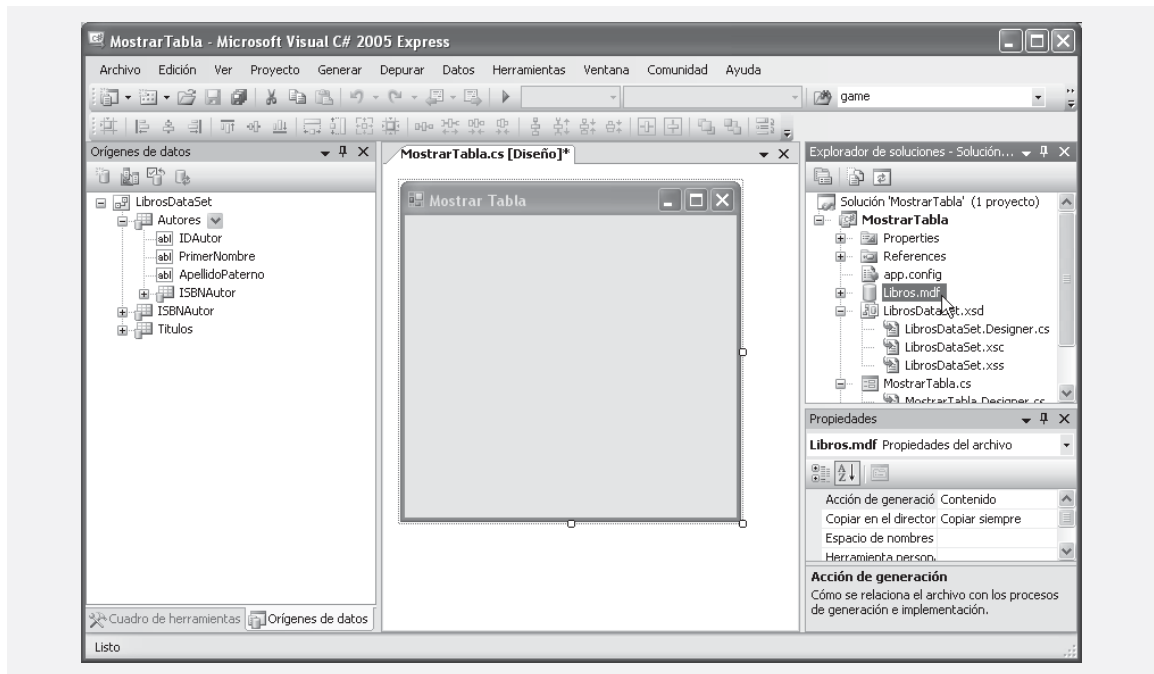


Figura 20.30 | Ver una base de datos listada en el **Explorador de soluciones**.

origen de datos en un formulario, sin necesidad de escribir código. Sólo tiene que arrastrar y colocar elementos de la ventana **Orígenes de datos** en un formulario, y el IDE genera los controles de la GUI y el código necesarios para mostrar el contenido del origen de datos seleccionado.

Para mostrar la tabla **Autores** de la base de datos **Libros**, arrastre el nodo **Autores** de la ventana **Orígenes de datos** hacia el formulario. La figura 20.31 presenta la vista de **Diseño** después de realizar esta acción y de cambiar el tamaño a los controles. El IDE genera dos controles de la GUI que aparecen en **MostrarTablaForm**: **autoresBindingNavigator** y **autoresDataGridView**. El IDE también genera varios componentes no visuales adicionales, que aparecen en la *bandeja de componentes*: la región gris debajo del formulario en vista de **Diseño**. Utilizamos los nombres predeterminados del IDE para estos componentes generados en forma automática (y otros a lo largo del capítulo) para mostrar exactamente qué es lo que crea el IDE. Aquí hablaremos brevemente sobre los controles **autoresBindingNavigator** y **autoresDataGridView**. La siguiente sección habla con detalle sobre todos los componentes generados en forma automática, y explica cómo el IDE utiliza esos componentes para conectar los controles de la GUI con la tabla **Autores** de la base de datos **Libros**.

Un control **DataGridView** muestra los datos organizados en filas y columnas, que corresponden a las filas y columnas del origen de datos subyacente. En este caso, el control **DataGridView** muestra los datos de la tabla **Autores**, por lo que sus columnas se llaman **IDAutor**, **PrimerNombre** y **ApellidoPaterno**. En la vista de **Diseño**, el control no muestra las filas de los datos actuales debajo de los encabezados de las columnas. Los datos se recuperan de la base de datos y se muestran en el control **DataGridView** sólo en tiempo de ejecución. Ejecute el programa. Cuando se carga el formulario, el control **DataGridView** contiene cuatro filas de datos; una para cada fila de la tabla **Autores** (figura 20.32).

La tira de botones debajo de la barra de título de la ventana es un control **BindingNavigator**, el cual permite a los usuarios explorar y manipular los datos mostrados por otro control de la GUI (en este caso, un control **DataGridView**) en el formulario. Los botones de un control **BindingNavigator** se asemejan a los controles en un reproductor de CD o DVD, y le permiten avanzar hacia la primera fila de datos, a la fila anterior, a la siguiente y a la última. El control también muestra el número de fila actual seleccionado en un cuadro de texto. Puede usar este cuadro de texto para introducir el número de una fila que desee seleccionar. En este ejemplo, el control **autoresBindingNavigator** le permite “navegar” por la tabla **Autores** que se muestra en el control **autoresDataGridView**. Al hacer clic en los botones o al introducir un valor en el cuadro de texto, el control **DataGridView**

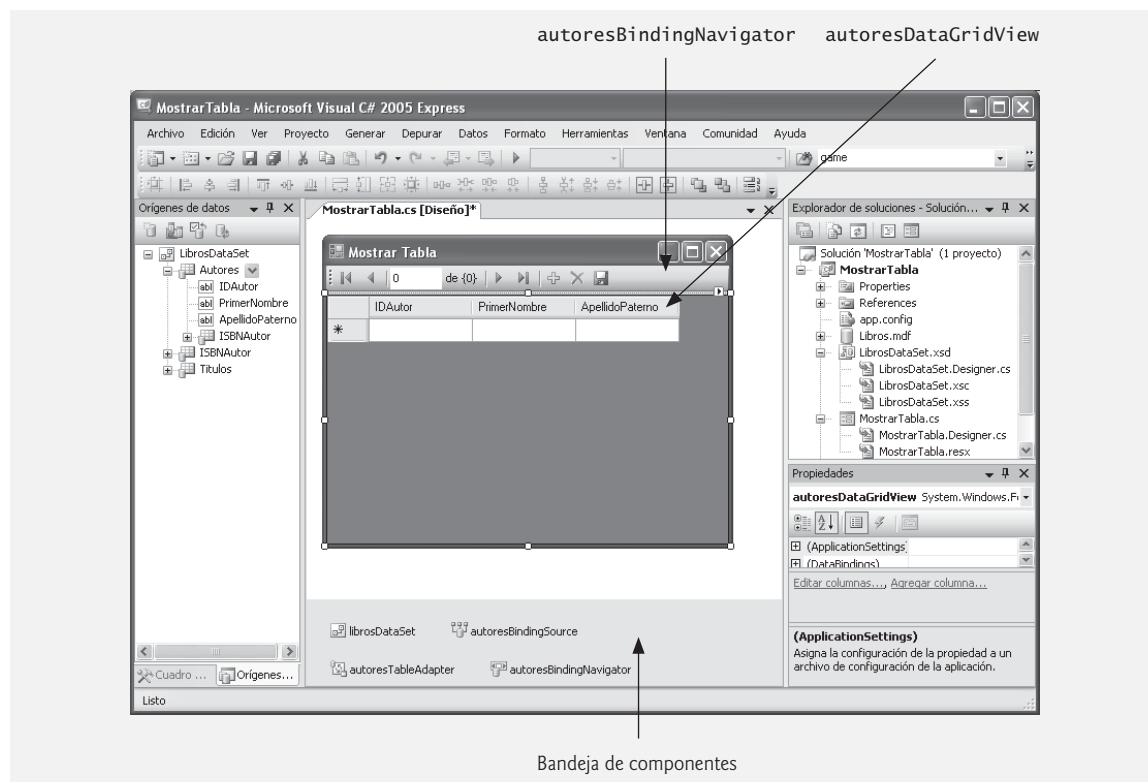


Figura 20.31 | Vista de **Diseño** después de arrastrar el nodo de origen de datos Autores al formulario.

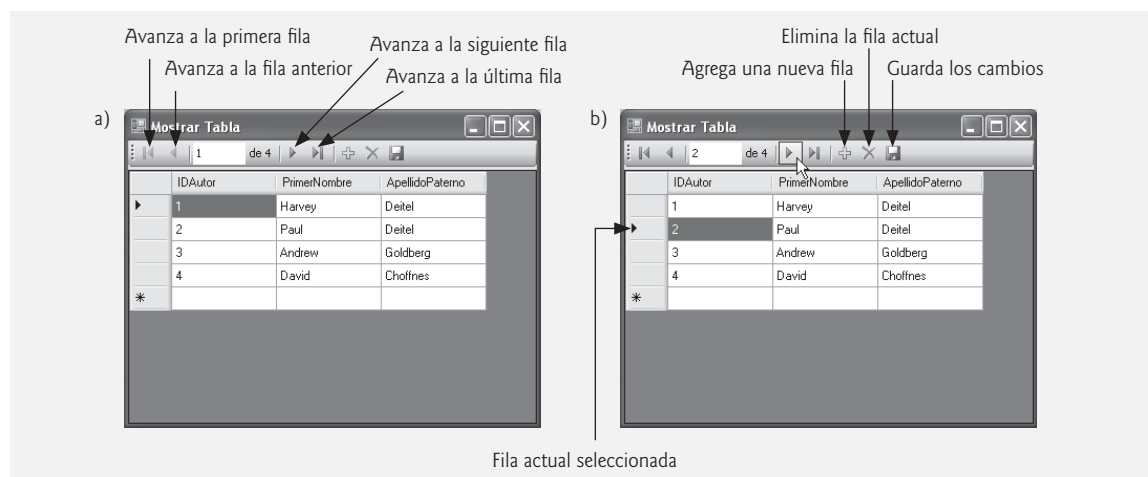


Figura 20.32 | Mostrar la tabla Autores en un control DataGridView.

selecciona la fila apropiada. Una flecha en la columna más a la izquierda del control DataGridView indica la fila actual seleccionada.

Un control BindingNavigator también tiene botones que le permiten agregar o eliminar una fila, y guardar los cambios de vuelta en el origen de datos subyacente (en este caso, la tabla Autores de la base de datos Libros). Al hacer clic en el botón con el icono de suma amarillo (+) se agrega una nueva fila al control DataGridView. Sin embargo, con sólo escribir valores en las columnas PrimerNombre y ApellidoPaterno

no se inserta una nueva fila en la tabla **Autores**. Para agregar la nueva fila a la base de datos en el disco, haga clic en el botón **Guardar** (💾). Al hacer clic en el botón con la X de color rojo (❌) elimina la fila actual seleccionada del control **DataGridView**. De nuevo, debe hacer clic en el botón **Guardar** para realizar el cambio en la base de datos. Pruebe estos botones. Ejecute el programa y agregue una nueva fila, después guarde los cambios y cierre el programa. Cuando reinicie el programa, deberá ver que la nueva fila se guardó en la base de datos y aparece en el control **DataGridView**. Ahora elimine la nueva fila y haga clic en el botón **Guardar**. Cierre y reinicie el programa para ver que la nueva fila ya no existe en la base de datos.

20.6.2 Cómo funciona el enlace de datos

La técnica a través de la que se conectan los controles de la GUI con los orígenes de datos se conoce como *enlace de datos*. El IDE permite que los controles tales como **DataGridView** se enlacen a un origen de datos, como un objeto **DataSet** que represente a una tabla en una base de datos. Cualquier modificación que usted realice a través de la aplicación al origen de datos subyacente, se reflejará de manera automática en la forma en que se presentan los datos en el control enlazado a datos (por ejemplo, el control **DataGridView**). De igual forma, si se modifican los datos en el control enlazado a datos y se guardan los cambios, se actualiza el origen de datos subyacente. En el ejemplo actual, el control **DataGridView** se enlaza al objeto **DataTable** del conjunto de datos **LibrosDataSet** que representa a la tabla **Autores** en la base de datos. Si se arrastra el nodo **Autores** de la ventana **Orígenes de datos** al formulario, el IDE crea este enlace de datos por usted, usando varios componentes generados en forma automática (es decir, objetos) en la bandeja de componentes. La figura 20.33 modela estos objetos y sus asociaciones, que las siguientes secciones analizan con detalle para explicar cómo funciona el enlace de datos.

LibrosDataSet

Como vimos en la sección 20.6.1, al agregar la base de datos **Libros** al proyecto, permitimos que el IDE generara el conjunto de datos **LibrosDataSet**. Recuerde que un objeto **DataSet** representa a una caché de datos, la cual

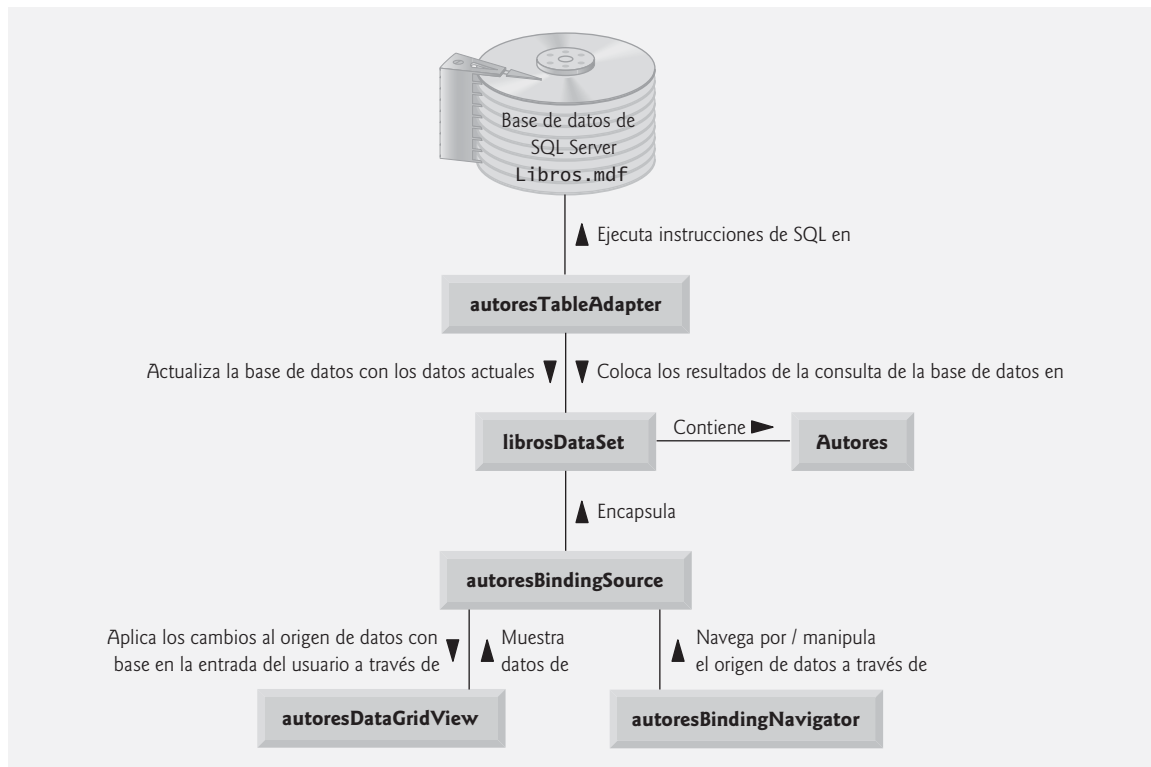


Figura 20.33 | Arquitectura de enlace de datos que se utiliza para mostrar la tabla **Autores** de la base de datos **Libros** en una GUI.

imita la estructura de una base de datos relacional. Puede explorar la estructura de conjunto de datos `LibrosDataSet` en la ventana **Orígenes de datos**. La estructura de un objeto `DataSet` puede determinarse en tiempo de ejecución o en tiempo de diseño. La estructura de un objeto ***DataSet sin tipo*** (es decir, las tablas que lo componen y las relaciones entre ellas) se determina en tiempo de ejecución, con base en el resultado de una consulta específica. Para acceder a las tablas y los valores de las columnas se utilizan índices en colecciones de objetos `DataTable` y `DataRow`, respectivamente. El tipo de cada pieza de datos en un objeto `DataSet` sin tipo se desconoce en tiempo de diseño. No obstante, el IDE crea el conjunto de datos `LibrosDataSet` en tiempo de diseño, como un objeto ***DataSet fuertemente tipificado***. El conjunto `LibrosDataSet` (una clase derivada de `DataSet`) contiene objetos de clases derivadas de `DataTable`, que representan las tablas en la base de datos `Libros`. `LibrosDataSet` proporciona propiedades que corresponden a los objetos cuyos nombres coinciden con los de las tablas subyacentes. Por ejemplo, `LibrosDataSet.Autores` representa una caché de los datos en la tabla `Autores`. Cada objeto `DataTable` contiene una colección de objetos `DataRow`. Cada objeto `DataRow` contiene miembros cuyos nombres y tipos corresponden a los de las columnas de la tabla de la base de datos subyacente. Por ende, `LibrosDataSet.Autores[ 0 ].IDAutor` se refiere al `IDAutor` de la primera fila de la tabla `Autores` en la base de datos `Libros`. Observe que se utilizan índices con base cero para acceder a los objetos `DataRow` en un objeto `DataTable`.

El objeto `LibrosDataSet` en la bandeja de componentes es un objeto de la clase `LibrosDataSet`. Cuando usted indica que desea mostrar el contenido de la tabla `Autores` en el formulario, el IDE genera un objeto `LibrosDataSet` para almacenar los datos que mostrará el formulario. Éstos son los datos con los que se enlazará el control `DataGridView`. Este control no muestra datos de la base de datos en forma directa, sino que muestra el contenido de un objeto `LibrosDataSet`. Como veremos en breve, el objeto `AutoresTableAdapter` llena el objeto `LibrosDataSet` con datos recuperados de la base de datos, mediante la ejecución de una consulta de SQL.

AutoresTableAdapter

El objeto `AutoresTableAdapter` es el componente que interactúa con la base de datos `Libros` en el disco (es decir, el archivo `Libros.mdf`). Cuando otros componentes necesitan recuperar datos de la base de datos, o escribir en ella, invocan a los métodos del objeto `AutoresTableAdapter`. El IDE genera la clase `AutoresTableAdapter` cuando usted arrastra una tabla de la base de datos `Libros` hacia el formulario. El objeto `autoresTableAdapter` en la bandeja de componentes es un objeto de esta clase. `AutoresTableAdapter` es responsable de llenar el objeto `LibrosDataSet` con los datos de `Autores` de la base de datos; almacena una copia de la tabla `Autores` en la memoria local. Como veremos pronto, esta copia en la caché puede modificarse durante la ejecución del programa. Por ende, `AutoresTableAdapter` es también responsable de actualizar la base de datos, cuando cambian los datos en `LibrosDataSet`.

La clase `AutoresTableAdapter` encapsula a un objeto `SqlDataAdapter`, el cual contiene objetos `SqlCommand` que especifican la manera en que el objeto `SqlDataAdapter` selecciona, inserta, actualiza y elimina datos en la base de datos. En la sección 20.5 vimos que un objeto `SqlCommand` debe tener un objeto `SqlConnection`, a través del cual el objeto `SqlCommand` puede comunicarse con una base de datos. En este ejemplo, el objeto `AutoresTableAdapter` establece la propiedad `Connection` de cada uno de los objetos `SqlCommand` de `SqlDataAdapter`, con base en la cadena de conexión que hace referencia a la base de datos `Libros`.

Para interactuar con la base de datos, el objeto `AutoresTableAdapter` invoca a los métodos de su objeto `SqlDataAdapter`, cada uno de los cuales ejecuta el objeto `SqlCommand` apropiado. Por ejemplo, para llenar la tabla `Autores` de `LibrosDataSet`, el método `Fill` de `AutoresTableAdapter` invoca al método `Fill` de su objeto `SqlDataAdapter`, el cual ejecuta un objeto `SqlCommand` que representa la siguiente consulta SELECT:

```
SELECT IDAutor, PrimerNombre, ApellidoPaterno FROM Autores
```

Esta consulta selecciona todas las filas y columnas de la tabla `Autores` y las coloca en `LibrosDataSet.Autores`. En breve, verá un ejemplo de la invocación del método `Fill` de `autoresTableAdapter`.

autoresBindingSource y autoresDataGridView

El objeto `autoresBindingSource` (un objeto de la clase ***BindingSource***) identifica a un origen de datos que un programa puede enlazar a un control, y sirve como intermediario entre un control enlazado a datos de la GUI y su origen de datos. En este ejemplo, el IDE utiliza un objeto `BindingSource` para conectar el objeto `autoresDataGridView` con `LibrosDataSet.Autores`. Para lograr este enlace de datos, el IDE primero establece la propiedad ***DataSource*** de `autoresBindingSource` a `LibrosDataSet`. Esta propiedad especifica el objeto `DataSet`

que contiene los datos que se van a enlazar. Después, el IDE establece la propiedad **DataMember** a Autores. Esta propiedad identifica a una tabla específica dentro del objeto DataSource. Después de configurar el objeto autoresBindingSource, el IDE asigna este objeto a la propiedad **DataSource** de autoresDataGridView para indicar qué es lo que va a mostrar el control DataGridView.

Un objeto BindingSource también administra la interacción entre un control enlazado a datos de la GUI y su origen de datos subyacente. Si usted edita los datos mostrados en un control DataGridView y desea guardar los cambios en el origen de datos, su código debe invocar el método **EndEdit** del objeto BindingSource. Este método aplica los cambios realizados en los datos a través del control de la GUI (es decir, los cambios pendientes) al origen de datos enlazado a ese control. Observe que esto sólo actualiza al objeto DataSet; se requiere un paso adicional para actualizar en forma permanente a la base de datos. En breve veremos un ejemplo de esto, cuando presentemos el código generado por el IDE en el archivo MostrarTabla.cs.

autoresBindingNavigator

Recuerde que un objeto BindingNavigator le permite recorrer (navegar) y manipular (agregar o eliminar filas) los datos enlazados a un control en un formulario. Un objeto BindingNavigator se comunica con un objeto BindingSource (especificado en la propiedad **BindingSource** de BindingNavigator) para llevar a cabo estas acciones en el origen de datos subyacente (es decir, el objeto DataSet). El objeto BindingNavigator no interactúa con el control enlazado a datos, sino que invoca a los métodos de BindingSource que hacen que el control enlazado a datos actualice su presentación de los datos. Por ejemplo, cuando usted hace clic en el botón de BindingNavigator para agregar una nueva fila, el objeto BindingNavigator invoca a un método del objeto BindingSource. Después, el objeto BindingSource agrega una nueva fila a su objeto DataSet asociado. Una vez que se modifica este objeto DataSet, el control DataGridView muestra la nueva fila, ya que DataGridView y BindingNavigator están enlazados al mismo objeto BindingSource (y por ende, al mismo DataSet).

Examinar el código generado en forma automática para MostrarTablaForm

La figura 20.34 presenta el código para MostrarTablaForm. Observe que no necesita escribir este código; el IDE lo genera cuando usted arrastra y suelta la tabla Autores de la ventana **Orígenes de datos** hacia el formulario. Nosotros modificamos el código generado en forma automática para agregar comentarios, dividir las líneas extensas para fines de visualización y eliminar las declaraciones using innecesarias. El IDE también genera una cantidad considerable de código adicional, como el código que define a las clases LibrosDataSet y AutoresTableAdapter, así como el código del diseñador que declara los objetos generados en forma automática en la bandeja de componentes. El código adicional generado por el IDE reside en archivos que pueden verse en el **Explorador de soluciones** al seleccionar **Mostrar todos los archivos**. Nosotros sólo presentamos el código en MostrarTabla.cs, ya que es el único archivo que necesitará modificar.

```

1  // Fig. 20.34: MostrarTabla.cs
2  // Muestra los datos de la tabla de una base de datos en un control DataGridView.
3  using System;
4  using System.Windows.Forms;
5
6  namespace MostrarTabla
7  {
8      public partial class MostrarTablaForm : Form
9      {
10         public MostrarTablaForm()
11         {
12             InitializeComponent();
13         } // fin del constructor
14
15         // El manejador del evento Click para el botón Guardar en el objeto
16         // BindingNavigator guarda los cambios realizados a los datos

```

Figura 20.34 | Código generado en forma automática para mostrar datos de una tabla de la base de datos en un control DataGridView. (Parte 1 de 2).


```

17     private void autoresBindingNavigatorSaveItem_Click(
18         object sender, EventArgs e )
19     {
20         this.Validate();
21         this.autoresBindingSource.EndEdit();
22         this.autoresTableAdapter.Update( this.librosDataSet.Autores );
23     } // fin del método autoresBindingNavigatorSaveItem_Click
24
25     // carga los datos en la tabla librosDataSet.Autores,
26     // que después se muestra en el control DataGridView
27     private void MostrarTablaForm_Load( object sender, EventArgs e )
28     {
29         // TODO: esta línea de código carga datos en la
30         // tabla 'librosDataSet.Autores'. Puede moverla o
31         // quitarla según sea necesario.
32         this.autoresTableAdapter.Fill( this.librosDataSet.Autores );
33     } // fin del método MostrarTablaForm_Load
34 } // fin de la clase MostrarTablaForm
35 } // fin del espacio de nombres MostrarTabla

```

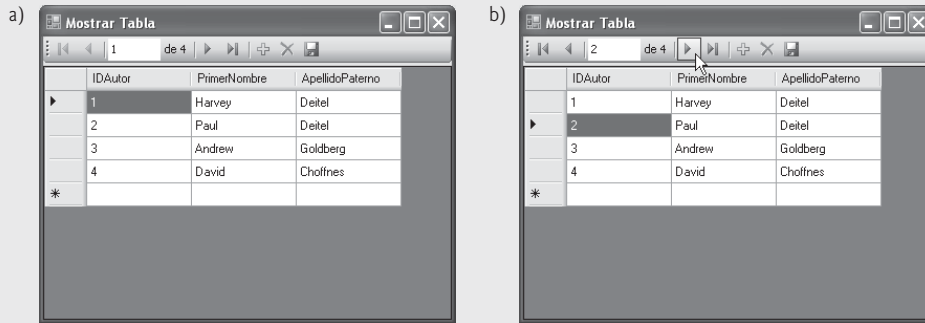


Figura 20.34 | Código generado en forma automática para mostrar datos de una tabla de la base de datos en un control DataGridView. (Parte 2 de 2).

Las líneas 17-23 contienen el manejador del evento Click para el botón **Guardar** en el objeto **AutoresBindingNavigator**. Recuerde que debe hacer clic en este botón para guardar en el origen de datos subyacente (es decir, la tabla **Autores** de la base de datos **Libros**) los cambios realizados a los datos en el control **DataGridView**. El proceso de guardar los cambios es de dos pasos:

1. El objeto **DataSet** asociado con el control **DataGridView** (indicado por su objeto **BindingSource**) debe actualizarse para incluir cualquier cambio realizado por el usuario.
2. La base de datos en el disco debe actualizarse para que coincida con el nuevo contenido del objeto **DataSet**.

Antes de que el manejador de eventos guarde cualquier cambio, la línea 21 invoca a **this.Validate()** para validar los controles en el formulario. Si usted implementa eventos **Validating** o **Validated** para cualquiera de los controles del formulario, estos eventos le permitirán validar la entrada del usuario y posiblemente indicar errores cuando los datos son inválidos. La línea 21 invoca al método **EndEdit** de **autoresBindingSource** para asegurar que el origen de datos asociado con el objeto (**librosDataSet.Autores**) se actualice con cualquier cambio que realice el usuario a la fila actual seleccionada en el control **DataGridView** (por ejemplo, agregar una fila, cambiar el valor de una columna). Cualquier cambio a las otras filas se aplica al objeto **DataSet** cuando usted selecciona otra fila. La línea 22 invoca al método **Update** de **autoresTableAdapter** para escribir la versión modificada de la tabla

Autores (en memoria) a la base de datos de SQL Server en el disco. El método `Update` ejecuta las instrucciones de SQL (encapsuladas en objetos `SqlCommand`) necesarias para hacer que la tabla `Autores` de la base de datos coincida con `LibrosDataSet.Autores`.

El manejador del evento `Load` para `MostrarTablaForm` (líneas 27-33) se ejecuta cuando se carga el programa. Este manejador de eventos llena el objeto `DataSet` en memoria con datos de la base de datos de SQL Server en el disco. Una vez que se llena el objeto `DataSet`, el control de la GUI enlazado a este objeto puede mostrar sus datos. La línea 32 llama al método `Fill` de `autoresTableAdapter` para recuperar información de la base de datos, colocando esta información en el miembro de `DataSet` que se proporciona como argumento. Recuerde que el IDE generó a `autoresTableAdapter` para ejecutar objetos `SqlCommand` a través de la conexión que creamos dentro del **Asistente para la configuración de orígenes de datos**. Por ende, el método `Fill` aquí ejecuta una instrucción `SELECT` para recuperar todas las filas de la tabla `Autores` de la base de datos `Libros`, y después coloca el resultado de esta consulta en `LibrosDataSet.Autores`. Recuerde que la propiedad `DataSource` de `autoresDataGridView` se establece a `autoresBindingSource` (que hace referencia a `LibrosDataSet.Autores`). Por lo tanto, después de que se carga este origen de datos, el control `autoresDataGridView` muestra automáticamente los datos recuperados de la base de datos.

20.7 Consulta de la base de datos Libros

Ahora que ha visto cómo mostrar toda una tabla completa de una base de datos en un control `DataGridView`, le demostraremos cómo ejecutar consultas `SELECT` de SQL específicas en una base de datos y mostrar los resultados. Aunque este ejemplo sólo consulta los datos, la aplicación podría modificarse fácilmente para ejecutar otras instrucciones de SQL. Realice los siguientes pasos para crear la aplicación de ejemplo, que ejecuta consultas personalizadas en la tabla `Titulos` de la base de datos `Libros`.

Paso 1: crear el proyecto

Cree una nueva Aplicación para Windows llamada `MostrarResultadoConsulta`. Cambie el nombre del formulario a `MostrarResultadoConsultaForm` y el nombre de su archivo de código fuente a `MostrarResultadoConsulta.cs`; después establezca la propiedad `Text` del formulario a `Mostrar resultado de consulta`.

Paso 2: agregar un origen de datos al proyecto

Realice los pasos en la sección 20.6.1 para incluir la base de datos `Libros` como un origen de datos en el proyecto.

Paso 3: crear un control DataGridView para mostrar la tabla Titulos

Arrastre el nodo `Titulos` de la ventana **Orígenes de datos** en el formulario para crear un control `DataGridView` que muestre todo el contenido de la tabla `Titulos`.

Paso 4: agregar consultas personalizadas al objeto TitulosTableAdapter

Recuerde que al invocar al método `Fill` de `TableAdapter` se llena el objeto `DataSet` que recibe como argumento con todo el contenido de la tabla de la base de datos que corresponde a ese objeto `TableAdapter`. Para poblar un miembro de `DataSet` (es decir, un objeto `DataTable`) con sólo una porción de una tabla (por ejemplo, con fechas de copyright del 2006), debe agregar un método al objeto `TableAdapter` que llene el objeto `DataTable` especificado con los resultados de una consulta personalizada. El IDE proporciona el **Asistente para la configuración de consultas de TableAdapter** para realizar esta tarea. Para abrir este asistente, primero haga clic con el botón derecho del ratón en el nodo `LibrosDataSet.xsd` del **Explorador de soluciones** y seleccione la opción **Ver diseñador**. También puede hacer clic en el icono **Editar DataSet con el Diseñador** (). Cualquiera de estas acciones abre el **Diseñador de DataSet** (figura 20.35), que muestra una representación visual del objeto `LibrosDataSet` (es decir, las tablas `ISBNAutor`, `Autores` y `Titulos`, y las relaciones entre ellas). El **Diseñador de DataSet** lista las columnas de cada tabla y el objeto `TableAdapter` generado en forma automática que accede a la tabla. Para seleccionar el objeto `TitulosTableAdapter`, haga clic en su nombre, después haga clic con el botón derecho sobre el nombre y seleccione **Agregar Query...** para iniciar el **Asistente para la configuración de consultas de TableAdapter** (figura 20.36).

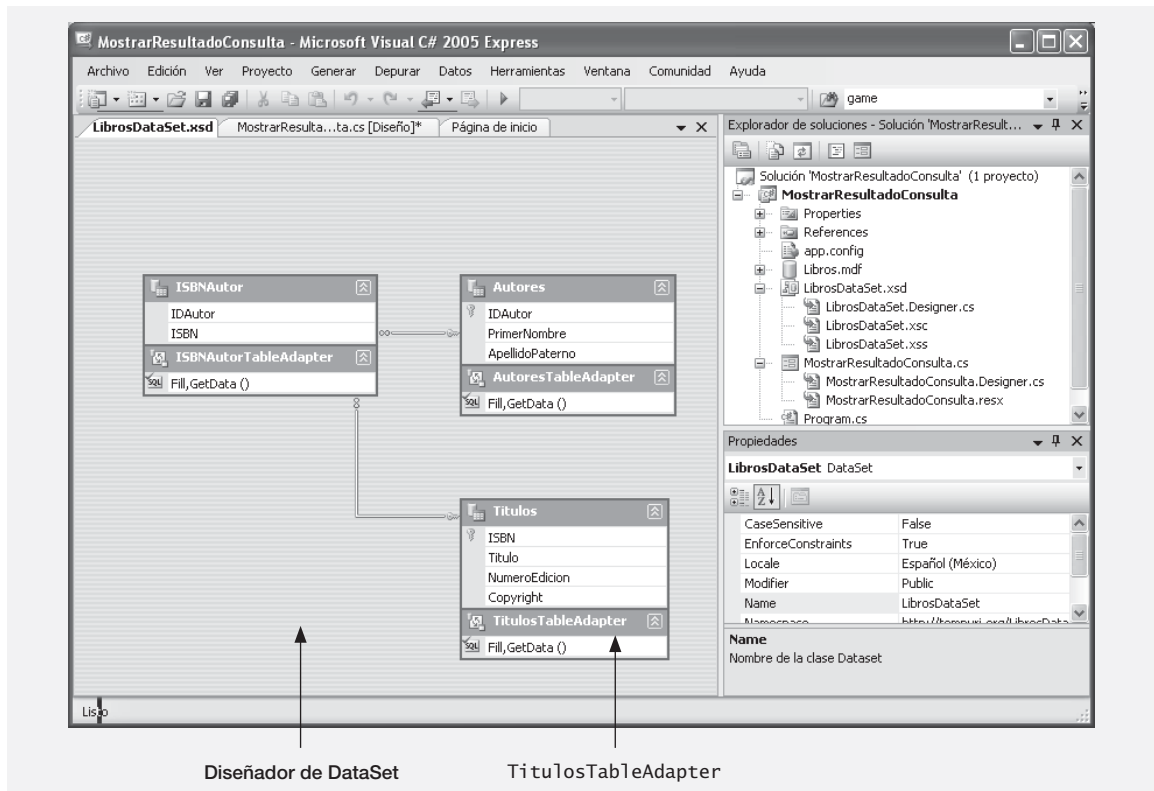


Figura 20.35 | Vista del conjunto de datos LibrosDataSet en el Diseñador de DataSet.

Paso 5: elegir la forma en que el objeto TableAdapter debe acceder a la base de datos

En la primera pantalla del asistente (figura 20.36), mantenga la opción predeterminada **Usar instrucciones SQL** y haga clic en **Siguiente**.

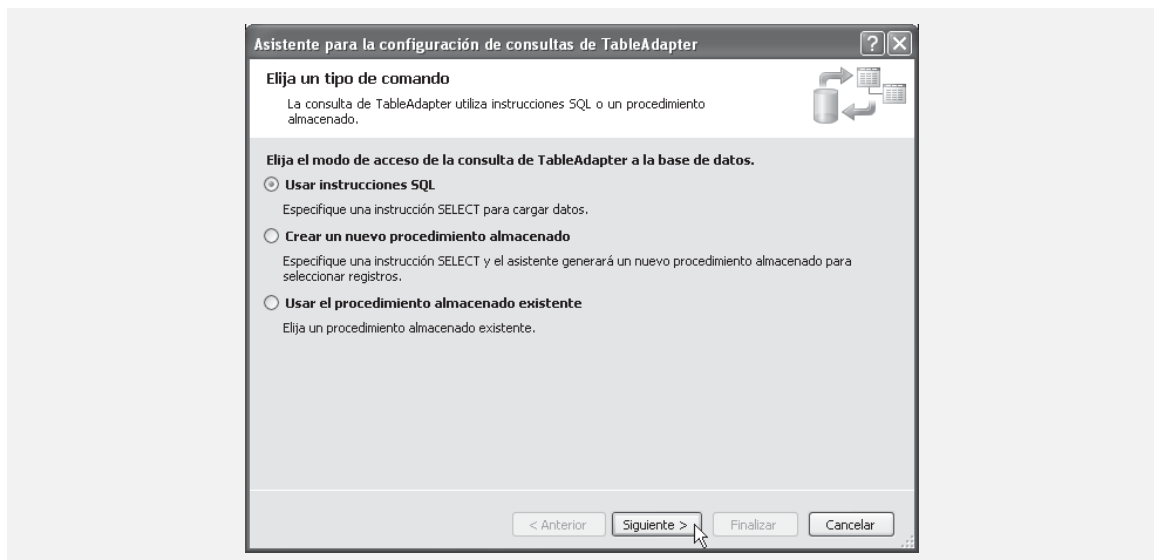


Figura 20.36 | Asistente para la configuración de consultas de TableAdapter para agregar una consulta a un objeto TableAdapter.

Paso 6: elegir un tipo de consulta

En la siguiente pantalla del asistente (figura 20.37), mantenga la opción predeterminada **SELECT** que devuelve filas y haga clic en **Siguiente**.

Paso 7: especificar una instrucción SELECT para la consulta

La siguiente pantalla (figura 20.38) le pide que escriba una consulta que se utilizará para recuperar datos de la base de datos Libros. Observe que la instrucción **SELECT** predeterminada antepone el prefijo “dbo.” a **Ti tulos**. Este prefijo significa “propietario de la base de datos”, e indica que la tabla **Ti tulos** pertenece al propietario de la base de datos (es decir, usted). En casos en los que se necesita hacer referencia a una tabla cuyo propietario es otro usuario del sistema, este prefijo se sustituye por el nombre de usuario del propietario. Usted puede modificar la instrucción de **SQL** en el cuadro de texto aquí (usando la sintaxis de **SQL** que vimos en la sección 20.4), o puede hacer clic en **Generador de consultas...** para diseñar y probar la consulta utilizando una herramienta visual.

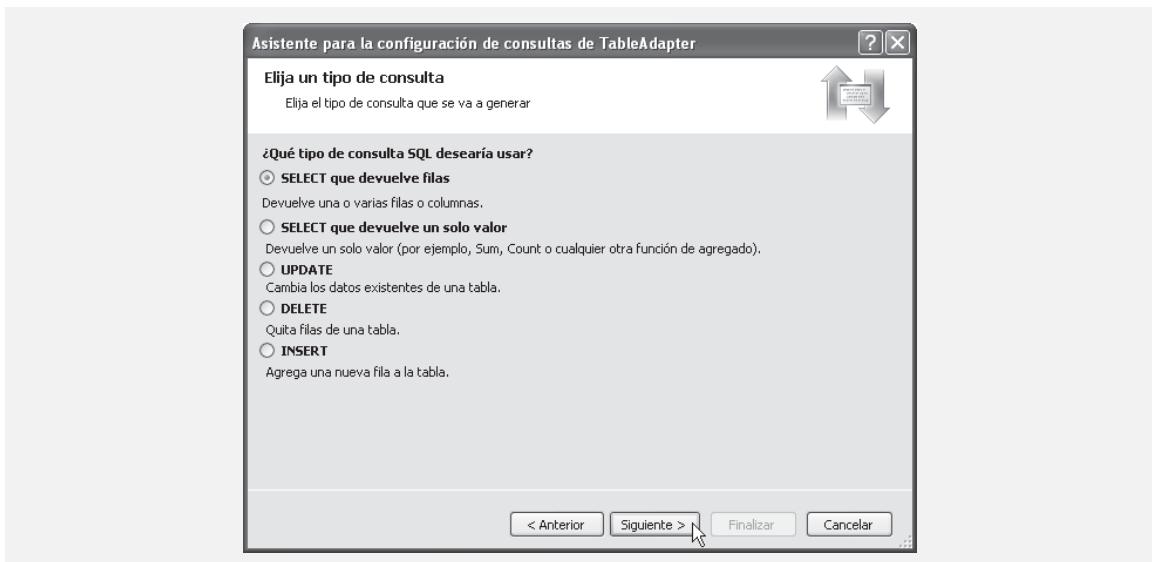


Figura 20.37 | Elegir el tipo de consulta que se va a generar para el objeto `TableAdapter`.

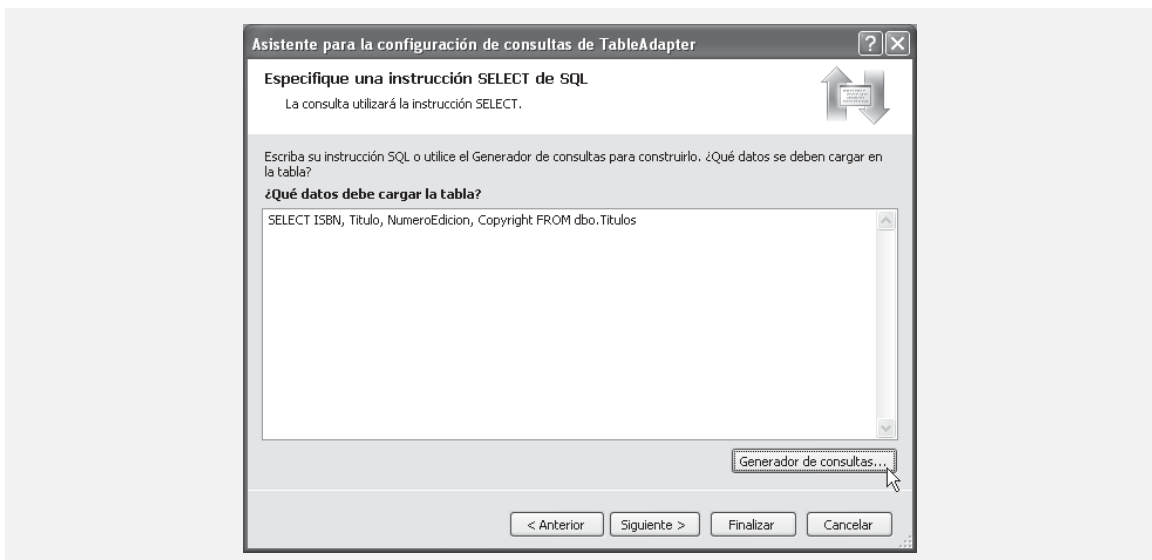


Figura 20.38 | Especificar una instrucción **SELECT** para la consulta.

Paso 8: crear una consulta con el Generador de consultas

Haga clic en el botón **Generador de consultas...** para abrir el **Generador de consultas** (figura 20.39). La porción superior de la ventana **Generador de consultas** contiene un cuadro que lista las columnas de la tabla **Titulos**. Todas las columnas están seleccionadas de manera predeterminada (figura 20.39a), lo cual indica que la consulta debe devolver todas las columnas. La porción intermedia de la ventana contiene una tabla en la que cada fila corresponde a una columna en la tabla **Titulos**. A la derecha de los nombres de las columnas, hay columnas en las que usted puede escribir valores o hacer selecciones para modificar la consulta. Por ejemplo, para crear una consulta que seleccione sólo libros que tengan fecha de copyright del 2006, escriba el valor 2006 en la columna **Filtro** de la fila **Copyright**. Observe que el **Generador de consultas** modifica su entrada para que sea “= '2006'” y agrega una

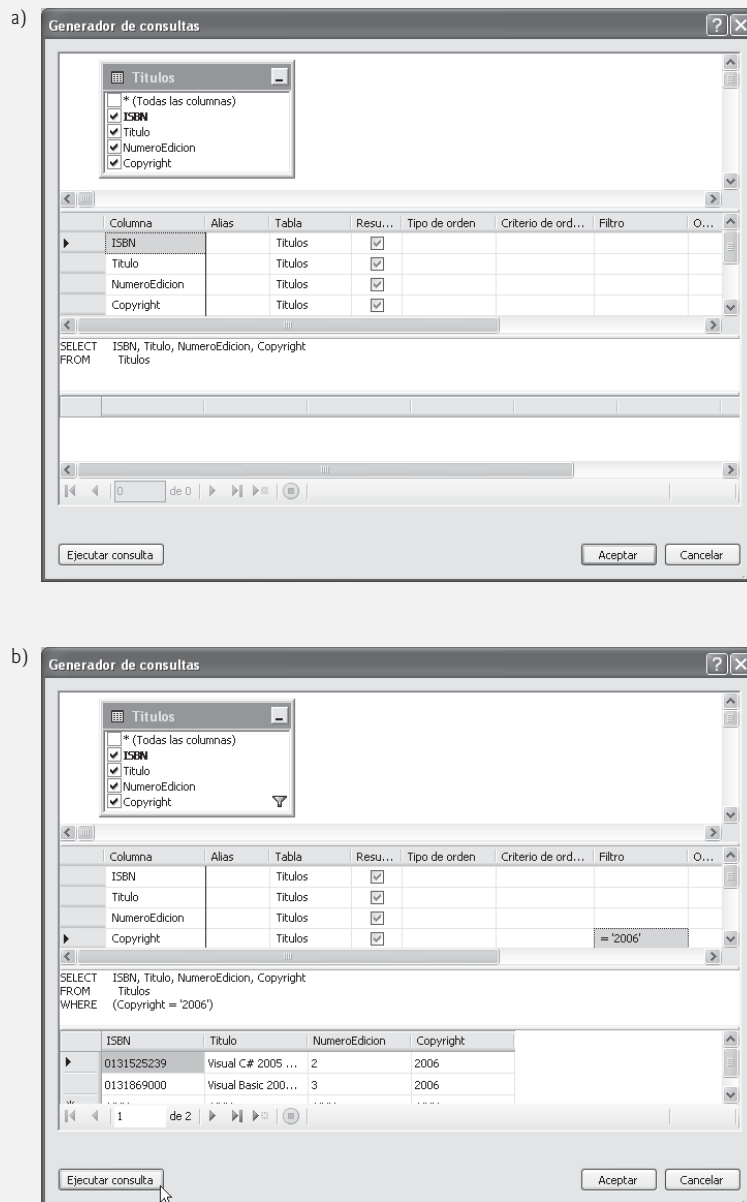


Figura 20.39 | El **Generador de consultas** después de agregar una cláusula **WHERE** al escribir un valor en la columna **Filtro**.

cláusula WHERE apropiada a la instrucción SELECT que se muestra en la parte intermedia de la figura 20.39b. Haga clic en el botón **Ejecutar consulta** para probar la consulta y mostrar los resultados en la parte inferior de la ventana del **Generador de consultas**. Para obtener más información sobre el **Generador de consultas**, vea los sitios msdn2.microsoft.com/library/ms172013.aspx (en inglés) y msdn2.microsoft.com/es-es/library/ms141087.aspx (en español).

Paso 9: cerrar el Generador de consultas

Haga clic en **Aceptar** para cerrar el **Generador de consultas** y regresar al **Asistente para la configuración de consultas de TableAdapter** (figura 20.40), el cual ahora muestra la consulta de SQL creada en el paso anterior. Haga clic en **Siguiente** para continuar.

Paso 10: establecer los nombres de los métodos generados en forma automática que realizan la consulta

Después de especificar la consulta de SQL, debe nombrar los métodos que el IDE generará para realizar la consulta (figura 20.41). Se generan dos métodos de manera predeterminada: un “método Fill” que llena un parámetro DataTable con el resultado de la consulta, y un “método Get” que devuelve un nuevo objeto DataTable lleno con el resultado de la consulta. Los cuadros de texto para escribir los nombres para estos métodos se llenan previamente con FillBy y GetDataBy, respectivamente. Modifique estos nombres por LlenarConCopyright2006 y ObtenerDatosConCopyright2006, como se muestra en la figura 20.41. Por último, haga clic en **Finalizar** para completar el asistente y regresar al **Diseñador de DataSet** (figura 20.42). Observe que ahora estos métodos se listan en la sección **TitulosTableAdapter** del cuadro que representa a la tabla **Titulos**.

Paso 11: agregar una consulta adicional

Repita los *pasos 4-10* para agregar otra consulta que seleccione todos los libros cuyos títulos terminen con el texto “How to Program”, y que almacene los resultados por título en orden ascendente (vea la sección 20.4.3). En el **Generador de consultas**, escriba LIKE ‘%How to Program’ en la columna **Filtro** de la fila **Título**. Para especificar el orden, seleccione **Ascendente** en la columna **Tipo de orden** de la fila **Título**. En el paso final del **Asistente para la configuración de consultas de TableAdapter**, cambie los nombres de los métodos Fill y Get por **LlenarConLibrosHowToProgram** y **ObtenerDatosParaLibrosHowToProgram**, respectivamente.

Paso 12: agregar un control ComboBox al formulario

Regrese a la vista de **Diseño** y agregue debajo del control DataGridView un control ComboBox llamado **consultasComboBox** al formulario. Los usuarios utilizarán este control para elegir una consulta SELECT a ejecutar,

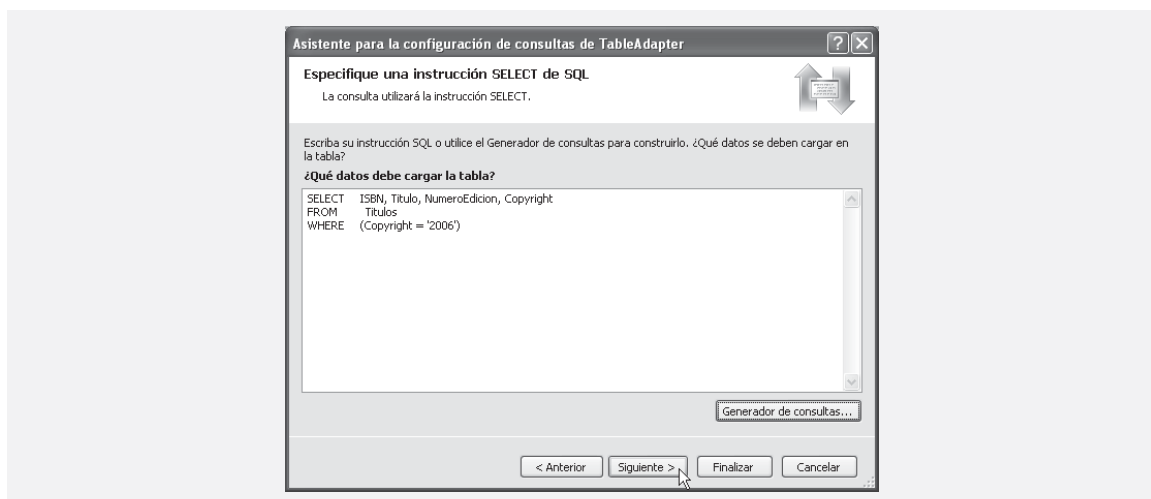


Figura 20.40 | La instrucción SELECT creada con el **Generador de consultas**.

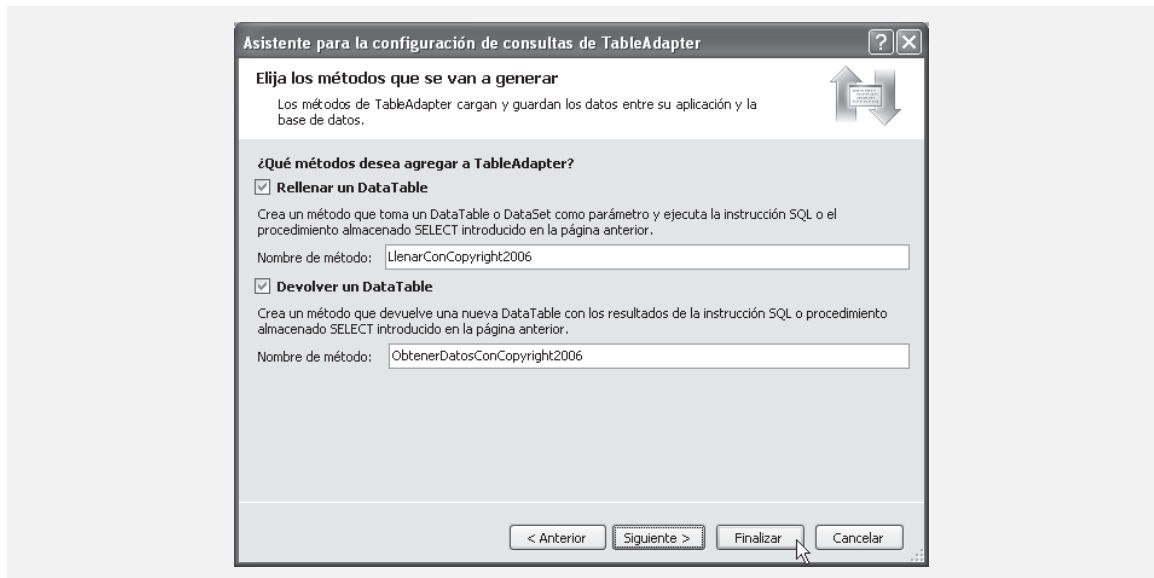


Figura 20.41 | Especificar nombres para los métodos que se van a agregar a `TitulosTableAdapter`.

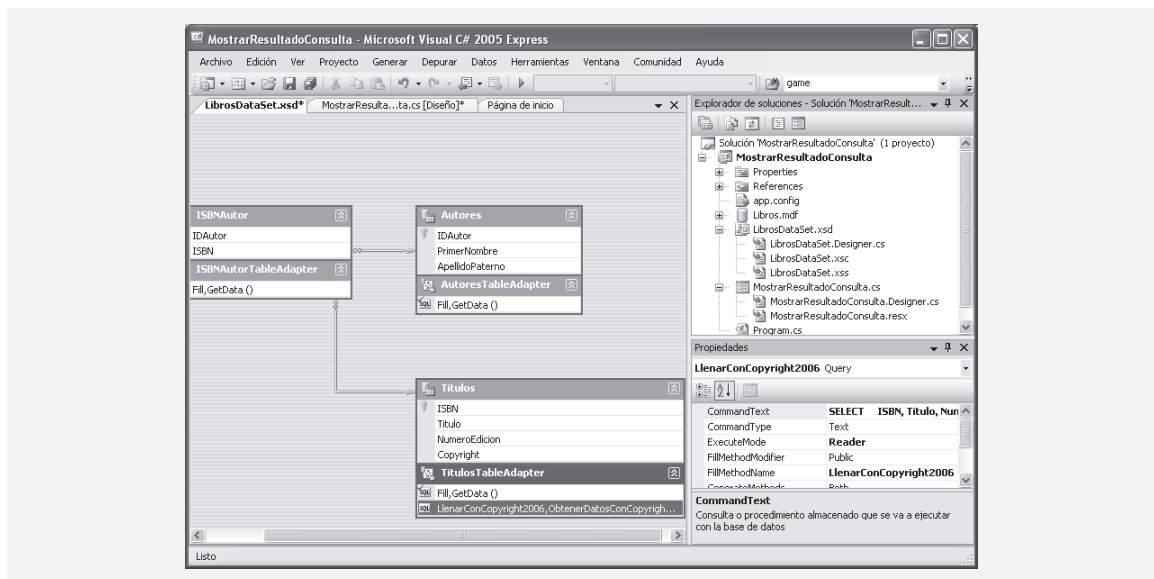


Figura 20.42 | El Diseñador de DataSet, después de agregar métodos `Fill` y `Get` a `TitulosTableAdapter`.

cuyo resultado se mostrará en el control `DataGridView`. Agregue tres elementos a `consultasComboBox`: uno que coincida con cada una de las tres consultas que ahora puede realizar el objeto `TitulosTableAdapter`:

```
SELECT ISBN, Titulo, NumeroEdicion, Copyright FROM Titulos
```

```
SELECT ISBN, Titulo, NumeroEdicion, Copyright FROM Titulos  
WHERE ( Copyright = '2006' )
```

```
SELECT ISBN, Titulo, NumeroEdicion, Copyright FROM Titulos  
WHERE ( Titulo LIKE '%How to Program' ) ORDER BY Titulo
```

Paso 13: personalizar el manejador de eventos Load del formulario

Agregue una línea de código al manejador de eventos `MostrarResultadoConsultaForm_Load` generado en forma automática, para establecer la propiedad `SelectedIndex` inicial de `consultasComboBox` en 0. Recuerde que el manejador del evento `Load` llama al método `Fill` de manera predeterminada, que ejecuta la primera consulta (el elemento en el índice 0). Por lo tanto, al establecer la propiedad `SelectedIndex` el control `ComboBox` muestra la consulta que se realiza en un principio, cuando se carga `MostrarResultadoConsultaForm` por primera vez.

Paso 14: programar un manejador de eventos para el control ComboBox

Ahora debe escribir código que ejecute la consulta apropiada, cada vez que el usuario seleccione un elemento distinto de `consultasComboBox`. Haga doble clic en el control `consultasComboBox` en vista de **Diseño** para generar un manejador de eventos `consultasComboBox_SelectedIndexChanged` (líneas 42-61) en el archivo `MostrarResultadoConsulta.cs` (figura 20.43). Después agregue al manejador de eventos una instrucción `switch` que invoque al método de `titulosTableAdapter` que ejecute la consulta asociada con la selección actual del control `ComboBox` (líneas 47-60). Recuerde que el método `Fill` (línea 50) ejecuta una consulta `SELECT` que selecciona todas las filas, el método `LlenarConCopyright2006` (líneas 53-54) ejecuta una consulta `SELECT` que selecciona todas las filas en las que el año de copyright es 2006 y el método `LlenarConLibrosHowToProgram` (líneas 57-58) ejecuta una consulta que selecciona todas las filas que tienen “How to Program” al final de sus títulos, y las ordena en forma ascendente, por título. Cada método llena `librosDataSet.Titulos` sólo con las filas devueltas en la columna correspondiente. Gracias a las relaciones de enlace de datos creadas por el IDE, al volver a llenar `librosDataSet.Titulos` el control `TitulosDataGridview` muestra el resultado de la consulta seleccionada, sin necesidad de código adicional.

La figura 20.43 también muestra la salida para `MostrarResultadoConsultaForm`. La figura 20.43a muestra el resultado de recuperar todas las filas de la tabla `Titulos`. La figura 20.43(b) demuestra la segunda consulta, que recupera sólo las filas para los libros con copyright del 2006. Por último, la figura 20.43(c) demuestra la tercera consulta, que selecciona filas para los libros `How to Program`, y las ordena en forma ascendente por título.

```

1  // Fig. 20.43: MostrarResultadoConsulta.cs
2  // Muestra el resultado de una consulta seleccionada por el usuario en un control
   DataGridView.
3  using System;
4  using System.Windows.Forms;
5
6  namespace MostrarResultadoConsulta
7  {
8      public partial class MostrarResultadoConsultaForm : Form
9      {
10         public MostrarResultadoConsultaForm()
11         {
12             InitializeComponent();
13         } // fin del constructor de MostrarResultadoConsultaForm
14
15         // el manejador del evento Click para el botón Guardar en el objeto
16         // BindingNavigator guarda los cambios realizados en los datos
17         private void titulosBindingNavigatorSaveItem_Click(
18             object sender, EventArgs e )
19         {
20             this.Validate();
21             this.titulosBindingSource.EndEdit();
22             this.titulosTableAdapter.Update( this.librosDataSet.Titulos );
23         } // fin del método titulosBindingNavigatorSaveItem_Click

```

Figura 20.43 | Muestra el resultado de una consulta seleccionada por el usuario en un control `DataGridView`. (Parte I de 3).


```

24
25 // carga datos en la tabla librosDataSet.Titulos,
26 // que después se muestra en el control DataGridView
27 private void MostrarResultadoConsultaForm_Load(
28     object sender, EventArgs e )
29 {
30     // TODO: esta línea de código carga datos en la
31     // tabla 'librosDataSet.Titulos' Puede moverla o
32     // quitarla según sea necesario.
33     this.titulosTableAdapter.Fill( this.librosDataSet.Titulos );
34
35     // establece el control ComboBox para que muestre la consulta
36     // predeterminada que selecciona todos los libros de la tabla Titulos
37     consultasComboBox.SelectedIndex = 0;
38 } // fin del método MostrarResultadoConsultaForm_Load
39
40 // carga datos en la tabla librosDataSet.Titulos, con base
41 // en la consulta seleccionada por el usuario
42 private void consultasComboBox_SelectedIndexChanged(
43     object sender, EventArgs e )
44 {
45     // llena el objeto DataTable de Titulos con
46     // el resultado de la consulta seleccionada
47     switch ( consultasComboBox.SelectedIndex )
48     {
49         case 0: // todos los libros
50             titulosTableAdapter.Fill( librosDataSet.Titulos );
51             break;
52         case 1: // libros con año de copyright 2006
53             titulosTableAdapter.LlenarConCopyright2006(
54                 librosDataSet.Titulos );
55             break;
56         case 2: // libros How to Program, ordenados por Título
57             titulosTableAdapter.LlenarConLibrosHowToProgram(
58                 librosDataSet.Titulos );
59             break;
60     } // fin de switch
61 } // fin del método consultasComboBox_SelectedIndexChanged
62 } // fin de la clase MostrarResultadoConsultaForm
63 } // fin del espacio de nombres MostrarResultadoConsulta

```

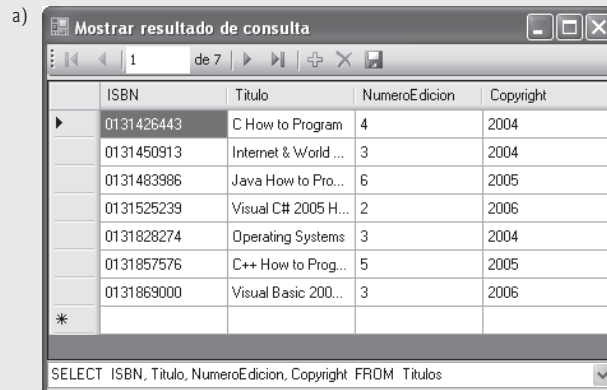


Figura 20.43 | Mostrar el resultado de una consulta seleccionada por el usuario en un control DataGridView. (Parte 2 de 3).

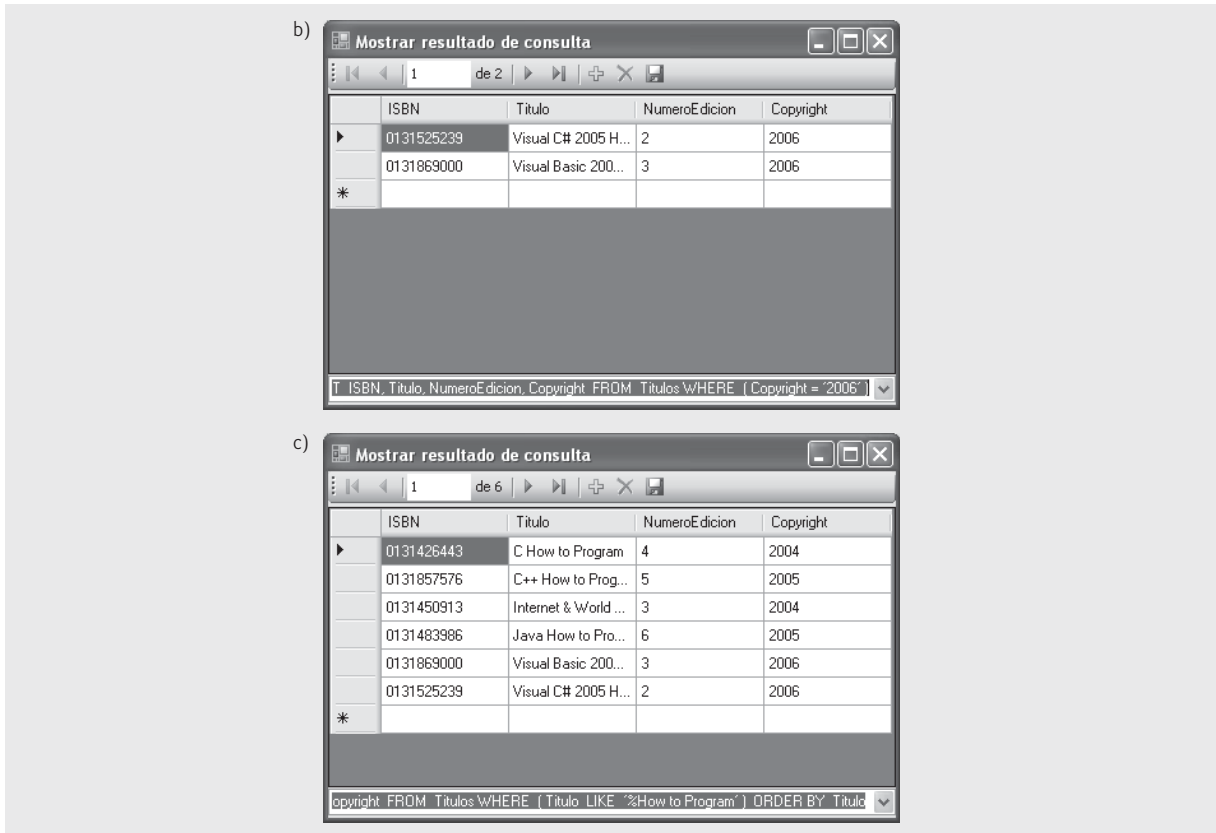


Figura 20.43 | Mostrar el resultado de una consulta seleccionada por el usuario en un control DataGridView. (Parte 3 de 3).

20.8 Programación con ADO.NET: caso de estudio de libreta de direcciones

Nuestro siguiente ejemplo implementa una aplicación simple de libreta de direcciones, la cual permite a los usuarios insertar filas, localizar filas y actualizar la base de datos `LibretaDirecciones.mdf` de SQL Server.

La aplicación `LibretaDirecciones` (figura 20.44) cuenta con una GUI, a través de la cual los usuarios pueden ejecutar instrucciones de SQL en la base de datos. No obstante, en vez de mostrar una tabla de la base

```

1 // Fig. 20.44: LibretaDirecciones.cs
2 // Permite a los usuarios manipular una libreta de direcciones.
3 using System;
4 using System.Windows.Forms;
5
6 namespace LibretaDirecciones
7 {
8     public partial class LibretaDireccionesForm : Form
9     {
10         public LibretaDireccionesForm()
11         {

```

Figura 20.44 | Aplicación `LibretaDirecciones` le permite manipular entradas en una base de datos de libreta de direcciones. (Parte 1 de 3).

```

12     InitializeComponent();
13 } // fin del constructor de LibretaDireccionesForm
14
15 // el manejador del evento Click para el botón Guardar en el objeto
16 // BindingNavigator guarda los cambios realizados a los datos
17 private void direccionesBindingNavigatorSaveItem_Click(
18     object sender, EventArgs e )
19 {
20     this.Validate();
21     this.direccionesBindingSource.EndEdit();
22     this.direccionesTableAdapter.Update(
23         this.libretaDireccionesDataSet.Direcciones );
24 } // fin del método direccionesBindingNavigatorSaveItem_Click
25
26 // carga datos en la tabla libretaDireccionesDataSet.Direcciones
27 private void LibretaDireccionesForm_Load( object sender, EventArgs e )
28 {
29     // TODO: esta línea de código carga datos en la
30     // tabla 'libretaDireccionesDataSet.Direcciones'.
31     // Puede moverla o quitarla según sea necesario.
32     this.direccionesTableAdapter.Fill(
33         this.libretaDireccionesDataSet.Direcciones );
34 } // fin del método LibretaDireccionesForm_Load
35
36 // carga los datos para las filas con el apellido paterno especificado
37 // en la tabla libretaDireccionesDataSet.Direcciones
38 private void buscarButton_Click( object sender, EventArgs e )
39 {
40     // llena el objeto DataTable de DataSet sólo con filas
41     // que contengan el apellido paterno especificado por el usuario
42     direccionesTableAdapter.LlenarPorApellidoPaterno(
43         libretaDireccionesDataSet.Direcciones, buscarTextBox.Text );
44 } // fin del método buscarButton_Click
45
46 // vuelve a cargar libretaDireccionesDataSet.Direcciones con todas las filas
47 private void explorarTodoButton_Click( object sender, EventArgs e )
48 {
49     // llena el objeto DataTable de DataSet con todas las filas en la base de datos
50     direccionesTableAdapter.Fill( libretaDireccionesDataSet.Direcciones );
51
52     buscarTextBox.Text = ""; // borra el control buscarTextBox
53 } // fin del método explorarTodoButton_Click
54 } // fin de la clase LibretaDireccionesForm
55 } // fin del espacio de nombres LibretaDirecciones

```

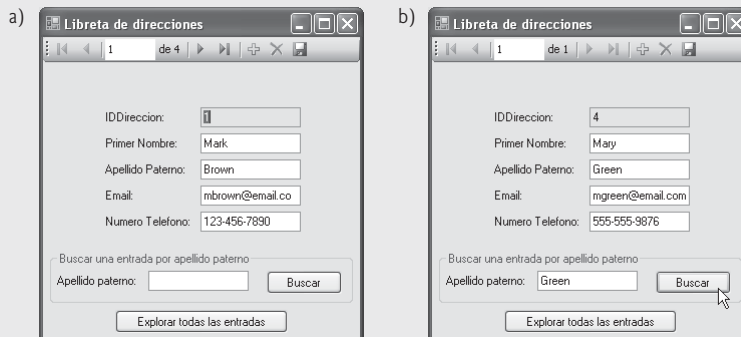


Figura 20.44 | Aplicación LibretaDirecciones le permite manipular entradas en una base de datos de libreta de direcciones. (Parte 2 de 3).



Figura 20.44 | Aplicación LibretaDirecciones le permite manipular entradas en una base de datos de libreta de direcciones. (Parte 3 de 3).

de datos en un control `DataGridView`, este ejemplo presenta los datos de una tabla, una fila a la vez, usando un conjunto de controles `TextBox` que muestran los valores de cada una de las columnas de la fila. Un control `BindingNavigator` le permite controlar cuál fila de la tabla se puede ver en cualquier momento dado. El control `BindingNavigator` también le permite agregar nuevas filas, eliminar filas y guardar cambios a los datos visualizados. Observe que las líneas 17-34 en la figura 20.44 son similares a las correspondientes líneas del código en los ejemplos anteriores de este capítulo. En breve hablaremos sobre la funcionalidad adicional de la aplicación y el código en las líneas 38-53 que soporta esta funcionalidad. Empezaremos por mostrarle los pasos requeridos para crear esta aplicación.

Paso 1: agregar la base de datos al proyecto

Como en los ejemplos anteriores, debe empezar por agregar la base de datos al proyecto. Después de agregar el archivo `LibretaDirecciones.mdf` como origen de datos, la ventana **Orígenes de datos** listará el objeto `LibretaDireccionesDataSet`, que contiene una tabla llamada `Direcciones`.

Paso 2: indicar que el IDE debe crear un conjunto de controles `Label` y `TextBox` para mostrar cada fila de datos

En las secciones anteriores, usted arrastraba un nodo de la ventana **Orígenes de datos** hacia el formulario para crear un control `DataGridView` enlazado al miembro del origen de datos representado por ese nodo. El IDE le permite especificar el tipo de control(es) que va a crear cuando usted arrastra y suelta un miembro de origen de datos en un formulario. En vista de **Diseño**, haga clic en el nodo `Direcciones` en la ventana **Orígenes de datos** (figura 20.45). Observe que este nodo se convierte en una lista desplegable cuando lo selecciona. Haga clic en la flecha hacia abajo para ver los elementos en la lista. Al principio, el icono a la izquierda de `DataGridView` estará resaltado en color azul, ya que el control predeterminado que se enlaza a una tabla es `DataGridView` (como vimos en los ejemplos anteriores). Seleccione la opción **Detalles** en la lista desplegable, para indicar que el IDE debe crear un conjunto de pares de controles `Label-TextBox` para cada par de valores nombre-columna, cuando arrastre y suelte la tabla `Direcciones` en el formulario. (En la figura 20.46 puede ver los resultados de esta operación.) La lista desplegable contiene sugerencias para los controles que van a mostrar los datos de la tabla, pero también puede elegir la opción **Personalizar...** para seleccionar otros controles que sean capaces de enlazarse con los datos de una tabla.

Paso 3: arrastrar el nodo de origen de datos `Direcciones` hacia el formulario

Arrastre el nodo `Direcciones` de la ventana **Orígenes de datos** hacia el formulario (figura 20.46). El IDE creará una serie de controles `Label` y `TextBox`, ya que usted seleccionó **Detalles** en el paso anterior. Como en los ejemplos anteriores, el IDE también crea un objeto `BindingNavigator` y los demás componentes en la bandeja de componentes. El IDE establece el texto de cada control `Label` con base en su correspondiente nombre de columna en la tabla de la base de datos, y utiliza expresiones regulares para insertar espacios en nombres de columnas con varias palabras, para que los controles `Label` sean más legibles.

Paso 4: hacer que el cuadro de texto IDDireccion sea de Sólo lectura

La columna IDDireccion de la tabla Direcciones es una columna de identidad autoincremental, por lo que no se debe permitir que los usuarios editen los valores en esta columna. Seleccione el control TextBox para IDDireccion y establezca su propiedad ReadOnly a true, usando la ventana **Propiedades**. Tal vez necesite hacer clic en una parte vacía del formulario para deseleccionar los demás controles Label y TextBox y poder seleccionar el control TextBox IDDireccion.

Paso 5: ejecutar la aplicación

Ejecute la aplicación y experimente con los controles en el objeto BindingNavigator que se encuentra en la parte superior de la ventana. Al igual que los ejemplos anteriores, este ejemplo llena un objeto DataSet (específicamente, un objeto LibretaDireccionesDataSet) con todas las filas de una tabla de la base de datos (es decir, Direcciones). Sin embargo, sólo aparece una sola fila del objeto DataSet en cualquier momento dado. Los botones del objeto BindingNavigator similares a los del reproductor de CD o DVD le permiten cambiar la fila que se muestra en un momento dado (es decir, cambian los valores en cada uno de los controles TextBox). Los botones para agregar una fila, eliminar una fila y guardar cambios también realizan sus tareas designadas. Al agregar una fila se borran los controles TextBox y aparece un nuevo ID autoincrementado (por ejemplo, 5) en el control TextBox a la derecha de ID Dirección. Después de escribir datos, haga clic en el botón **Guardar** para registrar la nueva fila en la base de datos. Después de cerrar y reiniciar la aplicación, debe haber todavía cinco filas. Elimine la nueva fila haciendo clic en el botón apropiado y después guarde los cambios.

Paso 6: agregar una consulta al objeto DireccionesTableAdapter

Aunque el objeto BindingNavigator le permite explorar la libreta de direcciones, sería más conveniente poder buscar una entrada específica con base en el apellido paterno. Para agregar esta funcionalidad a la aplicación, debe agregar una nueva consulta al objeto AddressesTableAdapter, usando el **Asistente para la configuración de consultas de TableAdapter**. Haga clic en el icono **Editar DataSet con el Diseñador** (🔗) en la ventana **Orígenes de datos**. Seleccione el cuadro que representa al objeto DireccionesTableAdapter. Haga clic con el botón derecho en el nombre del objeto TableAdapter y seleccione **Agregar Query....** En el **Asistente para la configuración de consultas de TableAdapter**, mantenga la opción predeterminada **Usar instrucciones SQL** y haga clic en **Siguiente**. En la siguiente pantalla, mantenga la opción predeterminada **SELECT que devuelve filas** y haga clic en **Siguiente**. En vez de usar

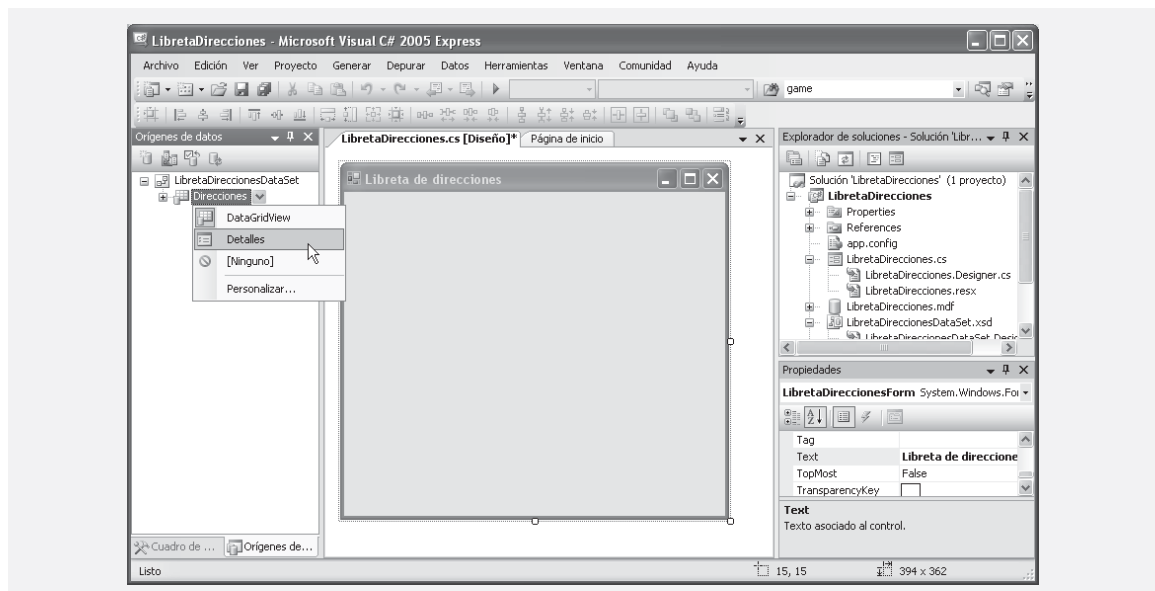


Figura 20.45 | Seleccionar el (los) control(es) que se va(n) a crear cuando se arrastra y suelta un miembro de origen de datos en el formulario.

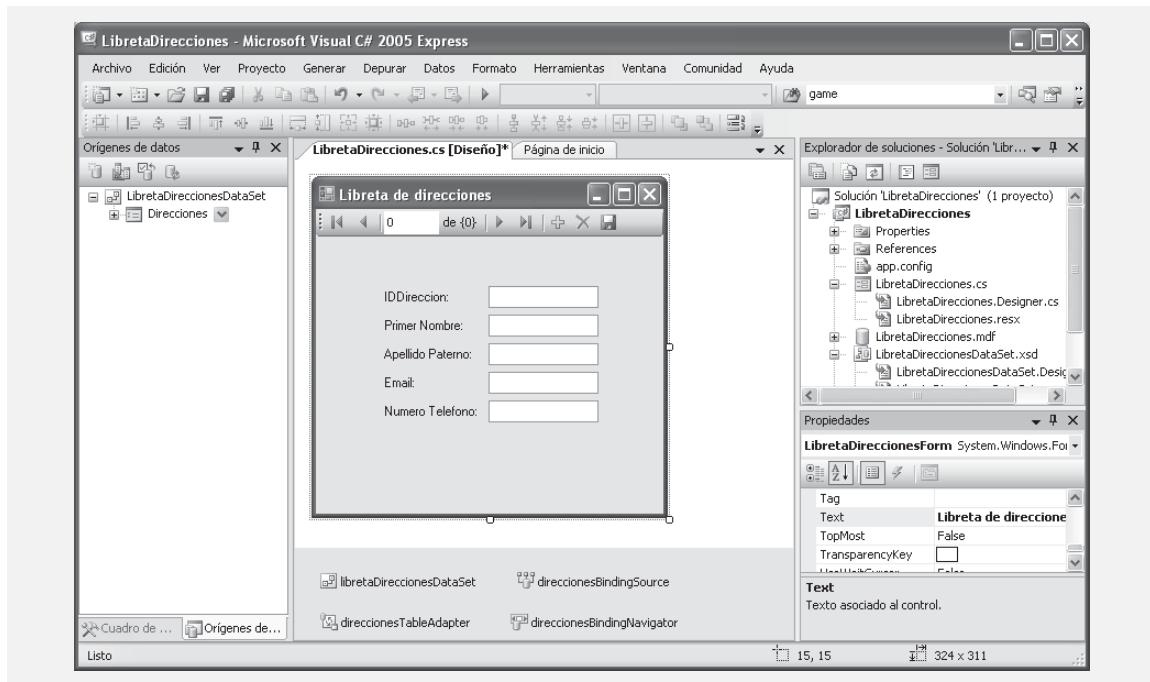


Figura 20.46 | Mostrar una tabla en un formulario, usando una serie de controles Label y TextBox.

el **Generador de consultas** para formar su consulta (como hicimos en el ejemplo anterior), modifique la consulta directamente en el cuadro de texto del asistente. Adjunte la cláusula “WHERE ApellidoPaterno = @apellidoPaterno” al final de la consulta predeterminada. Observe que @apellidoPaterno es un parámetro que se sustituirá con un valor cuando se ejecute la consulta. Haga clic en **Siguiente**, después escriba **LlenarPorApellidoPaterno** y **ObtenerDatosPorApellidoPaterno** como los nombres de los dos métodos que generará el asistente. La consulta contiene un parámetro, por lo que cada uno de estos métodos recibirá un parámetro para establecer el valor de @apellidoPaterno en la consulta. Haga clic en **Finalizar** para completar el asistente y regresar al **Diseñador de DataSet** (figura 20.47). Observe que los métodos **Fill** y **Get** recién creados aparecen bajo el objeto **DireccionesTableAdapter** y que el parámetro @apellidoPaterno está listado a la derecha de los nombres de los métodos.

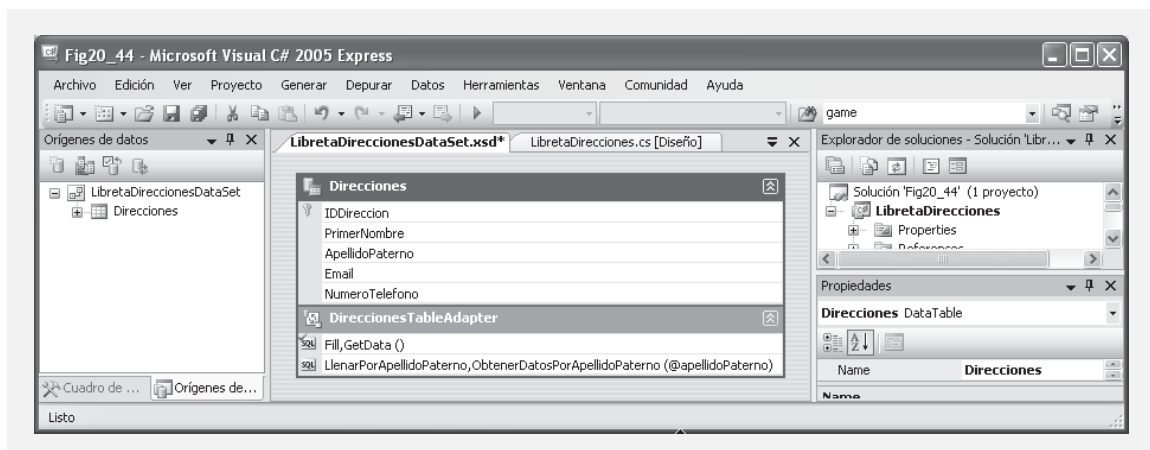


Figura 20.47 | El **Diseñador de DataSet** para **LibretaDireccionesDataSet**, después de agregar una consulta a **DireccionesTableAdapter**.

Paso 7: agregar controles para permitir a los usuarios especificar un apellido paterno para localizarlo

Ahora que ha creado una consulta para localizar filas con un apellido paterno específico, agregue controles para permitir a los usuarios escribir un apellido paterno y ejecutar esta consulta. Vaya a la vista de **Diseño** (figura 20.48) y agregue al formulario un control **Label** llamado `buscarLabel`, un control **TextBox** llamado `buscarTextBox` y un control **Button** llamado `buscarButton`. Coloque estos controles dentro de un control **GroupBox** llamado `buscarGroupBox` y después establezca su propiedad **Text** a **Buscar una entrada por apellido paterno**. Establezca las propiedades de los controles **Label** y **Button** según se muestra en la figura 20.48.

Paso 8: programar un manejador de eventos que localice el apellido paterno especificado por el usuario

Haga doble clic en `buscarButton` para agregar un manejador del evento **Click** para este objeto **Button**. En el manejador de eventos, escriba las siguientes líneas de código (líneas 42-43 de la figura 20.44):

```
direccionesTableAdapter.LlenarPorApellidoPaterno(
    libretaDireccionesDataSet.Direcciones, buscarTextBox.Text );
```

El método `LlenarPorApellidoPaterno` sustituye los datos actuales en `libretaDireccionesDataSet.Direcciones` con datos sólo para aquellas filas que tengan el apellido paterno que se escribió en `buscarTextBox`. Observe que, al invocar a `LlenarPorApellidoPaterno`, debe pasar el objeto **DataTable** a llenar, así como un argumento que especifique el apellido paterno a buscar. Este argumento se convierte en el valor del parámetro `@apellidoPaterno` en la instrucción **SELECT** creada en el *paso 6*. Inicie la aplicación para probar la nueva funcionalidad. Observe que cuando busca una entrada específica (es decir, escriba un apellido paterno y haga clic en **Buscar**), el objeto **BindingNavigator** permite al usuario explorar sólo las filas que contienen el apellido paterno especificado. Esto se debe a que el origen de datos enlazado a los controles del formulario (es decir, `libretaDireccionesDataSet.Direcciones`) ha cambiado, y ahora contiene sólo un número limitado de filas.

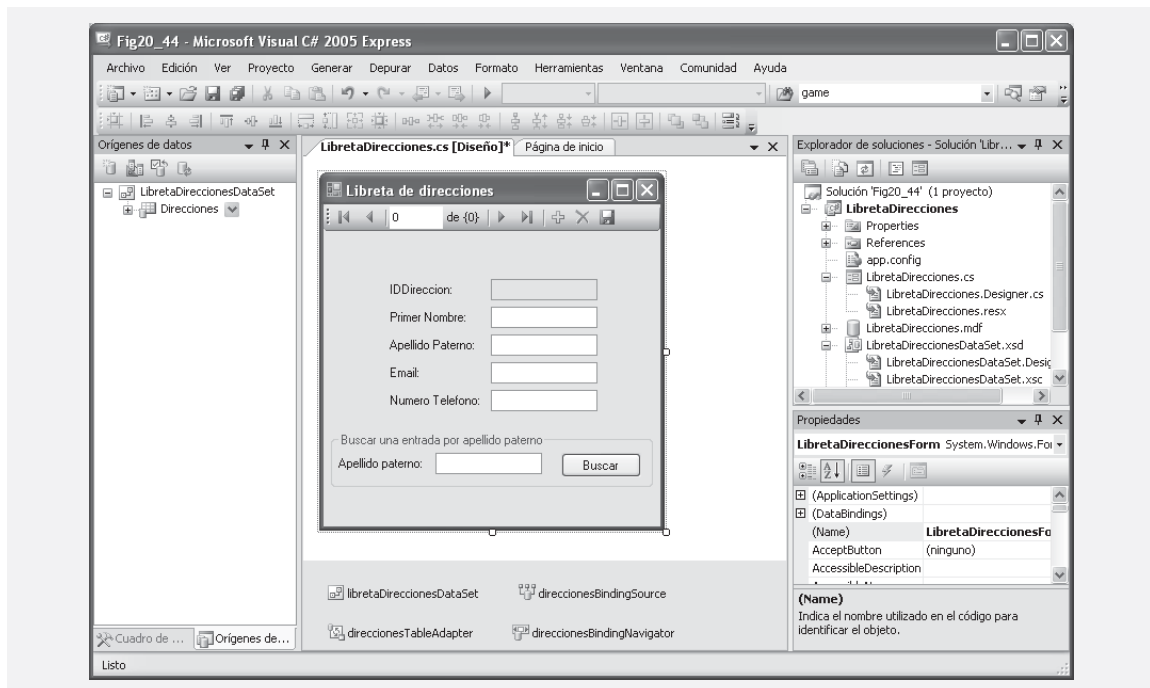


Figura 20.48 | Vista **Diseño** después de agregar controles para localizar un apellido paterno en la libreta de direcciones.

Paso 9: permitir que el usuario regrese a explorar todas las filas en la base de datos

Para permitir que los usuarios regresen a explorar todas las filas después de buscar filas específicas, agregue un control Button llamado `explorarTodoButton` debajo del control `buscarGroupBox`. Haga doble clic en `explorarTodoButton` para agregar un manejador del evento `Click` al código. Establezca la propiedad `Text` de `explorarTodoButton` a **Explorar todas las entradas** en la ventana **Propiedades**. Agregue una línea de código que llame a `direccionesTableAdapter.Fill( libretaDireccionesDataSet.Direcciones )` para volver a llenar el objeto `DireccionesDataTable` con todas las filas de la tabla en la base de datos (línea 50 de la figura 20.44). Agregue también una línea de código que borre la propiedad `Text` de `buscarTextBox` (línea 52). Inicie la aplicación. Busque un apellido paterno específico como en el paso anterior, y después haga clic en el botón `explorarTodoButton` para probar la nueva funcionalidad.

Enlace de datos en la aplicación LibretaDirecciones

El proceso de arrastrar y soltar el nodo `Direcciones` de la ventana **Orígenes de datos** en el formulario `LibretaDireccionesForm` de este ejemplo hizo que el IDE generara varios componentes en la bandeja de componentes. Éstos sirven para los mismos fines que los componentes generados para los ejemplos anteriores, en los que se utilizó la base de datos `Libros`. En este caso, `libretaDireccionesDataSet` es un objeto de un `DataSet` fuertemente tipificado, `LibretaDireccionesDataSet`, cuya estructura imita a la de la base de datos `LibretaDirecciones`. `direccionesBindingSource` es un objeto `BindingSource` que hace referencia a la tabla `Direcciones` del conjunto de datos `LibretaDireccionesDataSet`. `direccionesTableAdapter` encapsula a un objeto `SqlDataAdapter` configurado con objetos `SqlCommand` que ejecutan instrucciones de SQL en la base de datos `LibretaDirecciones`. Por último, `direccionesBindingNavigator` está enlazado al objeto `direccionesBindingSource`, con lo cual usted puede manipular en forma indirecta la tabla `Direcciones` del conjunto de datos `LibretaDireccionesDataSet`.

En cada uno de los ejemplos anteriores, en los que se utilizó un control `DataGridView` para mostrar todas las filas de una tabla de la base de datos, se estableció la propiedad `BindingSource` del control `DataGridView` al correspondiente objeto `BindingSource`. En este ejemplo, usted seleccionó **Detalles** de la lista desplegable para la tabla `Direcciones` en la ventana **Orígenes de datos**, por lo que los valores de una sola fila de la tabla aparecen en el formulario, en un conjunto de controles `TextBox`. El IDE configura el enlace de datos en este ejemplo, enlazando cada uno de los controles `TextBox` con una columna específica del objeto `DireccionesDataTable` en el conjunto de datos `LibretaDireccionesDataSet`. Para ello, el IDE establece la propiedad **DataBindings.Text** del control `TextBox`. Usted puede ver esta propiedad haciendo doble clic en el signo + que está enseguida de (**DataBindings**) en la ventana **Propiedades** (figura 20.49). Si hace clic en la lista desplegable para esta propiedad, podrá elegir un objeto `BindingSource` y una propiedad (es decir, columna) dentro del origen de datos asociado, para enlazarlos con el control `TextBox`.

Considere el control `TextBox` que muestra el valor `PrimerNombre`; el IDE lo nombró `primerNombreTextBox`. La propiedad `DataBindings.Text` de este control tiene asignada la propiedad `PrimerNombre` de `DireccionesBindingSource` (que hace referencia a `LibretaDireccionesDataSet.Direcciones`). Por ende, `primerNombreTextBox` siempre muestra el valor de la columna `PrimerNombre` en la fila actual seleccionada de `libretaDireccionesDataSet.Direcciones`. Cada control `TextBox` creado por el IDE en el formulario se configura de manera similar. Al explorar la libreta de direcciones con el objeto `DireccionesBindingNavigator`, cambia la posición actual en `libretaDireccionesDataSet.Direcciones` y, por consiguiente, cambian los valores que se muestran en cada control `TextBox`. Sin importar qué cambios se hagan al contenido de `libretaDireccionesDataSet.Addresses`, los controles `TextBox` permanecen enlazados a las mismas propiedades del objeto `DataTable` y siempre muestran los datos apropiados. Observe que los controles `TextBox` no muestran ningún valor si la versión en caché de `Direcciones` está vacía (es decir, si no hay filas en el objeto `DataTable`, debido a que la consulta que lo llenó no devolvió filas).

20.9 Uso de un objeto DataSet para leer y escribir XML

Una poderosa característica de ADO.NET es su habilidad para convertir en XML los datos almacenados en un origen de datos, para intercambiar datos entre aplicaciones, en un formato que sea portable. La clase `DataSet` del espacio de nombres `System.Data` proporciona los métodos **WriteXML**, **ReadXml** y **GetXml**, los cuales permiten a los desarrolladores crear documentos de XML a partir de orígenes de datos, y convertir datos de XML en orígenes de datos.

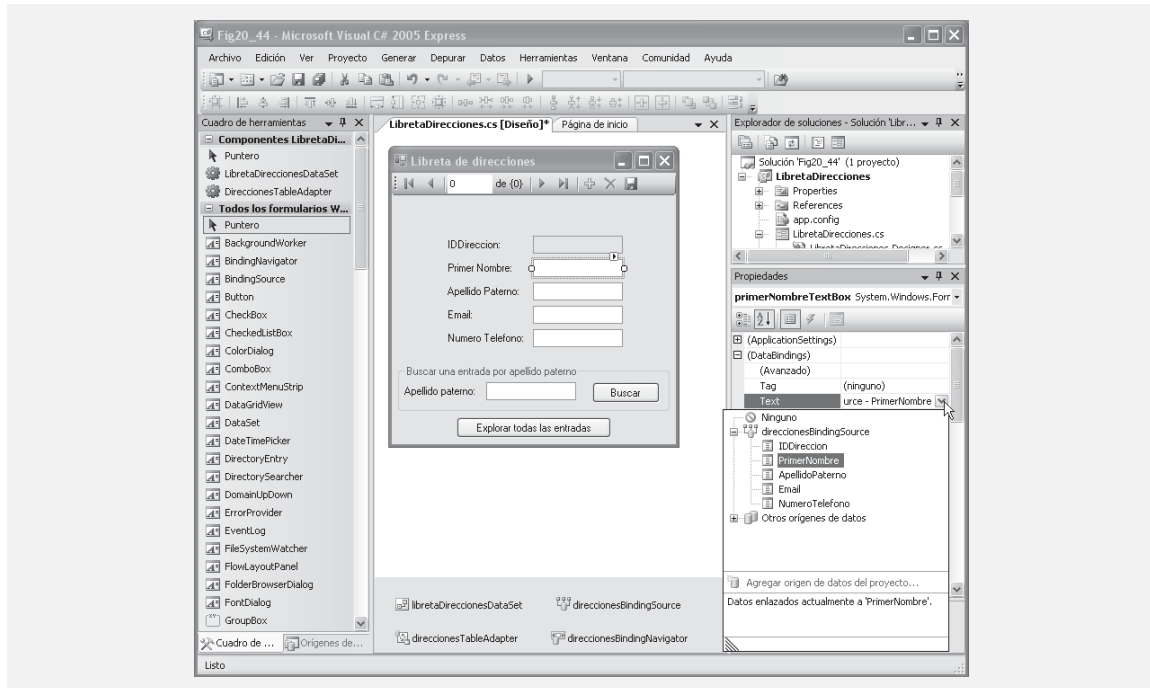


Figura 20.49 | Vista de la propiedad `DataBindings.Text` de un control `TextBox` en la ventana **Propiedades**.

Escribir datos de un origen de datos a un documento de XML

La aplicación de la figura 20.50 llena un objeto `DataSet` con estadísticas acerca de jugadores de béisbol, y después escribe los datos en un documento de XML. La aplicación también muestra el XML en un control `TextBox`.

Para crear esta GUI, primero agregamos la base de datos `Beisbol.mdf` (que se encuentra en el directorio de ejemplos de este capítulo) al proyecto, y después arrastramos el nodo **Jugadores** de la ventana **Orígenes de datos** hacia el formulario. Esta acción creó los controles `BindingNavigator` y `DataGridView` que aparecen en los resultados de la figura 20.50. Después agregamos el botón `escribirButton` y el cuadro de texto `salidaTextBox`. La representación en XML de la tabla `Jugadores` no debe editarse y abarca más líneas de las que el control `TextBox` puede mostrar a la vez, por lo que establecimos las propiedades `ReadOnly` y `MultiLine` del control `TextBox` a `true`, y su propiedad `ScrollBars` a `Vertical`. Para crear el manejador de eventos de `escribirButton`, haga doble clic sobre este control en la vista de **Diseño**.

El manejador de eventos `XMLWriterForm_Load` generado en forma automática (líneas 26-32) llama al método `Fill` de la clase `JugadoresTableAdapter` para llenar el conjunto de datos `beisbolDataSet` con datos provenientes de la tabla `Jugadores` en la base de datos `Béisbol`. Observe que el IDE enlaza el control `DataGridView` a `beisbolDataSet.Jugadores` (a través del objeto `JugadoresBindingSource`) para mostrar la información al usuario.

```

1  // Fig. 20.50: XMLWriter.cs
2  // Demuestra la generación de XML a partir de un DataSet de ADO.NET.
3  using System;
4  using System.Windows.Forms;
5
6  namespace XMLWriter
7  {
8      public partial class XMLWriterForm : Form
9      {

```

Figura 20.50 | Escribir la representación en XML de un objeto `DataSet` a un archivo. (Parte I de 2).

```

10  public XMLWriterForm()
11  {
12      InitializeComponent();
13  } // fin del constructor de XMLWriterForm
14
15  // el manejador del evento Click para el botón Guardar en el objeto
16  // BindingNavigator guarda los cambios realizados a los datos
17  private void jugadoresBindingNavigatorSaveItem_Click(
18      object sender, EventArgs e )
19  {
20      this.Validate();
21      this.jugadoresBindingSource.EndEdit();
22      this.jugadoresTableAdapter.Update( this.beisbolDataSet.Jugadores );
23  } // fin del método jugadoresBindingNavigatorSaveItem_Click
24
25  // carga los datos en la tabla beisbolDataSet.Jugadores
26  private void XMLWriterForm_Load( object sender, EventArgs e )
27  {
28      // TODO: esta línea de código carga datos en la
29      // tabla 'beisbolDataSet.Jugadores'.
30      // Puede moverla o quitarla según sea necesario.
31      this.jugadoresTableAdapter.Fill( this.beisbolDataSet.Jugadores );
32  }
33
34  // escribe la representación en XML de DataSet cuando se hace clic en el botón
35  private void escribirButton_Click( object sender, EventArgs e )
36  {
37      // establece el espacio de nombres para este objeto DataSet
38      // y el documento de XML resultante
39      beisbolDataSet.Namespace = "http://www.deitel.com/beisbol";
40
41      // escribe la representación en XML de DataSet a un archivo
42      beisbolDataSet.WriteXml( "Jugadores.xml" );
43
44      // muestra la representación en XML en un cuadro de texto
45      salidaTextBox.Text += "Se escribió el siguiente XML:\r\n" +
46          beisbolDataSet.GetXml() + "\r\n";
47  } // fin del método escribirButton_Click
48  } // fin de la clase XMLWriterForm
49  } // fin del espacio de nombres XMLWriter

```

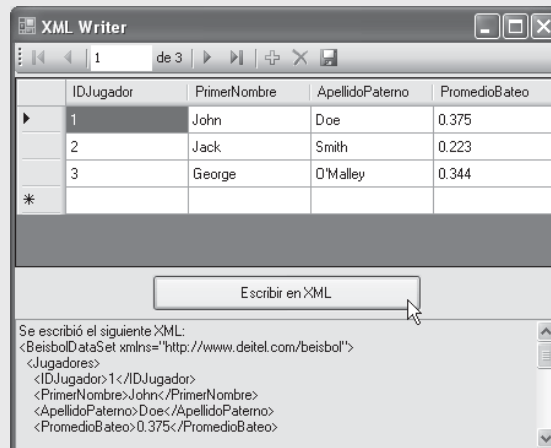


Figura 20.50 | Escribir la representación en XML de un objeto DataSet a un archivo. (Parte 2 de 2).

Las líneas 35-47 definen el manejador de eventos para el botón **Escribir a XML**. Cuando el usuario hace clic en este botón, la línea 39 establece la propiedad **Namespace** de `beisbolDataSet` para especificar un nombre de espacio para el objeto `DataSet` y cualquier documento de XML basado en ese objeto `DataSet` (en la sección 19.4 aprenderá acerca de los espacios de nombres de XML). La línea 42 invoca al método `WriteXml` de `DataSet`, el cual genera una representación en XML de los datos contenidos en el objeto `DataSet`, y después escribe el XML en el archivo especificado. Este archivo se crea en el directorio `bin/Debug` o `bin/Release` del proyecto, dependiendo de cómo haya ejecutado el programa. A continuación, las líneas 45-46 muestran esta representación en XML, que se obtiene mediante la invocación del método `GetXml` de `DataSet`, el cual devuelve un objeto `string` que contiene el XML.

Examinar un documento de XML generado por el método `WriteXml` de `DataSet`

La figura 20.51 presenta el documento `Jugadores.xml` generado por el método `WriteXml` de `DataSet` en la figura 20.50. Observe que el elemento raíz de `BeisbolDataSet` (línea 2) declara el espacio de nombres predeterminado del documento para que sea el espacio de nombres especificado en la línea 39 de la figura 20.50. Cada elemento de `Jugadores` representa un registro en la tabla `Jugadores`. Los elementos `IDJugador`, `PrimerNombre`, `ApellidoPaterno` y `PromedioBateo` corresponden a las columnas con esos nombres en la tabla `Jugadores` de la base de datos.

20.10 Conclusión

En este capítulo se introdujeron las bases de datos relacionales, SQL, ADO.NET y las herramientas de programación visual del IDE para trabajar con bases de datos. Usted analizó el contenido de una base de datos simple llamada `Libros`, y aprendió acerca de las relaciones entre las tablas de la base de datos. Después aprendió SQL básico para recuperar datos de, agregar nuevos datos a, y actualizar datos en, una base de datos.

Aprendió acerca de las clases de los espacios de nombres `System.Data` y `System.Data.SqlClient`, que permiten a los programas conectarse a una base de datos, para después acceder a esa base de datos y manipular sus datos. En este capítulo también se explicó el modelo sin conexión de ADO.NET, el cual permite a un programa almacenar de manera temporal en la memoria local los datos de una base de datos, como un objeto `DataSet`.

La segunda parte del capítulo se enfocó en el uso de las herramientas y asistentes del IDE para acceder a los orígenes de datos y manipularlos como una base de datos en las aplicaciones de GUI en C#. Usted aprendió a agregar orígenes de datos a los proyectos, y a utilizar las herramientas de “arrastrar y colocar” del IDE para mostrar

```

1  <?xml version="1.0" standalone="yes"?>
2  <BeisbolDataSet xmlns="http://www.deitel.com/beisbol">
3      <Jugadores>
4          <IDJugador>1</IDJugador>
5          <PrimerNombre>John</PrimerNombre>
6          <ApellidoPaterno>Doe</ApellidoPaterno>
7          <PromedioBateo>0.375</PromedioBateo>
8      </Jugadores>
9      <Jugadores>
10         <IDJugador>2</IDJugador>
11         <PrimerNombre>Jack</PrimerNombre>
12         <ApellidoPaterno>Smith</ApellidoPaterno>
13         <PromedioBateo>0.223</PromedioBateo>
14     </Jugadores>
15     <Jugadores>
16         <IDJugador>3</IDJugador>
17         <PrimerNombre>George</PrimerNombre>
18         <ApellidoPaterno>O'Malley</ApellidoPaterno>
19         <PromedioBateo>0.344</PromedioBateo>
20     </Jugadores>
21 </BeisbolDataSet>

```

Figura 20.51 | Documento de XML generado a partir de `BeisbolDataSet` en `XMLWriter`.

tablas de bases de datos en las aplicaciones. Le mostramos cómo el IDE oculta de usted el SQL que se utiliza para interactuar con la base de datos. También demostramos cómo agregar consultas personalizadas a las aplicaciones de GUI, de manera que se muestren sólo las filas de datos que cumplan con criterios específicos. Por último, aprendió a escribir datos de un origen de datos hacia un archivo XML.

En el siguiente capítulo, demostraremos cómo crear aplicaciones Web mediante el uso de la tecnología ASP.NET de Microsoft. También presentaremos el concepto de una aplicación de tres niveles, en donde una aplicación se divide en tres piezas que pueden residir en el mismo equipo, o pueden distribuirse entre equipos separados a través de una red como Internet. Como veremos, uno de estos niveles (el nivel de información) por lo general almacena datos en un RDBMS como SQL Server.

20.11 Recursos Web

www.microsoft.com/spain/sql/default.msp

Este sitio ofrece información en español acerca de SQL Server 2005, incluyendo tutoriales, artículos, noticias, actualizaciones y descargas.

www.microsoft.com/spanish/msdn/centro_recursos/sql2005/default.msp

El sitio del Centro de desarrollo de Microsoft SQL Server 2005 ofrece información en español acerca de SQL Server 2005 Express, incluyendo una descripción general, tutoriales y artículos.

msdn.microsoft.com/sql/

El Centro para desarrolladores de SQL Server ofrece información actualizada sobre productos, descargas, artículos y foros comunitarios.

lab.msdn.microsoft.com/express/sql

La página inicial para SQL Server 2005 Express ofrece artículos con ejemplos, blogs, grupos de noticias y demás recursos valiosos.

msdn2.microsoft.com/library/system.data.aspx

La documentación de Microsoft acerca del espacio de nombres System.Data.

msdn.microsoft.com/SQL/sqlreldata/TSQL/default.aspx

La guía de referencia del lenguaje SQL de Microsoft.

msdn2.microsoft.com/library/ms172013.aspx

La documentación de Microsoft para el **Generador de consultas** y otras herramientas visuales de bases de datos.

www.w3schools.com/sql/default.asp

El tutorial de SQL del W3C presenta las características básicas y avanzadas de SQL con ejemplos.

www.sql.org

El portal SQL ofrece vínculos a muchos recursos, incluyendo la sintaxis de SQL, tips, tutoriales, libros, revistas, grupos de discusión, compañías con servicios de SQL, consultores de SQL y software gratuito.

www.oracle.com/database/index.html

La página inicial para los sistemas de administración de bases de datos de Oracle.

www.sybase.com

La página inicial para el sistema de administración de bases de datos de Sybase.

www-306.ibm.com/software/data/db2/

La página inicial para el sistema de administración de bases de datos DB2 de IBM.

www.postgresql.org

La página inicial para el sistema de administración de bases de datos PostgreSQL.

www.mysql.com

La página inicial para el servidor de bases de datos MySQL.

21

ASP.NET 2.0, formularios Web Forms y controles Web

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Desarrollar aplicaciones Web mediante el uso de ASP.NET.
- Crear formularios Web Forms.
- Crear aplicaciones ASP.NET que consistan en varios formularios Web Forms.
- Mantener la información de estado acerca de un usuario, con rastreo de sesión y cookies.
- Usar la **Herramienta Administración de sitios Web** para modificar las opciones de configuración de una aplicación Web.
- Controlar el acceso de los usuarios a las aplicaciones Web, usando autenticación de formularios y controles de inicio de sesión de ASP.NET.
- Utilizar bases de datos en aplicaciones ASP.NET.
- Diseñar una página principal y páginas de contenido para crear un sitio Web con apariencia visual uniforme.

*Si cualquier hombre prepara
su caso y coloca su nombre
al pie de la primera página,
yo le daré una respuesta inmediata.
Si me obliga a dar vuelta a la hoja,
deberá esperar a mi conveniencia.*

—Lord Sandwich

*Regla uno: nuestro cliente
siempre tiene la razón.*

*Regla dos: si piensas que nuestro
cliente está mal, consulta
la Regla uno.*

—Anónimo

*Una pregunta justa debe ir
seguida de un acto en silencio.*

—Dante Alighieri

*Vendrás aquí y obtendrás
libros que abrirán tus
ojos, oídos, y tu
curiosidad; y sacarán tu
interior, o meterán tu exterior.*

—Ralph Waldo Emerson

- 21.1 Introducción
- 21.2 Transacciones HTTP simples
- 21.3 Arquitectura de aplicaciones multinivel
- 21.4 Creación y ejecución de un ejemplo de formulario Web Forms simple
 - 21.4.1 Análisis de un archivo ASPX
 - 21.4.2 Análisis de un archivo de código subyacente (code-behind)
 - 21.4.3 Relación entre un archivo ASPX y un archivo de código subyacente
 - 21.4.4 Cómo se ejecuta el código en una página Web ASP.NET
 - 21.4.5 Análisis del XHTML generado por una aplicación ASP.NET
 - 21.4.6 Creación de una aplicación Web ASP.NET
- 21.5 Controles Web
 - 21.5.1 Controles de texto y gráficos
 - 21.5.2 Control `AdRotator`
 - 21.5.3 Controles de validación
- 21.6 Rastreo de sesiones
 - 21.6.1 Cookies
 - 21.6.2 Rastreo de sesiones con `HttpSessionState`
- 21.7 Caso de estudio: conexión a una base de datos en ASP.NET
 - 21.7.1 Creación de un formulario Web Forms que muestra datos de una base de datos
 - 21.7.2 Modificación del archivo de código subyacente para la aplicación Libro de visitantes
- 21.8 Caso de estudio: aplicación segura de la base de datos Libros
 - 21.8.1 Análisis de la aplicación segura de la base de datos Libros completa
 - 21.8.2 Creación de la aplicación segura de la base de datos Libros
- 21.9 Conclusión
- 21.10 Recursos Web

21.1 Introducción

En capítulos anteriores, utilizamos formularios Windows Forms y controles de Windows para desarrollar aplicaciones para Windows. En este capítulo presentaremos el *desarrollo de aplicaciones Web* con la tecnología *ASP.NET 2.0* de Microsoft. Las aplicaciones basadas en Web crean contenido Web para los exploradores Web clientes. Este contenido Web incluye Lenguaje de marcado de hipertexto extensible (XHTML), secuencias de comandos del lado cliente, imágenes y datos binarios. Es conveniente que si usted no está familiarizado con XHTML lea primero el apéndice F, Introducción a XHTML: Parte 1, y el apéndice G, Introducción a XHTML: Parte 2, antes de estudiar este capítulo.

Presentaremos varios ejemplos que demuestran el desarrollo de aplicaciones Web mediante el uso de *formularios Web Forms, controles Web* (también llamados *controles de servidor ASP.NET*) y programación en C#. Los archivos de los formularios Web Forms tienen la extensión de archivo **.aspx** y contienen la GUI de la página Web. Usted personaliza los formularios Web Forms agregando controles Web, incluyendo etiquetas, cuadros de texto, imágenes, botones y demás componentes de la GUI. El archivo de formulario Web Forms representa la página Web que se envía al explorador cliente. De aquí en adelante, nos referiremos a los archivos de formularios Web Forms como *archivos ASPX*.

Todo archivo ASPX creado en Visual Studio tiene su correspondiente clase escrita en un lenguaje .NET, como C#. Esta clase contiene los manejadores de eventos, el código de inicialización, los métodos utilitarios y demás código de soporte. El archivo que contiene esta clase se llama *archivo de código subyacente (code-behind)*, y es el que proporciona la implementación programática del archivo ASPX.

Para desarrollar el código y las GUIs en este capítulo, utilizamos Microsoft Visual Web Developer 2005 Express: un IDE diseñado para desarrollar aplicaciones Web de ASP.NET. Visual Web Developer y Visual C# 2005

Express comparten muchas características y herramientas de programación visual comunes, que simplifican la creación de aplicaciones complejas, como las que acceden a una base de datos (que presentaremos en las secciones 21.7-21.8). La versión completa de Visual Studio 2005 incluye la funcionalidad de Visual Web Developer, por lo que las introducciones que presentaremos para Visual WEB Developer se aplican también para Visual Studio 2005. Tenga en cuenta que debe instalar Visual Web Developer 2005 Express (disponible en inglés: lab.msdn.microsoft.com/express/vwd/default.aspx, o en español: www.microsoft.com/spanish/msdn/vstudio/express/VWD/default.mspx) o una versión completa de Visual Studio 2005 para implementar los programas en este capítulo y en el capítulo 22, Servicios Web. El sitio www.deitel.com/books/csharpforprogrammers2 proporciona instrucciones para ejecutar los ejemplos de ASP.NET 2.0 que se presentan en este capítulo, en caso de que no desee recrearlos.

21.2 Transacciones HTTP simples

El desarrollo de aplicaciones Web requiere una comprensión básica de las redes y World Wide Web. En esta sección veremos el *Protocolo de transferencia de hipertexto (HTTP)* y lo que ocurre detrás de las cámaras cuando un explorador visualiza una página Web. HTTP especifica un conjunto de *métodos* y *encabezados* que permiten a los clientes y servidores interactuar e intercambiar información, de una manera uniforme y predecible.

En su forma más simple, una página Web no es más que un documento XHTML: un archivo de texto simple que contiene *marcado* (es decir, *etiquetas*) para describir a un explorador Web cómo visualizar y dar formato a la información del documento. Por ejemplo, el siguiente marcado de XHTML:

```
<title>Mi página Web</title>
```

indica que el explorador debe mostrar el texto entre la *etiqueta inicial* `<title>` y la *etiqueta final* `</title>` en la barra de título. Los documentos de XHTML también pueden contener datos de *hipertexto* (que por lo general se conocen como *hipervínculos*) con vínculos hacia distintas páginas, o a otras partes de la misma página. Cuando el usuario activa un hipervínculo (por lo general, haciendo clic sobre él con el ratón), la página Web solicitada se carga en la ventana del explorador del usuario.

Cualquier documento de XHTML disponible para verlo a través de la Web tiene su correspondiente Localizador de recursos uniforme (URL). Un URL es una dirección que indica la ubicación de un recurso de Internet, como un documento de XHTML. El URL contiene información que dirige a un explorador al recurso que el usuario desea acceder. Las computadoras que ejecutan software de *servidor Web* hacen que dichos recursos estén disponibles. Cuando se realizan peticiones de aplicaciones Web ASP.NET, por lo general el servidor Web es Microsoft *Internet Information Services (IIS)*. Como veremos en breve, también es posible probar aplicaciones ASP.NET mediante el uso del Servidor de desarrollo ASP.NET integrado en Visual Web Developer.

Examinaremos los componentes del URL

```
http://www.deitel.com/books/downloads.html
```

El `http://` indica que el recurso se debe obtener utilizando el protocolo HTTP. La porción intermedia, `www.deitel.com`, es el *nombre de host* completamente calificado del servidor: el nombre de la computadora en la que reside el recurso. Por lo general, a esta computadora se le denomina *host*, ya que aloja los recursos y los mantiene. El nombre de host `www.deitel.com` se traduce en una *dirección IP* (68.236.123.125), que identifica al servidor en forma similar al número telefónico que identifica en forma única a una línea telefónica específica. El nombre de host se traduce en una dirección IP mediante un *servidor del sistema de nombre de dominios (DNS)*: una computadora que mantiene una base de datos de nombres de hosts y sus correspondientes direcciones IP. Esta operación de traducción se conoce como *búsqueda en el DNS*.

El resto del URL (es decir, `/books/downloads.html`) especifica el nombre del recurso solicitado (el documento de XHTML `downloads.html`) y su ruta, o ubicación (`/books`) en el servidor Web. La ruta podría especificar la ubicación de un directorio real en el sistema de archivos del servidor Web. No obstante, por razones de seguridad la ruta a menudo especifica la ubicación de un *directorio virtual*. En dichos sistemas, el servidor traduce el directorio virtual en una ubicación real en el servidor (o en otra computadora en la red del servidor), con lo cual se oculta la verdadera ubicación del recurso. Algunos recursos se crean en forma dinámica y, por lo tanto, no residen en ninguna parte del equipo servidor. El nombre de host en el URL para un recurso de este tipo especifica el servidor correcto, y la información de la ruta y del recurso identifican la ubicación del recurso con el cual se responderá a la solicitud del cliente.

Cuando el explorador Web recibe un URL, realiza una transacción HTTP simple para recuperar y visualizar la página Web que se encuentra en esa dirección. La figura 21.1 ilustra la transacción con detalle. Esta transacción consiste en la interacción entre el explorador Web (el lado cliente) y la aplicación del servidor Web (el lado servidor).

En la figura 21.1, el explorador Web envía una solicitud HTTP al servidor. La solicitud (en su forma más simple) es:

```
GET /books/downloads.html HTTP/1.1
```

La palabra **GET** es un método de HTTP, el cual indica que el cliente desea obtener un recurso del servidor. El resto de la solicitud proporciona el nombre de la ruta del recurso (un documento de XHTML), junto con el nombre del protocolo y el número de versión (HTTP/1.1).

Cualquier servidor que entienda HTTP (versión 1.1) puede traducir esta solicitud y responder en forma apropiada. La figura 21.2 ilustra el resultado de una solicitud exitosa. El servidor primero responde enviando una línea de texto que indica la versión de HTTP, seguida de un código numérico y una frase que describe el estado de la transacción. Por ejemplo,

```
HTTP/1.1 200 OK
```

indica éxito, mientras que

```
HTTP/1.1 404 Not found
```

informa al cliente que el servidor Web no pudo localizar el recurso solicitado.

Después, el servidor envía uno o más **encabezados HTTP**, que proporcionan información adicional acerca de los datos que se van a enviar. En este caso, el servidor envía un documento de texto de XHTML, por lo que el encabezado HTTP para este ejemplo sería:

```
Content-type: text/html
```

La información que se proporciona en este encabezado especifica el tipo de *Extensiones multipropósito de correo Internet (MIME)* del contenido que el servidor transmitirá al explorador. MIME es un estándar de Internet que especifica formatos de datos, de manera que los programas puedan interpretar los datos en forma correcta. Por ejemplo, el tipo MIME `text/plain` indica que la información que se envía es texto que puede visualizarse de manera directa, sin necesidad de interpretar el contenido como marcado de XHTML. De manera similar, el tipo MIME `image/jpeg` indica que el contenido es una imagen JPEG. Cuando el explorador recibe este tipo MIME, trata de visualizar la imagen.

El encabezado, o conjunto de encabezados, va seguido de una línea en blanco, la cual indica al cliente que el servidor terminó de enviar encabezados HTTP. Después, el servidor envía el contenido del documento de XHTML solicitado (`downloads.html`). El servidor termina la conexión cuando se completa la transferencia

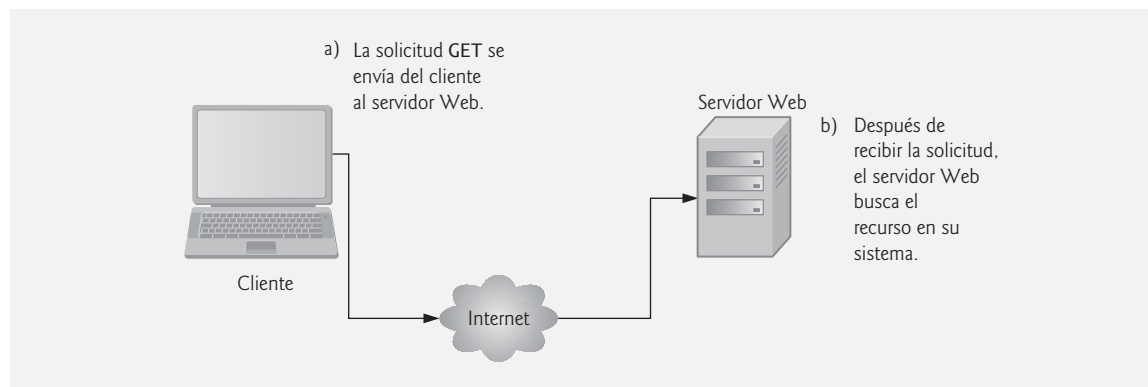


Figura 21.1 | Un cliente interactuando con el servidor Web. Paso 1: la solicitud GET.

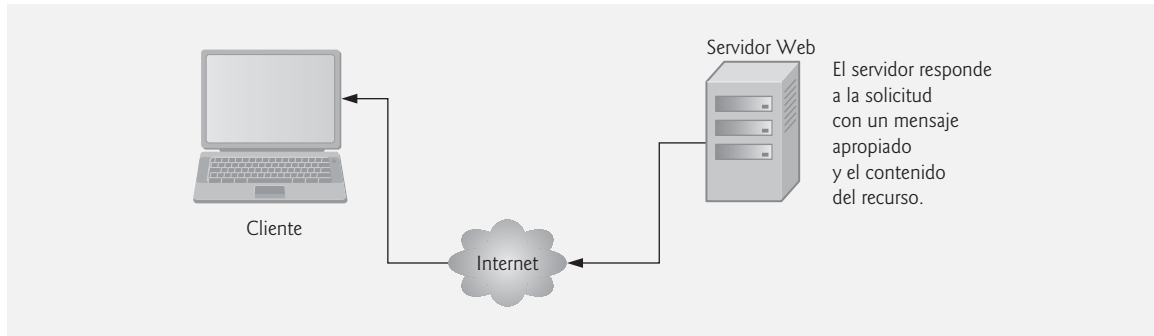


Figura 21.2 | Un cliente interactuando con el servidor Web. Paso 2: la respuesta HTTP.

del recurso. En este punto, el explorador del lado cliente analiza el marcado de XHTML que recibió y *representa* (o visualiza) los resultados.

21.3 Arquitectura de aplicaciones multinivel

Las aplicaciones basadas en Web son *aplicaciones multinivel* (a las que algunas veces se les conoce como *aplicaciones de n niveles*). Las aplicaciones multinivel dividen su funcionalidad en *niveles* separados (es decir, agrupamientos lógicos de funcionalidad). Aunque los niveles pueden ubicarse en la misma computadora, por lo general los niveles de las aplicaciones basadas en Web residen en computadoras separadas. La figura 21.3 presenta la estructura básica de una aplicación basada en Web de tres niveles.

El *nivel de información* (también conocido como *nivel de datos* o *nivel inferior*) mantiene los datos que pertenecen a la aplicación. Por lo común, este nivel almacena los datos en un sistema de administración de bases de datos relacionales (RDBMS). En el capítulo 20 hablamos sobre los sistemas RDBMS. Por ejemplo, una tienda de ventas al detalle podría tener una base de datos para almacenar la información de sus productos, como las descripciones, los precios y las cantidades en existencia. La misma base de datos también podría contener información sobre los clientes, como los nombres de usuario, las direcciones de facturación y los números de tarjetas de crédito. Este nivel puede contener varias bases de datos, que en conjunto conforman los datos necesarios para nuestra aplicación.

El *nivel intermedio* implementa la *lógica comercial*, la *lógica de control* y la *lógica de presentación*, para controlar las interacciones entre los clientes de la aplicación y los datos de la misma. El nivel intermedio actúa como un intermediario entre los datos en el nivel de información y los clientes de la aplicación. La lógica de control del nivel intermedio procesa las peticiones de los clientes (como las peticiones para ver un catálogo de productos) y recupera datos de la base de datos. Después, la lógica de presentación del nivel intermedio procesa los datos del nivel de información y presenta el contenido al cliente. Por lo general, las aplicaciones Web presentan datos a los clientes en forma de documentos de XHTML.

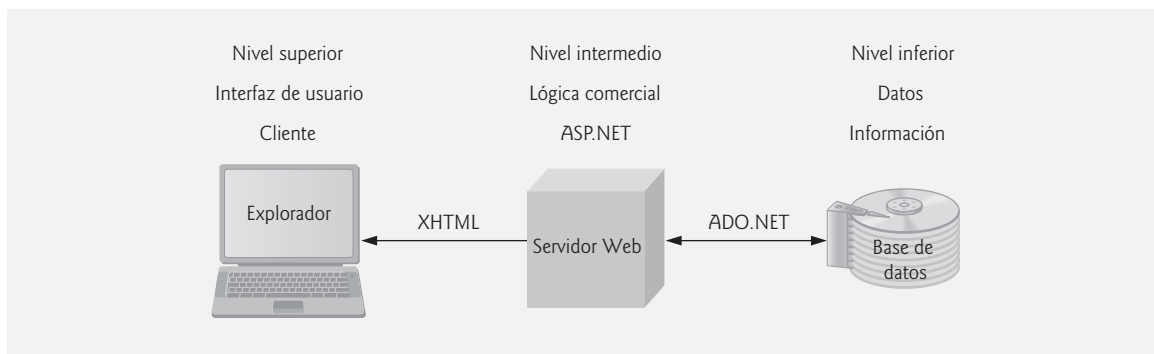


Figura 21.3 | Arquitectura de tres niveles.

La lógica comercial en el nivel intermedio hace valer las *reglas comerciales* y asegura que los datos sean confiables antes de que la aplicación servidor actualice la base de datos o presente los datos a los usuarios. Las reglas comerciales dictan la forma en que los clientes pueden o no tener acceso a los datos de la aplicación, y cómo las aplicaciones procesan los datos. Por ejemplo, una regla comercial en el nivel intermedio de la aplicación basada en Web de una tienda de ventas al detalle podría asegurar que las cantidades de todos los productos permanezcan siempre positivas. La lógica comercial del nivel intermedio rechazaría la solicitud de un cliente de establecer una cantidad negativa en la base de datos de información de productos del nivel inferior.

El *nivel cliente*, o *nivel superior*, es la interfaz de usuario de la aplicación, la cual recopila los datos de entrada y visualiza los resultados. Los usuarios interactúan en forma directa con la aplicación a través de la interfaz de usuario, que por lo general es un explorador Web, un teclado y un ratón. En respuesta a las acciones del usuario (por ejemplo, hacer clic en un hipervínculo), el nivel cliente interactúa con el nivel intermedio para realizar peticiones y para recuperar datos del nivel de información. Después, el nivel cliente muestra al usuario los datos recuperados del nivel intermedio. El nivel cliente nunca interactúa en forma directa con el nivel de información.

21.4 Creación y ejecución de un ejemplo de formulario Web Forms simple

Nuestro primer ejemplo muestra la hora del día del servidor Web en una ventana del explorador Web. Al ejecutarse, este programa muestra el texto Un ejemplo simple de formularios Web Forms, seguido de la hora del servidor Web. Como dijimos antes, el programa consiste en dos archivos relacionados: un archivo ASPX (figura 21.4) y un archivo de código subyacente en C# (figura 21.5). Primero le mostraremos el marcado, el código y los resultados; después lo guiaremos cuidadosamente a través del proceso paso a paso para crear este programa. [Nota: el marcado en la figura 21.4 y en los demás listados de archivos ASPX en este capítulo es el mismo que el marcado que aparece en Visual Web Developer, pero nosotros cambiamos el formato del marcado para fines de presentación y para mejorar la legibilidad del código.]

Visual Web Developer genera todo el marcado que se muestra en la figura 21.4 cuando usted establece el título de la página Web, escribe texto en el formulario Web Forms, arrastra un control Label hacia el formulario Web y establece las propiedades del texto de la página y del control Label. En breve le mostraremos estos pasos.

```

1  <!-- Fig. 21.4: HoraWeb.aspx -->
2  <!-- Una página que muestra la hora actual en un control Label -->
3  <%@ Page Language="C#" AutoEventWireup="true" CodeFile="HoraWeb.aspx.cs"
4     Inherits="HoraWeb" EnableSessionState="False" %>
5
6  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
7     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
8
9  <html xmlns="http://www.w3.org/1999/xhtml">
10     <head runat="server">
11         <title>Un ejemplo simple de un formulario Web Forms</title>
12     </head>
13     <body>
14         <form id="form1" runat="server">
15             <div>
16                 <h2>Hora actual en el servidor Web:</h2>
17                 <p>
18                     <asp:Label ID="horaLabel" runat="server" BackColor="Black"
19                         Font-Size="XX-Large" ForeColor="Yellow"
20                         EnableViewState="False"></asp:Label>
21                 </p>
22             </div>
23         </form>
24     </body>
25 </html>

```

Figura 21.4 | Archivo ASPX que muestra la hora del servidor Web.

21.4.1 Análisis de un archivo ASPX

El archivo ASPX contiene otra información además de XHTML. Las líneas 1-2 son *comentarios de ASP.NET* que indican el número de figura, el nombre del archivo y el propósito del mismo. Los comentarios de ASP.NET empiezan con `<!--` y terminan con `-->`. Nosotros agregamos estos comentarios al archivo. Las líneas 3-4 utilizan una directiva **Page** (en un archivo ASPX, una *directiva* se delimita con `<%@ y %>`) para especificar información que ASP.NET necesita para procesar este archivo. El atributo **Language** de la directiva Page especifica el lenguaje del archivo de código subyacente (code-behind) como C#; el archivo de código subyacente (es decir, **CodeFile**) es `HoraWeb.aspx.cs`. Observe que el nombre de un archivo de código subyacente por lo general consiste en el nombre completo del archivo ASPX (por ejemplo, `HoraWeb.aspx`), seguido de la extensión `.cs`.

El atributo **AutoEventWireup** (línea 3) determina cómo se manejan los eventos de los formularios Web Forms. Cuando **AutoEventWireup** se establece en `true`, ASP.NET determina cuáles métodos de la clase se van a llamar, en respuesta a un evento generado por la directiva Page. Por ejemplo, ASP.NET llama a los métodos `Page_Load` y `Page_Init` en el archivo de código subyacente para manejar los eventos `Load` e `Init` de Page, respectivamente. (Más adelante en el capítulo hablaremos sobre estos eventos.)

El atributo **Inherits** (línea 4) especifica la clase en el archivo de código subyacente de la que esta clase ASP.NET hereda; en este caso, `HoraWeb`. En unos momentos hablaremos más sobre **Inherits**. [Nota: establecimos de manera explícita el atributo **EnableSessionState** (línea 4) a `False`. Más adelante en el capítulo explicaremos el significado de este atributo. Algunas veces el IDE genera valores para los atributos (por ejemplo, `True` y `False`) y nombres para los controles (como verá más adelante en el capítulo) que no se adhieren a nuestras convenciones estándar de capitalización de código (es decir, `true` y `false`). No obstante, y a diferencia del código de C#, el marcado de ASP.NET no es sensible al uso de mayúsculas o minúsculas, por lo que no hay ningún problema si utilizamos de manera distinta las mayúsculas y las minúsculas. Para permanecer consistentes con el código generado por el IDE, no modificamos estos valores en nuestros listados de código ni en nuestras discusiones complementarias.]

Para este primer archivo ASPX, proporcionamos una breve discusión acerca del marcado de XHTML. No hablaremos sobre la mayoría del XHTML contenido en los siguientes archivos ASPX. Las líneas 6-7 contienen la declaración de tipo de documento, que especifica el nombre del elemento de documento (HTML) y el Identificador de recursos uniforme (URI) PUBLIC para la DTD que define el vocabulario XHTML.

Las líneas 9-10 contienen las etiquetas iniciales `<html>` y `<head>`, respectivamente. Los documentos de XHTML tienen el elemento raíz `html` y la información de marcado sobre el documento en el elemento `head`. Observe además que el elemento `html` especifica el espacio de nombres XML del documento, usando el atributo `xmlns` (vea la sección 19.4).

La línea 11 establece el título de esta página Web. En breve demostraremos cómo establecer el título a través de una propiedad en el IDE. Observe el atributo **runat** en la línea 10, que se estableció a `"server"`. Este atributo indica que, cuando un cliente solicita este archivo ASPX, ASP.NET procesa el elemento `head` y sus elementos anidados en el servidor, y genera el XHTML correspondiente, el cual se envía posteriormente al cliente. En este caso, el XHTML que se envía al cliente será idéntico al marcado en el archivo ASPX. Sin embargo, y como veremos pronto, ASP.NET puede generar marcado de XHTML complejo a partir de elementos simples en un archivo ASPX.

La línea 13 contiene la etiqueta inicial `<body>`, la cual empieza el cuerpo del documento de XHTML; el cuerpo tiene el contenido principal que el explorador visualiza. El formulario que contiene nuestro texto XHTML y los controles se define en las líneas 14-23. De nuevo, el atributo **runat** en el elemento `form` indica que este elemento se ejecuta en el servidor, el cual genera el XHTML equivalente y lo envía al cliente. Las líneas 15-22 contienen un elemento `div`, que agrupa los elementos del formulario en un bloque de marcado.

La línea 16 es un elemento de encabezado `h2`, el cual contiene texto que indica el propósito de la página Web. Como demostraremos en breve, el IDE genera este elemento en respuesta a la acción de escribir texto directamente en el formulario Web Forms y seleccionar el texto como encabezado de segundo nivel.

Las líneas 17-21 contienen un elemento `p` para marcar el contenido y que se visualice como un párrafo en el explorador. Las líneas 18-20 marcan un control Web tipo etiqueta. Las propiedades que establecemos en la ventana **Propiedades**, como `Font-Size` y `BackColor` (es decir, el color de fondo), son atributos aquí. El atributo **ID** (línea 18) asigna un nombre al control, de manera que pueda manipularse mediante programación en el archivo de código subyacente. Establecimos el atributo **EnableViewState** del control (línea 20) a `False`. Más adelante en el capítulo explicaremos el significado de este atributo.

El *prefijo de etiqueta asp:* en la declaración de la etiqueta **Label** (línea 18) indica que la etiqueta es un control Web de ASP.NET, no un elemento de XHTML. Cada control Web se asigna a un elemento correspondiente de XHTML (o grupo de elementos); cuando procesa un control Web en el servidor, ASP.NET genera marcado de XHTML que se enviará al cliente para representar a ese control en un explorador Web.



Tip de portabilidad 21.1

El mismo control Web puede asignarse a distintos elementos de XHTML, dependiendo del explorador cliente y de las configuraciones de las propiedades del control Web.

En este ejemplo, el control `asp:Label` se asigna al elemento **span** de XHTML (es decir, ASP.NET crea un elemento `span` para representar a este control en el explorador Web del cliente). Un elemento `span` contiene texto que se visualiza en una página Web. Este elemento específico se utiliza debido a que los elementos `span` permiten aplicar estilos de formato al texto. Varios de los valores de las propiedades que se aplicaron a nuestra etiqueta se representan como parte del atributo `style` del elemento `span`. Pronto verá cuál es la apariencia del marcado del elemento `span` generado.

El control Web en este ejemplo contiene el par atributo-valor `runat="server"` (línea 18), ya que este control debe procesarse en el servidor, de manera que éste pueda traducir el control en XHTML que pueda representarse en el explorador cliente. Si este par de atributos no está presente, el elemento `asp:Label` se escribe como texto en el cliente (es decir, el control no se convierte en un elemento `span` y no se representa en forma apropiada).

21.4.2 Análisis de un archivo de código subyacente (code-behind)

La figura 21.5 presenta el archivo de código subyacente. Recuerde que el archivo ASPX en la figura 21.4 hace referencia a este archivo en la línea 3.

La línea 13 empieza la declaración de la clase `WebTime`. En el capítulo 9 vimos que la declaración de una clase puede abarcar varios archivos de código fuente, y que las porciones separadas de la declaración de la clase en cada archivo se conocen como clases parciales. El modificador `partial` en la línea 13 de la figura 21.5 indica que el archivo de código subyacente en realidad es una clase parcial. En breve hablaremos sobre el resto de esta clase.

La línea 13 indica que `HoraWeb` hereda de la clase **Page** en el espacio de nombres **System.Web.UI**. Este espacio de nombres contiene clases y controles que ayudan a crear aplicaciones basadas en Web. La clase `Page` proporciona los manejadores de eventos y los objetos necesarios para crear aplicaciones basadas en Web. Además de la clase `Page` (de la que todas las aplicaciones Web heredan, ya sea en forma directa o indirecta), `System.Web.UI` incluye también la clase **Control**: la clase base que proporciona una funcionalidad común para todos los controles Web.

```

1 // Fig. 21.5: HoraWeb.aspx.cs
2 // Archivo de código subyacente para una página que muestra la hora actual.
3 using System;
4 using System.Data;
5 using System.Configuration;
6 using System.Web;
7 using System.Web.Security;
8 using System.Web.UI;
9 using System.Web.UI.WebControls;
10 using System.Web.UI.WebControls.WebParts;
11 using System.Web.UI.HtmlControls;
12
13 public partial class HoraWeb : System.Web.UI.Page
14 {
15     // Inicializa el contenido de la página
16     protected void Page_Init( object sender, EventArgs e )
17     {

```

Figura 21.5 | Archivo de código subyacente para una página que muestra la hora del servidor Web. (Parte 1 de 2).

```

18      // muestra la hora actual del servidor en horaLabel
19      horaLabel.Text = string.Format( "{0:D2}:{1:D2}:{2:D2}",
20          DateTime.Now.Hour, DateTime.Now.Minute, DateTime.Now.Second );
21  } // fin del método Page_Init
22  } // fin de la clase HoraWeb

```



Figura 21.5 | Archivo de código subyacente para una página que muestra la hora del servidor Web. (Parte 2 de 2).

Las líneas 16-21 definen el método **Page_Init**, el cual maneja el evento **Init** de la página. Este evento (el primero que se genera después de que se solicita una página) indica que la página está lista para inicializarse. La única inicialización requerida para esta página es establecer la propiedad **Text** de **horaLabel** con la hora del servidor (es decir, la computadora en la que se ejecuta este código). La instrucción en las líneas 19-20 recupera la hora actual y le aplica el formato **HH:MM:SS**. Por ejemplo, a las 9 AM se les aplica el formato 09:00:00 y a las 2:30 PM se les aplica el formato 14:30:00. Observe que el archivo de código subyacente puede acceder a **horaLabel** (el ID del control **Label** en el archivo ASPX) mediante programación, aun cuando el archivo no contiene una declaración para una variable llamada **horaLabel**. En unos momentos veremos por qué.

21.4.3 Relación entre un archivo ASPX y un archivo de código subyacente

¿Cómo se utilizan los archivos ASPX y de código subyacente para crear la página Web que se envía al cliente? En primer lugar, recuerde que la clase **HoraWeb** es la clase base especificada en la línea 3 del archivo ASPX (figura 21.4). Esta clase (declarada parcialmente en el archivo de código subyacente) hereda de **Page**, la cual define la funcionalidad general de una página Web. La clase parcial **HoraWeb** hereda esta funcionalidad y define cierta funcionalidad propia (es decir, mostrar la hora actual). El archivo de código subyacente contiene el código para mostrar la hora, mientras que el archivo ASPX contiene el código para definir la GUI.

Cuando un cliente solicita un archivo ASPX, ASP.NET crea dos clases detrás de las cámaras. Recuerde que el archivo de código subyacente contiene una clase parcial llamada **HoraWeb**. El primer archivo que ASP.NET genera es otra clase parcial, la cual contiene el resto de la clase **HoraWeb**, con base en el marcado que contiene el archivo ASPX. Por ejemplo, **HoraWeb.aspx** contiene un control **Web Label** con un ID de **horaLabel**, por lo que la clase parcial generada contendría una declaración para una variable **Label** llamada **horaLabel**. Esta clase parcial podría verse así:

```

public partial class HoraWeb
{
    protected System.Web.UI.WebControls.Label horaLabel;
}

```

Observe que **Label** es un control Web definido en el espacio de nombres **System.Web.UI.WebControls**, el cual contiene controles Web para diseñar la interfaz de usuario de una página. Los controles Web en este espacio de nombres se derivan de la clase **WebControl**. Al compilarse, la declaración de la clase parcial anterior que contiene declaraciones de controles Web se combina con la declaración de la clase parcial del archivo de código subyacente, para formar la clase **HoraWeb** completa. Esto explica por qué la línea 19 en el método **Page_Init** de **HoraWeb.aspx.cs** (figura 21.5) puede acceder a **horaLabel**, la cual se crea en las líneas 18-20 de **HoraWeb.aspx** (figura 21.4); el método **Page_Init** y el control **horaLabel** son en realidad miembros de la misma clase, pero están definidos en clases parciales separadas.

La segunda clase generada por ASP.NET se basa en el archivo ASPX que define la representación visual de la página. Esta nueva clase hereda de la clase `Horaweb`, la cual define la lógica de la página. La primera vez que se solicita la página Web, esta clase se compila y se crea una instancia. Esta instancia representa a nuestra página; crea el XHTML que se envía al cliente. El ensamblado creado a partir de nuestras clases compiladas se coloca dentro de un subdirectorio de

```
C:\Windows\Microsoft.NET\Framework\NúmeroVersión\
Temporary ASP.NET Files\Horaweb
```

en donde *NúmeroVersión* es el número de versión del .NET Framework (por ejemplo, v2.0.50215) que está instalado en su computadora.



Tip de rendimiento 21.1

Una vez que se ha creado una instancia de la página Web, varios clientes pueden usarla para acceder a la página; no es necesario volver a compilar. El proyecto se volverá a compilar sólo cuando usted modifique la aplicación; el entorno en tiempo de ejecución detecta los cambios y el proyecto se recompila para reflejar el contenido alterado.

21.4.4 Cómo se ejecuta el código en una página Web ASP.NET

Veamos brevemente cómo se ejecuta el código para nuestra página Web. Cuando se crea una instancia de la página, el evento `Init` ocurre primero, invocando al método `Page_Init`. Este método puede contener el código necesario para inicializar objetos y otros aspectos de la página. Una vez que se ejecuta `Page_Init`, ocurre el evento **Load** y se ejecuta el manejador de eventos **Page_Load**. Aunque no está presente en este ejemplo, este evento se hereda de la clase `Page`. Más adelante en el capítulo verá ejemplos del manejador de eventos `Page_Load`. Cuando este manejador de eventos termina su ejecución, la página procesa eventos que generan los controles de la página, como las interacciones del usuario con la GUI. Cuando el objeto Web Forms está listo para la recolección de basura se produce un evento **Unload**, el cual llama al manejador de eventos **Page_Unload**. Este evento también se hereda de la clase `Page`. Por lo general, `Page_Unload` contiene código que libera los recursos utilizados por la página.

21.4.5 Análisis del XHTML generado por una aplicación ASP.NET

La figura 21.6 muestra el XHTML generado por ASP.NET cuando un explorador Web cliente solicita el archivo `Horaweb.aspx` (figura 21.4). Para ver este XHTML, seleccione **Ver > Código fuente** en Internet Explorer. [Nota: agregamos los comentarios de XHTML en las líneas 1-2 y cambiamos el formato de XHTML para que se conforme a nuestras convenciones de codificación.]

El contenido de esta página es similar al del archivo ASPX. Las líneas 7-9 definen un encabezado de documento comparable al de la figura 21.4. Las líneas 10-28 definen el cuerpo del documento. La línea 11 empieza el formulario, un mecanismo para recolectar la información del usuario y enviarlo al servidor Web. En este programa específico, el usuario no envía datos al servidor Web para que los procese; sin embargo, el procesamiento de los datos del usuario es una parte crucial de muchas aplicaciones, y el formulario lo facilita. En ejemplos posteriores le demostraremos cómo enviar datos al servidor.

Los formularios de XHTML pueden contener componentes visuales y no visuales. Los componentes visuales incluyen botones en los que se puede hacer clic, y otros componentes de la GUI con los que interactúan los usuarios. Los componentes no visuales, llamados *entradas ocultas*, almacenan datos tales como direcciones de correo

```
1 <!-- Fig. 21.6: Horaweb.aspx -->
2 <!-- El XHTML que se genera cuando se carga Horaweb.aspx -->
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 Transitional//EN"
4   "http://www.w3.org/TR/xhtml11/DTD/xhtml11-transitional.dtd">
5
6 <html xmlns="http://www.w3.org/1999/xhtml">
7   <head>
8     <title>Un ejemplo simple de un formulario Web Forms</title>
```

Figura 21.6 | Respuesta de XHTML cuando el explorador solicita `Horaweb.aspx`. (Parte 1 de 2).

```

 9      </head>
10      <body>
11          <form method="post" action="HoraWeb.aspx" id="form1">
12              <div>
13                  <input type="hidden" name="__VIEWSTATE"
14                      id="__VIEWSTATE" value=
15                      "/wEPDwUJODExMDE5NzY5ZGQjRXwQI1p2TGKLC6eh/k3WzxKkw==" />
16              </div>
17
18              <div>
19                  <h2>Hora actual en el servidor Web:</h2>
20                  <p>
21                      <span id="horaLabel1" style="color:Yellow;
22                          background-color:Black;font-size:XX-Large;">
23                          21:23:28
24                      </span>
25                  </p>
26              </div>
27          </form>
28      </body>
29  </html>

```

Figura 21.6 | Respuesta de XHTML cuando el explorador solicita HoraWeb.aspx. (Parte 2 de 2).

electrónico, que el autor del documento especifica. Una de estas entradas ocultas se define en las líneas 13-15. Más adelante en el capítulo, hablaremos sobre el significado preciso de esta entrada oculta. El atributo **method** del elemento **form** (línea 11) especifica el método mediante el cual el explorador Web envía el formulario al servidor. El atributo **action** identifica el nombre y la ubicación del recurso que se solicitará cuando se envíe este formulario; en este caso, HoraWeb.aspx. Recuerde que el elemento **form** del archivo ASPX contiene el par atributo-valor **runat="server"** (línea 14 de la figura 21.4). Cuando el elemento **form** se procesa en el servidor, se elimina el atributo **runat**. Se agregan los atributos **method** y **action**, y el elemento **form** de XHTML resultante se envía al explorador cliente.

En el archivo ASPX, el control **Label** del formulario (es decir, horaLabel1) es un control Web. Aquí estamos viendo el XHTML creado por nuestra aplicación, por lo que el elemento **form** contiene un elemento **span** (líneas 21-24 de la figura 21.6) para representar el texto en la etiqueta. En este caso específico, ASP.NET asigna el control Web **Label** a un elemento **span** de XHTML. Las opciones de formato que se especificaron como propiedades de horaLabel1, tales como el tamaño de fuente y el color del texto en el control **Label**, ahora se especifican en el atributo **style** del elemento **span**.

Observe que sólo los elementos en el archivo ASPX que están marcados con el par atributo-valor **runat="server"**, o que se especifican como controles Web, se modifican o sustituyen cuando el servidor procesa el archivo. Los elementos de XHTML puro, como el **h2** en la línea 19, se envían al explorador exactamente como aparecen en el archivo ASPX.

21.4.6 Creación de una aplicación Web ASP.NET

Ahora que hemos presentado el archivo ASPX, el archivo de código subyacente y la página Web resultante que se envía al explorador Web, describiremos el proceso mediante el cual creamos esta aplicación. Para crear la aplicación HoraWeb, realice los siguientes pasos en Visual Web Developer:

Paso 1: crear el proyecto de aplicación Web

Seleccione **Archivo > Nuevo sitio Web...** para mostrar el cuadro de diálogo **Nuevo sitio Web** (figura 21.7). En este cuadro de diálogo, seleccione **Sitio Web ASP.NET** en el panel **Plantillas**. Debajo de este panel, el cuadro de diálogo **Nuevo sitio Web** contiene dos campos, con los cuales usted puede especificar el tipo y la ubicación de la aplicación Web que creará. Si no está seleccionado de antemano, seleccione **HTTP** de la lista desplegable más cercana a **Ubicación**. Esto indica que la aplicación Web debe configurarse para ejecutarse como una aplicación IIS

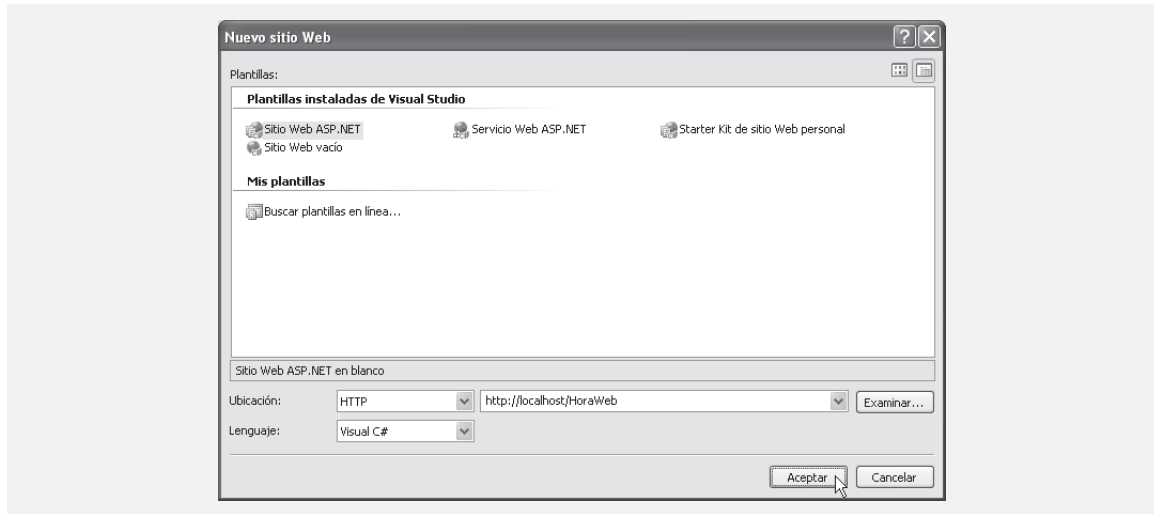


Figura 21.7 | Creación de un **Sitio Web ASP.NET** en Visual Web Developer.

usando HTTP (ya sea en su computadora, o en una computadora remota). Queremos que nuestro proyecto se ubique en `http://localhost`, que es el URL del directorio raíz de IIS (este URL corresponde al directorio `C:\InetPub\wwwroot` en su equipo). El nombre **localhost** indica que el cliente y el servidor residen en el mismo equipo. Si el servidor Web se encontrara en un equipo distinto, `localhost` se sustituiría con la dirección IP o el nombre de host apropiados. De manera predeterminada, Visual Web Developer establece la ubicación en la que se creará el sitio Web a `http://localhost/Website`, que cambiaremos por `http://localhost/HoraWeb`.

Si no tiene acceso a IIS, puede seleccionar **Sistema de archivos** de la lista desplegable que está enseguida de **Ubicación**, para crear la aplicación Web en una carpeta en su computadora. Podrá probar la aplicación utilizando el Servidor de desarrollo ASP.NET interno de Visual Web Developer, pero no podrá acceder a la aplicación en forma remota, a través de Internet.

La lista desplegable **Lenguaje** en el cuadro de diálogo **Nuevo sitio Web** le permite especificar el lenguaje (es decir, Visual Basic, Visual C# o Visual J#) en el que escribirá el (los) archivo(s) de código subyacente para la aplicación Web. Cambie la opción a Visual C#. Haga clic en **Aceptar** para crear el proyecto de la aplicación Web. Esta acción crea el directorio `C:\Inetpub\wwwroot\HoraWeb` y lo hace accesible a través del URL `http://localhost/HoraWeb`. Esta acción también crea un directorio `HoraWeb` dentro del directorio `Visual Studio 2005/Projects`, del directorio `Mis documentos` de su usuario de Windows, para almacenar los archivos de solución del proyecto (por ejemplo, `HoraWeb.sln`).

Paso 2: examinar el Explorador de soluciones del proyecto recién creado

Las siguientes figuras describen el contenido del nuevo proyecto, empezando con el **Explorador de soluciones** que se muestra en la figura 21.8. Al igual que Visual C# 2005 Express, Visual Web Developer crea varios archivos cuando se crea un nuevo proyecto. Se crea un archivo ASPX (es decir, formulario Web Forms) llamado `Default.aspx` para cada nuevo proyecto **Sitio Web ASP.NET**. Este archivo se abre de manera predeterminada en el Diseñador de formularios Web Forms, en modo **Source** cuando se carga el proyecto por primera vez (en breve hablaremos sobre esto). Como dijimos antes, se incluye un archivo de código subyacente como parte del proyecto. Visual Web Developer crea un archivo de código subyacente llamado `Default.aspx.cs`. Para abrir el archivo de código subyacente del archivo ASPX, haga clic con el botón derecho del ratón en el archivo ASPX y seleccione la opción **Ver código** o haga clic en el botón **Ver código** en la parte superior del **Explorador de soluciones**. De manera alternativa, puede expandir el nodo para que el archivo ASPX revele el nodo del archivo de código subyacente (vea la figura 21.8). También puede elegir listar todos los archivos en el proyecto de manera individual (en vez de anidados), haciendo clic en el botón **Anidar archivos relacionados**; esta opción está activada de manera predeterminada, por lo que si hace clic en este botón se desactiva la opción.



Figura 21.8 | Ventana del **Explorador de soluciones** para el proyecto HoraWeb.

Los botones **Propiedades** y **Actualizar** en el **Explorador de soluciones** de Visual Web Developer se comportan en forma idéntica a los de Visual C# 2005 Express. El **Explorador de soluciones** de Visual Web Developer también contiene tres botones adicionales: **Ver diseñador**, **Copiar sitio Web** y **Configuración de ASP.NET**. El botón **Ver Diseñador** le permite abrir el formulario Web en modo de **Diseño**, el cual veremos en breve. El botón **Copiar sitio Web** abre un cuadro de diálogo que le permite desplazar los archivos en este proyecto hacia otra ubicación, como un servidor Web remoto. Esto es útil si está desarrollando la aplicación en su equipo local, pero desea que esté disponible al público desde una ubicación distinta. Por último, el botón **Configuración de ASP.NET** lo lleva a una página Web llamada **Herramienta Administración de sitios Web**, en donde usted podrá manipular varias configuraciones y opciones de seguridad para su aplicación. En la sección 21.8 veremos esta herramienta con mucho más detalle.

Paso 3: examinar el Cuadro de herramientas en Visual Web Developer

La figura 21.9 muestra el **Cuadro de herramientas** que aparece en el IDE cuando se carga el proyecto. La figura 21.9(a) muestra el inicio de la lista **Estándar** de controles Web, y la figura 21.9(b) muestra el resto de los controles Web, así como la lista de controles de **Datos** utilizados en ASP.NET. Hablaremos sobre ciertos controles específicos de la figura 21.9 a medida que los utilicemos en el capítulo. Observe que algunos controles en el **Cuadro de herramientas** son similares a los controles de Windows que presentamos antes en este libro.

Paso 4: examinar el Diseñador de Web Forms

La figura 21.10 muestra el Diseñador de Web Forms en modo **Código**, que aparece en el centro del IDE. Cuando el proyecto se carga por primera vez, el Diseñador de Web Forms muestra el archivo ASPX generado en forma automática (es decir, `Default.aspx`) en modo **Código**, lo cual nos permite ver y editar el marcado que compone la página Web. El IDE creó el marcado que se lista en la figura 21.10, y sirve como una plantilla que modificaremos en unos momentos. Si hacemos clic en el botón **Diseño** que está en la esquina inferior izquierda del Diseñador de Web Forms, el IDE cambia al modo **Diseño** (figura 21.11), que le permite arrastrar y soltar controles del **Cuadro de herramientas** en el formulario Web Forms. También puede escribir en la posición actual del cursor para agregar texto a la página Web. Demostraremos esto en breve. En respuesta a dichas acciones, el IDE genera el marcado apropiado en el archivo ASPX. Observe que el modo **Diseño** indica el elemento de XHTML en el que se encuentra actualmente el cursor. Al hacer clic en el botón **Código**, el Diseñador de Web Forms regresa al modo **Código**, en donde se puede ver el marcado generado.

Paso 5: examinar el archivo de código subyacente en el IDE

La siguiente figura (figura 21.12) muestra a `Default.aspx.cs`, el archivo de código subyacente que Visual Web Developer genera para `Default.aspx`. Haga clic con el botón derecho del ratón en el archivo ASPX, dentro del **Explorador de soluciones**, y seleccione la opción **Ver código** para abrir el archivo de código subyacente. Cuando se crea por primera vez, este archivo no contiene nada más que una declaración de clase parcial con un manejador de eventos `Page_Load` vacío. En unos momentos agregaremos el manejador de eventos `Page_Init` a este código.

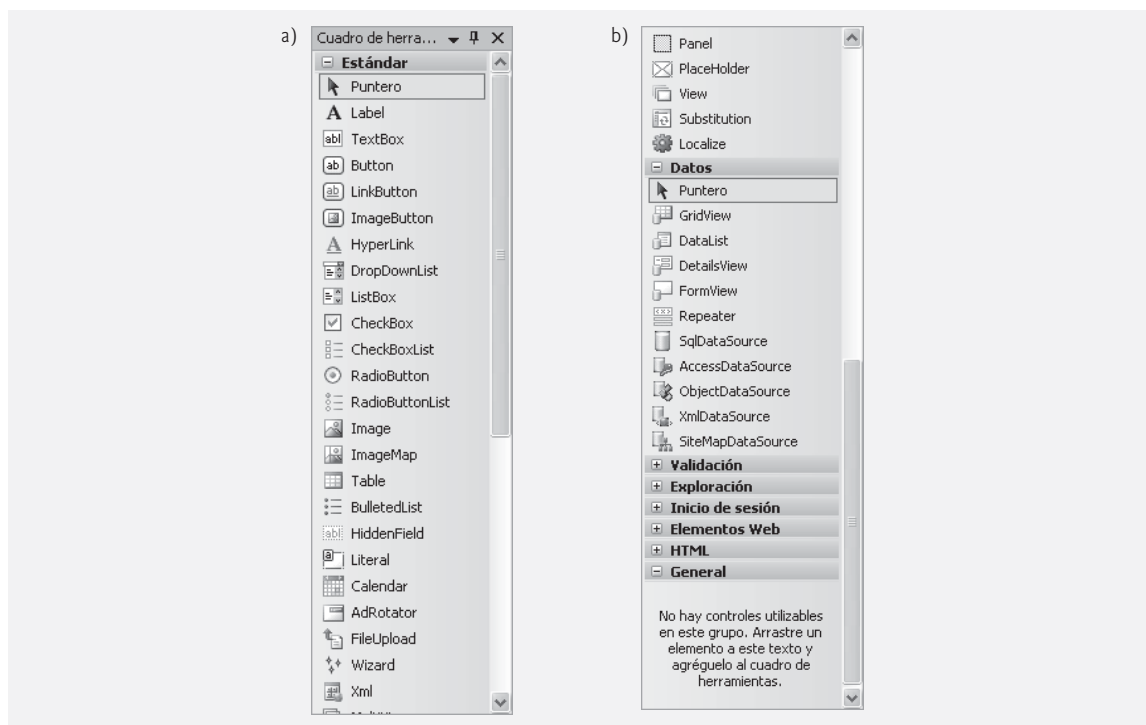


Figura 21.9 | El Cuadro de herramientas en Visual Web Developer.

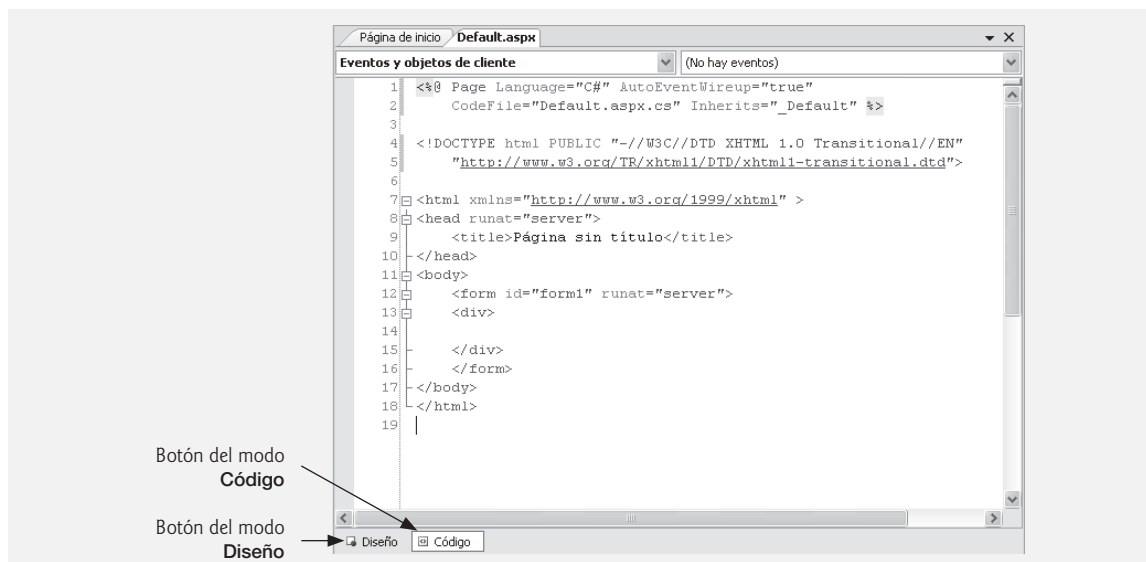


Figura 21.10 | El modo Código del Diseñador de Web Forms.

Paso 6: cambiar el nombre del archivo ASPX

Ya mostramos el contenido de los archivos ASPX predeterminado y de código subyacente. Ahora cambiaremos el nombre de estos archivos. Haga clic con el botón derecho del ratón en el archivo ASPX dentro del **Explorador de soluciones**, y seleccione la opción **Cambiar nombre**. Escriba el nuevo nombre de archivo `HoraWeb.aspx` y oprima *Intro*. Esta operación actualiza el nombre del archivo ASPX y del archivo de código subyacente. Observe que el IDE también actualiza el atributo `CodeFile` de la directiva `Page` de `HoraWeb.aspx`.

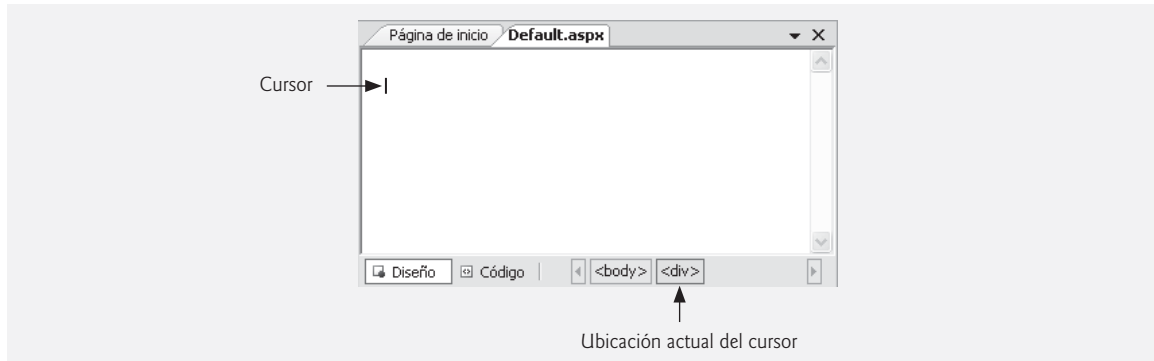


Figura 21.11 | El modo **Diseño** del Diseñador de Web Forms.

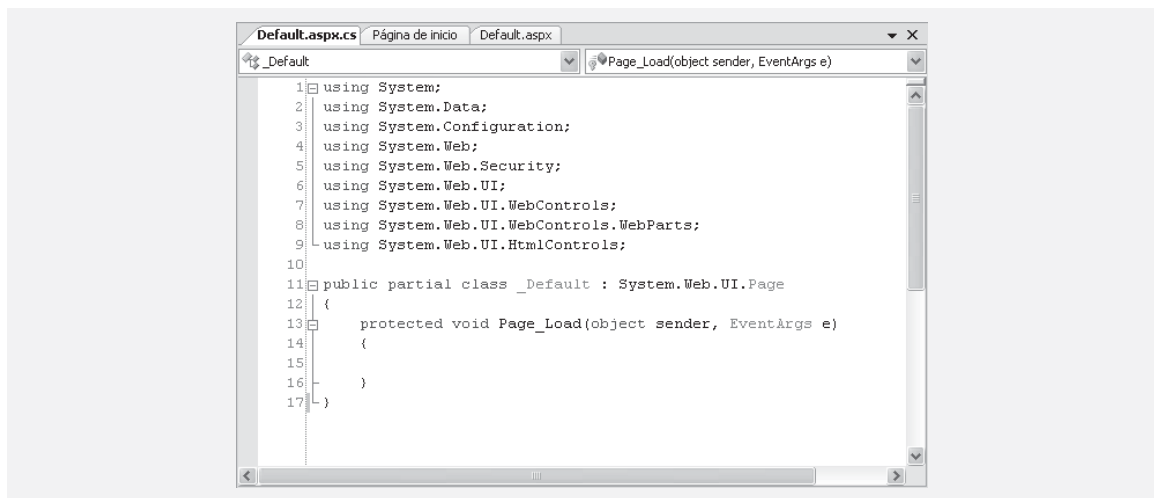


Figura 21.12 | Archivo de código subyacente para Default.aspx, generado por Visual Web Developer.

Paso 7: cambiar el nombre de la clase en el archivo de código subyacente y actualizar el archivo ASPX

Aunque al cambiar el nombre del archivo ASPX, el archivo de código subyacente también cambia su nombre, esta acción no afecta al nombre de la clase parcial declarada en el archivo de código subyacente. Abra el archivo de código subyacente y cambie el nombre de la clase, de `_Default` (línea 11 en la figura 21.12) a `HoraWeb`, de manera que la declaración parcial de la clase aparezca como en la línea 13 de la figura 21.5. Recuerde que la directiva `Page` del archivo ASPX también hace referencia a esta clase. Utilice el modo **Código** del Diseñador de Web Forms para modificar el atributo `Inherits` de la directiva `Page` en `HoraWeb.aspx`, para que aparezca como en la línea 4 de la figura 21.4. El valor del atributo `Inherits` y el nombre de la clase en el archivo de código subyacente deben ser idénticos; de lo contrario, obtendrá errores cuando genere la aplicación Web.

Paso 8: cambiar el título de la página

Antes de diseñar el contenido del formulario Web Forms, hay que cambiar su título del valor predeterminado `Página sin título` (línea 9 de la figura 21.10) por `Un ejemplo simple de un formulario Web Forms`. Para ello, abra el archivo ASPX en modo **Código** y modifique el texto entre las etiquetas `<title>` inicial y final. De manera alternativa, puede abrir el archivo ASPX en modo **Diseño** y modificar la propiedad **Title** en la ventana **Propiedades**. Para ver las propiedades del formulario Web Forms, seleccione `DOCUMENT` de la lista desplegable en la ventana **Propiedades**; **DOCUMENT** es el nombre que se utiliza para representar el formulario Web Forms en la ventana **Propiedades**.

Paso 9: diseñar la página

Diseñar un formulario Web Forms es tan simple como diseñar un formulario Windows. Para agregar controles a la página, puede arrastrarlos y soltarlos desde el **Cuadro de herramientas**, hacia el formulario Web Forms en modo **Diseño**. Al igual que el formulario Web Forms en sí, cada control es un objeto que tiene propiedades, métodos y eventos. Usted puede establecer en forma visual esas propiedades y eventos mediante el uso de la ventana **Propiedades**, o mediante programación en el archivo de código subyacente. No obstante, a diferencia de trabajar con un formulario Windows, puede escribir texto directamente en un formulario Web Forms en la posición del cursor, o insertar elementos de XHTML mediante comandos de menú.

Los controles y otros elementos se colocan en forma secuencial en un formulario Web Forms, de manera muy similar a la forma en que se colocan el texto y las imágenes en un documento, usando software de procesamiento de palabras como Microsoft Word. Los controles se colocan uno después de otro, en el orden en el que usted los arrastre y suelte en el formulario Web Forms. El cursor indica el punto en el cual se van a insertar los elementos de texto y de XHTML. Si desea posicionar un control entre el texto o los controles ya existentes, puede soltarlo en una posición específica dentro de los elementos existentes. También puede reordenar los controles existentes, usando acciones de arrastrar y colocar. Las posiciones de los controles y demás elementos son relativas a la esquina superior izquierda del formulario Web Forms. A este tipo de distribución se le conoce como **posicionamiento relativo**.

Hay un tipo alternativo de distribución conocido como **posicionamiento absoluto**, en el cual los controles se posicionan exactamente en donde se sueltan en el formulario Web Forms. Puede habilitar el posicionamiento absoluto en modo **Diseño** seleccionando **Diseño > Posición > Opciones de autoposición...**, y después haciendo clic en la primer casilla de verificación en el panel **Opciones de posición** del cuadro de diálogo **Opciones** que aparezca.



Tip de portabilidad 21.2

No se recomienda usar el posicionamiento absoluto, ya que las páginas diseñadas de esta forma tal vez no se representen en forma correcta en computadoras con distintas resoluciones de pantalla y tamaños de fuente. Esto podría hacer que los elementos con posicionamiento absoluto se traslaparan entre sí, o que se mostraran fuera de la pantalla, y el cliente tendría que desplazarse por la pantalla para ver el contenido de la página completa.

En este ejemplo, utilizamos una pieza de texto y un control `Label`. Para agregar el texto al formulario Web Forms, haga clic en este formulario en modo **Diseño** y escriba `Hora actual en el servidor Web:`. Visual Web Developer es un editor **WYSIWYG** (*Lo que se ve es lo que se obtiene*); cada vez que usted realiza una modificación a un formulario Web Forms en modo de **Diseño**, el IDE crea el marcado (visible en modo **Código**) necesario para lograr los efectos visuales deseados que se ven en modo **Diseño**. Después de agregar el texto al formulario Web Forms, cambie a modo **Código**. En este modo podrá ver que el IDE agregó este texto al elemento `div` que aparece en el archivo `ASPX` de manera predeterminada. Regrese al modo **Diseño** y resalte el texto que agregó. En la lista desplegable **Formato del bloque** (vea la figura 21.13), seleccione **Heading 2** para dar formato a este texto como un encabezado, que aparecerá en negritas, en una fuente un poco más grande que la predeterminada. Esta acción hace que el IDE encierre el texto recién escrito en un elemento `h2`. Por último, haga clic a la derecha del texto y oprima **Intro** para desplazar el cursor a un nuevo párrafo. Esta acción genera un elemento `p` vacío en el marcado del archivo `ASPX`. Ahora, el IDE deberá verse como la figura 21.13.

Para colocar un control `Label` en un formulario Web Forms, puede arrastrar y soltar el control **Label** sobre el formulario, o hacer doble clic en este control dentro del **Cuadro de herramientas**. Asegúrese que el cursor se encuentre en el párrafo recién creado y después agregue un control `Label`, el cual se utilizará para mostrar la hora. Use la ventana **Propiedades** para establecer la propiedad (ID) del control `Label` a `horaLabel1`. Eliminamos el texto de `horaLabel1`, ya que este texto se establecerá mediante programación en el archivo de código subyacente. Cuando un control `Label` no contiene texto, su nombre aparece entre corchetes en el Diseñador de Web Forms (figura 21.14), pero no se muestra en tiempo de ejecución. El nombre de la etiqueta es un receptáculo para fines de diseño y distribución. Establezca las propiedades `BackColor`, `ForeColor` y `Font-Size` de `horaLabel1` a `Black`, `Yellow` y `XX-Large`, respectivamente. Para cambiar las propiedades de la fuente, expanda el nodo **Font** en la ventana **Propiedades** y después modifique cada una de las propiedades relevantes de manera individual. Una vez que se establezcan las propiedades del control `Label` en la ventana **Propiedades**, Visual Web Developer actualizará el contenido del archivo `ASPX`. La figura 21.14 muestra el IDE, una vez que se establecen estas propiedades.

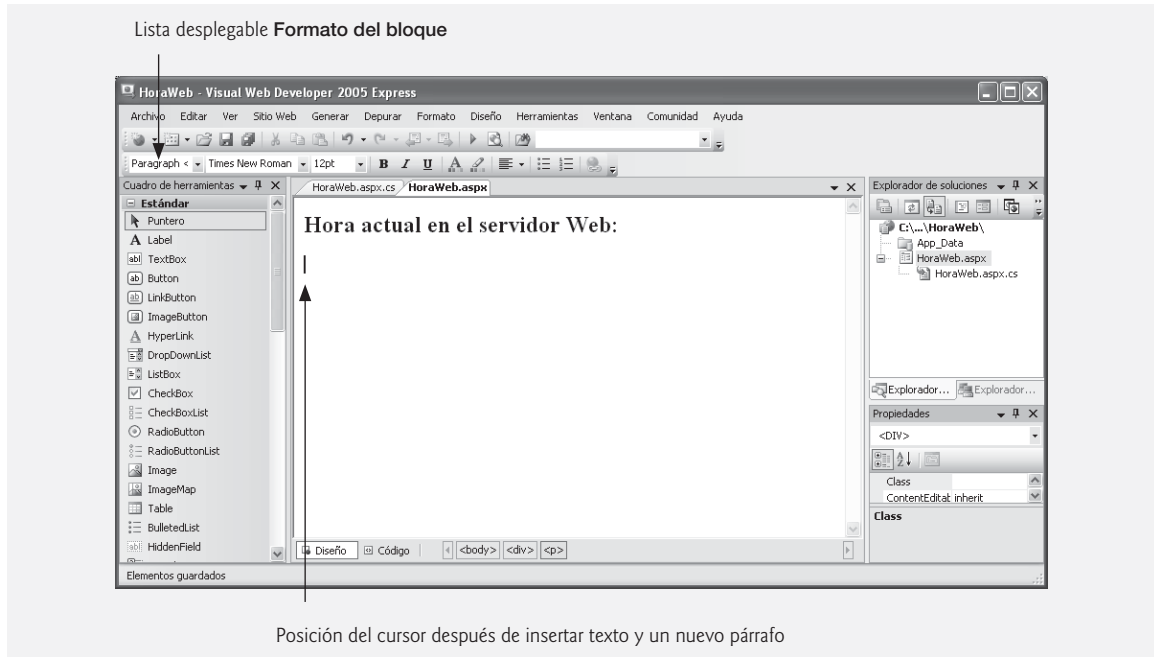


Figura 21.13 | HoraWeb.aspx después de insertar texto y un nuevo párrafo.

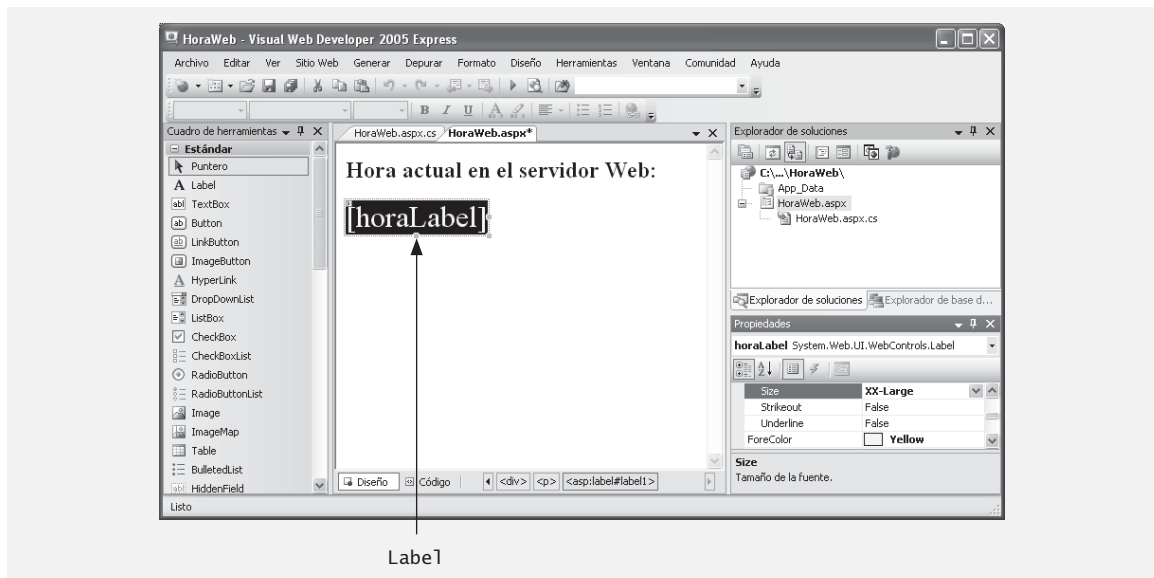


Figura 21.14 | HoraWeb.aspx después de agregar un control Label y establecer sus propiedades.

A continuación, hay que establecer la propiedad `EnableViewState` del control `Label` a `False`. Por último, seleccione `DOCUMENT` de la lista desplegable en la ventana **Propiedades** y establezca la propiedad `EnableSession-State` del formulario Web Forms a `False`. Más adelante en este capítulo hablaremos sobre estas dos propiedades.

Paso 10: agregar la lógica de la página

Una vez diseñada la interfaz de usuario, hay que agregar código en C# al archivo de código subyacente. Abra el archivo `HoraWeb.aspx.cs`, haciendo doble clic sobre su nodo en el **Explorador de soluciones**. En este ejemplo,

agregaremos un manejador de eventos `Page_Init` (líneas 16-21 de la figura 21.5) al archivo de código subyacente. Recuerde que `Page_Init` maneja el evento `Init` y contiene código para inicializar la página. La instrucción en las líneas 19-20 de la figura 21.5 establece mediante programación el texto de `horaLabel` con la hora actual en el servidor.

Paso 11: ejecutar el programa

Una vez creado el formulario Web Forms, puede verlo de varias formas. Primero, puede seleccionar **Depurar > Iniciar sin depurar**, con lo cual se abre una ventana del explorador para ejecutar la aplicación. Si creó la aplicación en su servidor IIS local (como hicimos en este ejemplo), el URL que aparezca en el explorador será `http://localhost/HoraWeb/HoraWeb.aspx` (figura 21.5), lo cual indica que la página Web (el archivo ASPX) se encuentra dentro del directorio virtual `HoraWeb` en el servidor Web IIS local. Para probar el sitio Web en un explorador, IIS debe estar ejecutándose. Para iniciar IIS, ejecute el archivo `inetmgr.exe` desde **Inicio > Ejecutar...**, haga clic con el botón derecho del ratón en **Sitio Web predeterminado** y seleccione **Start**. [Nota: Tal vez necesite expandir el nodo que representa a su equipo para visualizar el **Sitio Web predeterminado**.]

Tenga en cuenta que, si creó la aplicación ASP.NET en el sistema de archivos local, el URL que aparecerá en el explorador será `http://localhost:NumeroPuerto/HoraWeb/HoraWeb.aspx`, en donde *NumeroPuerto* es el número del puerto asignado al azar en el que se ejecuta el servidor de prueba integrado de Visual Web Developer. El IDE asigna el número de puerto con base en cada solución. Este URL indica que se está accediendo a la carpeta del proyecto `HoraWeb` a través del directorio raíz del servidor de prueba que se ejecuta en `localhost:NumeroPuerto`. Cuando seleccione **Depurar > Iniciar sin depurar**, aparecerá un icono en la bandeja de notificación, cerca de la parte inferior derecha de su pantalla, enseguida de la fecha y hora de la computadora, para mostrar que el **Servidor de desarrollo ASP.NET** se está ejecutando. El servidor se detiene cuando usted sale de Visual Web Developer.

También puede seleccionar **Depurar > Iniciar depuración** para ver la página Web en un explorador Web con depuración habilitada. Tenga en cuenta que no puede depurar un sitio Web a menos que se habilite explícitamente la depuración mediante el archivo **Web.config**; un archivo que almacena opciones de configuración para una aplicación Web de ASP.NET. Muy raras veces tendrá que crear o modificar el archivo `Web.config` en forma manual. La primera vez que selecciona **Depurar > Iniciar depuración** en un proyecto, aparece un cuadro de diálogo y le pregunta si desea que el IDE genere el archivo `Web.config` necesario y lo agregue al proyecto; después el IDE entra en modo **Ejecución**. Para salir del modo **Ejecución**, seleccione **Depurar > Detener depuración** en Visual Web Developer, o cierre la ventana del explorador que muestra el sitio Web.

También puede hacer clic con el botón derecho del ratón en el Diseñador de Web Forms o en el nombre del archivo ASPX (en el **Explorador de soluciones**) y seleccionar la opción **Ver en el explorador**, para abrir una ventana del explorador y cargar una página Web. Si hace clic, con el botón derecho del ratón, en el archivo ASPX dentro del **Explorador de soluciones** y selecciona **Explorar con...** también se abre la página en un explorador, pero primero nos permite especificar el explorador Web que debe abrir la página, junto con su resolución de pantalla.

Por último, puede ejecutar su aplicación abriendo una ventana del explorador y escribiendo el URL de la página Web en el campo **Dirección**. Si está probando una aplicación ASP.NET en la misma computadora que ejecuta IIS, escriba `http://localhost/CarpetaProyecto/NombrePágina.aspx`, en donde *CarpetaProyecto* es la carpeta en la que reside la página (por lo general, el nombre del proyecto), y *NombrePágina* es el nombre de la página ASP.NET. Si su aplicación reside en el sistema de archivos local, primero debe iniciar el **Servidor de desarrollo ASP.NET** ejecutando la aplicación mediante uno de los métodos antes descritos. Después puede escribir el URL (incluyendo el *NumeroPuerto* que aparezca en el icono del servidor de prueba en la bandeja de notificación) en el explorador para ejecutar la aplicación.

Observe que todos estos métodos para ejecutar la aplicación compilan el proyecto por usted. De hecho, ASP.NET compila su página Web cada vez que ésta cambia entre las peticiones HTTP. Por ejemplo, suponga que explora la página y después modifica el archivo ASPX, o agrega código al archivo de código subyacente. Cuando vuelva a cargar la página, ASP.NET la recompilará en el servidor antes de devolver la respuesta HTTP al usuario. Este nuevo e importante comportamiento de ASP.NET 2.0 asegura que el cliente que solicita la página siempre reciba la versión más reciente de la misma. No obstante, usted puede compilar una página Web, o todo un sitio Web completo, seleccionando **Generar página** o **Generar sitio Web**, respectivamente, en el menú **General** de Visual Web Developer.

Si desea probar su aplicación Web a través de una red, tal vez necesite modificar su configuración de Firewall de Windows. Por cuestiones de seguridad, Firewall de Windows no permite el acceso remoto a un servidor Web en su equipo local de manera predeterminada. Para modificar esta configuración, abra la herramienta Firewall de Windows en el Panel de control de Windows. Haga clic en la ficha **Opciones avanzadas** y seleccione su conexión de red de la lista **Configuración de conexión de red**. Después haga clic en el botón **Configuración...**. En la ficha **Servicios** del cuadro de diálogo **Configuración avanzada**, asegúrese que esté seleccionada la opción **Servidor Web (HTTP)**.

21.5 Controles Web

Esta sección presenta algunos de los controles Web ubicados en la sección **Estándar** del **Cuadro de herramientas** (figura 21.9). La figura 21.15 muestra un resumen de algunos de los controles Web utilizados en los ejemplos de este capítulo.

21.5.1 Controles de texto y gráficos

La figura 21.16 ilustra un formulario simple para recopilar la entrada del usuario. Este ejemplo utiliza todos los controles listados en la figura 21.15 excepto el control `Label`, que utilizamos en la sección 21.4. Observe que todo el código de la figura 21.16 lo generó Visual Web Developer, en respuesta a las acciones realizadas en el modo **Diseño**. [Nota: este ejemplo no contiene ninguna funcionalidad; es decir, no ocurre acción alguna cuando el usuario hace clic en **Registrar**. Le pediremos a usted que proporcione la funcionalidad como un ejercicio. En los ejemplos subsiguientes, demostraremos cómo agregar funcionalidad a muchos de estos controles Web.]

Antes de hablar sobre los controles Web utilizados en este archivo ASPX, explicaremos el XHTML que crea la distribución que podemos ver en la figura 21.16. La página contiene un elemento de encabezado `h3` (línea 17), seguido de una serie de bloques de XHTML adicionales. Colocamos la mayor parte de los controles Web dentro de elementos `p` (es decir, párrafos), pero usamos un elemento `table` de XHTML (líneas 26-60) para organizar los controles `Image` y `TextBox` en la sección de la página que trata acerca de la información sobre el usuario. En la sección anterior describimos cómo agregar elementos de encabezado y párrafos en forma visual, sin manipular XHTML en el archivo ASPX directamente. Visual Web Developer nos permite agregar una tabla de una manera similar.

Agregar una tabla de XHTML a un formulario Web Forms

Para crear una tabla con dos filas y dos columnas en modo **Diseño**, seleccione el comando **Insertar tabla** del menú **Diseño**. En el cuadro de diálogo **Insertar tabla** que aparezca, asegúrese que el botón de opción **Personalizada** esté seleccionado. En el cuadro de grupo **Diseño**, modifique los valores de los cuadros combinados **Filas** y **Columnas** a 2. De manera predeterminada, el contenido de la celda de una tabla se alinea en forma vertical con la parte media de la celda. Para modificar la alineación vertical de todas las celdas en la tabla, haga clic en el botón **Propiedades de celda...** y después seleccione la opción **top** del cuadro combinado **Alineación vertical** en el cuadro de diálogo que

Control Web	Descripción
<code>Label</code>	Muestra texto que el usuario no puede editar.
<code>TextBox</code>	Recopila la entrada del usuario y muestra texto.
<code>Button</code>	Activa un evento cuando se oprime.
<code>HyperLink</code>	Muestra un hipervínculo.
<code>DropDownList</code>	Muestra una lista desplegable de opciones, de donde el usuario puede seleccionar un elemento.
<code>RadioButtonList</code>	Agrupar botones de opción.
<code>Image</code>	Muestra imágenes (por ejemplo, GIF y JPG).

Figura 21.15 | Controles Web de uso frecuente.

aparezca. Esto hace que el contenido de cada celda de la tabla se alinee con la parte superior de la celda. Haga clic en **Aceptar** para cerrar el cuadro de diálogo **Propiedades de celda** y después haga clic en **Aceptar** para cerrar el cuadro de diálogo **Insertar tabla** y crear la tabla. Una vez creada, podrá agregar controles y texto a celdas específicas, para crear un diseño muy bien organizado.

Seleccionar el color del texto en un formulario Web Forms

Observe que algunas de las instrucciones para el usuario en el formulario aparecen en color verde azulado. Para establecer el color de una pieza específica de texto, resalte el texto y seleccione la opción **Formato > Color de primer plano....** En el cuadro de diálogo **Selector de color**, haga clic en la ficha **Colores con nombre** y seleccione un color de la paleta que se muestra. Haga clic en **Aceptar** para aplicar el color. Observe que el IDE coloca el texto de color en un elemento `span` de XHTML (por ejemplo, líneas 23-24) y aplica el color usando el atributo `style` de `span`.

```

1  <!-- Fig. 21.16: ControlesWeb.aspx -->
2  <!-- Formulario de registro que demuestra los controles Web. -->
3  <%@ Page Language="C#" AutoEventWireup="true"
4      CodeFile="ControlesWeb.aspx.cs" Inherits="ControlesWeb"
5      EnableSessionState="False" %>
6
7  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
8      "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
9
10 <html xmlns="http://www.w3.org/1999/xhtml">
11   <head id="Head1" runat="server">
12     <title>Demostración de los controles Web</title>
13   </head>
14   <body>
15     <form id="form1" runat="server">
16       <div>
17         <h3>Éste es un formulario de registro simple.</h3>
18         <p><em>Por favor llene todos los campos y haga clic en Registrar.</em></p>
19         <p>
20           <asp:Image ID="InformacionUsuarioImage" runat="server"
21             ImageUrl="~/Imagenes/usuario.png" EnableViewState="False" />
22           &nbsp;
23           <span style="color: teal">
24             Por favor llene los siguientes campos.</span>
25         </p>
26         <table>
27           <tr>
28             <td style="width: 230px; height: 21px; valign="top">
29               <asp:Image ID="PrimerNombreImage" runat="server"
30                 ImageUrl="~/Imagenes/pnombre.png"
31                 EnableViewState="False" />
32               <asp:TextBox ID="PrimerNombreTextBox" runat="server"
33                 EnableViewState="False"></asp:TextBox>
34             </td>
35             <td style="width: 245px; height: 21px; valign="top">
36               <asp:Image ID="ApellidoPaternoImage" runat="server"
37                 ImageUrl="~/Imagenes/apaterno.png"
38                 EnableViewState="False" />
39               <asp:TextBox ID="ApellidoPaternoTextBox" runat="server"
40                 EnableViewState="False"></asp:TextBox>
41             </td>
42           </tr>
43           <tr>

```

Figura 21.16 | Formulario Web que demuestra el uso de los controles Web. (Parte I de 3).

```

44         <td style="width: 230px" valign="top">
45             <asp:Image ID="EmailImage" runat="server"
46                 ImageUrl="~/Imagenes/email.png"
47                 EnableViewState="False" />
48             <asp:TextBox ID="EmailTextBox" runat="server"
49                 EnableViewState="False"></asp:TextBox>
50         </td>
51         <td style="width: 245px" valign="top">
52             <asp:Image ID="TelefonoImage" runat="server"
53                 ImageUrl="~/Imagenes/telefono.png"
54                 EnableViewState="False" />
55             <asp:TextBox ID="TelefonoTextBox" runat="server"
56                 EnableViewState="False"></asp:TextBox>
57             Debe tener el formato (555) 555-5555.
58         </td>
59     </tr>
60 </table>
61 <p>
62     <asp:Image ID="PublicacionesImage" runat="server"
63         ImageUrl="~/Imagenes/publicaciones.png"
64         EnableViewState="False" />
65     &nbsp;
66     <span style="color: teal">
67         ¿De cuál libro desea información?</span>
68 </p>
69 <p>
70     <asp:DropDownList ID="LibrosDropDownList" runat="server"
71         EnableViewState="False">
72         <asp:ListItem>Visual Basic 2005 How to Program 3e
73         </asp:ListItem>
74         <asp:ListItem>Visual C# 2005 How to Program 2e
75         </asp:ListItem>
76         <asp:ListItem>Java How to Program 6e</asp:ListItem>
77         <asp:ListItem>C++ How to Program 5e</asp:ListItem>
78         <asp:ListItem>XML How to Program 1e</asp:ListItem>
79     </asp:DropDownList>
80 </p>
81 <p>
82     <asp:HyperLink ID="LibrosHyperLink" runat="server"
83         NavigateUrl="http://www.deitel.com" Target="_blank"
84         EnableViewState="False">
85         Haga clic aquí para ver más información acerca de nuestros libros
86     </asp:HyperLink>
87 </p>
88 <p>
89     <asp:Image ID="SOImage" runat="server"
90         ImageUrl="~/Imagenes/so.png" EnableViewState="False" />
91     &nbsp;
92     <span style="color: teal">
93         ¿Qué sistema operativo utiliza?</span>
94 </p>
95 <p>
96     <asp:RadioButtonList ID="SistemaOperativoRadioButtonList"
97         runat="server" EnableViewState="False">
98         <asp:ListItem>Windows XP</asp:ListItem>
99         <asp:ListItem>Windows 2000</asp:ListItem>
100        <asp:ListItem>Windows NT</asp:ListItem>
101        <asp:ListItem>Linux</asp:ListItem>
102        <asp:ListItem>Otro</asp:ListItem>

```

Figura 21.16 | Formulario Web que demuestra el uso de los controles Web. (Parte 2 de 3).

```

103     </asp:RadioButtonList>
104 </p>
105 <p>
106     <asp:Button ID="RegistrarButton" runat="server"
107         Text="Registrar" EnableViewState="False" />
108 </p>
109 </div>
110 </form>
111 </body>
112 </html>

```

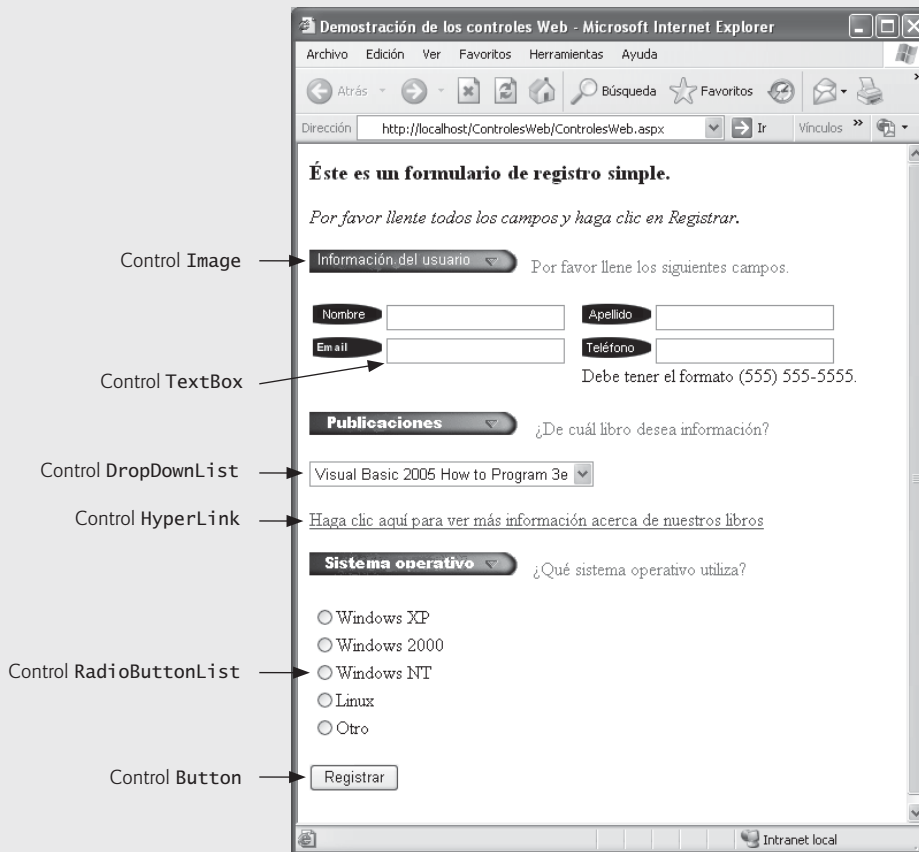


Figura 21.16 | Formulario Web que demuestra el uso de los controles Web. (Parte 3 de 3).

Examinar los controles Web en un ejemplo de formulario de registro

Las líneas 20-21 de la figura 21.16 definen un control **Image**, el cual inserta una imagen en una página Web. Las imágenes que utilizamos en este ejemplo se encuentran en el directorio de ejemplos de este capítulo. Puede descargar los ejemplos de www.deitel.com/books/csharpforprogrammers2. Antes de poder mostrar una imagen en una página Web mediante el uso de un control Web Image, hay que agregar la imagen al proyecto. Nosotros agregamos una carpeta llamada **Imágenes** a este proyecto (y a cada uno de los proyectos de ejemplo de este capítulo que utilizan imágenes); para ello, hicimos clic con el botón derecho en la ubicación del proyecto dentro del **Explorador de soluciones**, seleccionamos la opción **Nueva carpeta** y escribimos el nombre **Imágenes** para la carpeta. Después agregamos a esta carpeta cada una de las imágenes utilizadas en el ejemplo; para ello, hicimos clic con el botón derecho del ratón sobre la carpeta, seleccionamos **Agregar elemento existente...** y exploramos en busca de los archivos de imagen para agregarlos.

La propiedad **ImageUrl** (línea 21) especifica la ubicación de la imagen que se mostrará en el control Image. Para seleccionar una imagen, haga clic en la elipsis que está enseguida de la propiedad ImageUrl en la ventana **Propiedades**, y utilice el cuadro de diálogo **Seleccionar imagen** para explorar en busca de la imagen deseada en la carpeta Imágenes del proyecto. Cuando el IDE llena la propiedad ImageUrl con base en su selección, incluye una tilde y una barra diagonal (~/) al principio de ImageUrl; esto indica que la carpeta Imágenes se encuentra en el directorio raíz del proyecto (es decir, <http://localhost/ControlesWeb>, cuya ruta física es C:\Inetpub\wwwroot\ControlesWeb).

Las líneas 26-60 contienen el elemento table creado por los pasos que vimos antes. Cada elemento td contiene un control Image y un control **TextBox**, el cual le permite obtener texto del usuario y mostrar texto al usuario. Por ejemplo, las líneas 32-33 definen un control TextBox que se utiliza para recolectar el primer nombre del usuario.

Las líneas 70-79 definen un control **DropDownList**. Este control es similar al control ComboBox de Windows. Cuando un usuario hace clic en la lista desplegable, ésta se expande y muestra una lista para que el usuario pueda realizar una selección. Cada elemento en la lista desplegable se define mediante un elemento **ListItem** (líneas 72-78). Después de arrastrar un control DropDownList hacia un formulario Web Forms, puede agregar elementos en él usando el **Editor de la colección ListItem**. Este proceso es similar al de personalizar un control ListBox en una aplicación para Windows. En Visual Web Developer, puede acceder al **Editor de la colección ListItem** haciendo clic en la elipsis que está a un lado de la propiedad Items del control DropDownList. También puede acceder a este editor usando el menú **Tareas de DropDownList**, el cual se abre al hacer clic en la pequeña punta de flecha que aparece en la esquina superior derecha del control en el modo **Diseño** (figura 21.17). A este menú se le conoce como *menú de etiquetas inteligentes*. Visual Web Developer muestra menús de etiquetas inteligentes para muchos controles ASP.NET, para facilitar la realización de tareas comunes. Al hacer clic en la opción **Editar elementos...** del menú **Tareas de DropDownList** se abre el **Editor de la colección ListItem**, el cual le permite agregar elementos ListItem al control DropDownList.

El control **HyperLink** (líneas 82-86 de la figura 21.16) agrega un hipervínculo a una página Web. La propiedad **NavigateUrl** (línea 83) de este control especifica el recurso (es decir, <http://www.deitel.com>) que se solicita cuando un usuario hace clic en el hipervínculo. Al establecer la propiedad **Target** a **_blank** se especifica que la página Web solicitada debe abrirse en una nueva ventana del explorador. De manera predeterminada, los controles HyperLink hacen que se abran páginas en la misma ventana del explorador.

Las líneas 96-103 definen un control **RadioButtonList**, el cual proporciona una serie de botones de opción, de los que el usuario sólo puede seleccionar uno. Al igual que las opciones en un control DropDownList, los botones de opción individuales se definen mediante elementos ListItem. Observe que, al igual que el menú de etiquetas inteligentes **Tareas de DropDownList**, el menú de etiquetas inteligentes **Tareas de RadioButtonList** también cuenta con un vínculo **Editar elementos...** para abrir el **Editor de la colección ListItem**.

El último control Web en la figura 21.16 es **Button** (líneas 106-107). Al igual que un control Button de Windows, el control Web Button representa a un botón que desencadena una acción cuando se hace clic sobre él. Por lo general, un control Web Button se asigna a un elemento input de XHTML, cuyo atributo type se establece a "button". Como dijimos antes, al hacer clic en el botón **Registrar** en el ejemplo no pasa nada.

21.5.2 Control AdRotator

A menudo, las páginas Web contienen anuncios de productos o servicios, los que por lo general consisten en imágenes. Aunque los autores de sitios Web desean incluir todos los patrocinadores que sea posible, las páginas Web sólo pueden mostrar un número limitado de anuncios. Para tratar este problema, ASP.NET cuenta con el control Web **AdRotator** para mostrar anuncios. Usando los datos sobre los anuncios ubicados en un archivo de

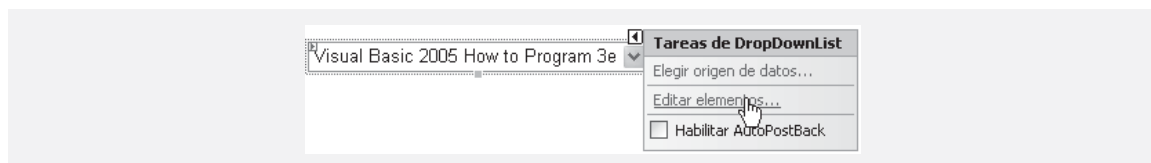


Figura 21.17 | Menú de etiquetas inteligentes **Tareas de DropDownList**.

XML, el control `AdRotator` selecciona una imagen al azar para mostrarla en la página, y genera un hipervínculo a la página Web asociada con esa imagen. Los exploradores que no soportan imágenes muestran texto alternativo, el cual se especifica en el documento XML. Si un usuario hace clic en la imagen o en el texto que la sustituye, el explorador carga la página Web asociada con esa imagen.

Demostración del control Web AdRotator

La figura 21.18 demuestra el control Web `AdRotator`. En este ejemplo, los “anuncios” que rotamos son las banderas de 10 países. Cuando un usuario hace clic en la imagen de la bandera mostrada, el explorador se redirecciona a una página Web que contiene información acerca del país que representa la bandera. Si un usuario actualiza el explorador o solicita la página de nuevo, se vuelve a elegir al azar una de las 11 banderas y se muestra en la pantalla.

El archivo ASPX en la figura 21.18 es similar al de la figura 21.4. Sin embargo, en vez de texto de XHTML y un control `Label`, esta página contiene texto de XHTML (es decir, el elemento `h3` en la línea 17) y un control `AdRotator` llamado `paisRotator` (líneas 19-21). Esta página también contiene un control `XmlDataSource` (líneas 22-24), el cual provee los datos al control `AdRotator`. El atributo `background` del elemento `body` de la página (línea 14) se establece para mostrar la imagen `fondo.png`, ubicada en la carpeta `Imágenes` del proyecto. Para especificar este archivo, haga clic en la elipsis que se encuentra a un lado de la propiedad `Background` de `DOCUMENT` en la ventana **Propiedades**, y utilice el cuadro de diálogo que aparezca para explorar en busca de `fondo.png`.

No necesita agregar código al archivo de código subyacente, debido a que el control `AdRotator` realiza “todo el trabajo”. Las ventanas de resultados muestran dos peticiones distintas. La figura 21.18(a) muestra la primera vez que se solicita la página, cuando se muestra la bandera estadounidense. En la segunda solicitud, como se muestra en la figura 21.18(b), se muestra la bandera francesa. La figura 21.18(c) muestra la página Web que se carga al hacer clic en la bandera francesa.

```

1  <!-- Fig. 21.18: BanderaRotator.aspx -->
2  <!-- Un formulario Web Forms que muestra banderas mediante el uso de un control
    AdRotator. -->
3  <%@ Page Language="C#" AutoEventWireup="false"
4      CodeFile="BanderaRotator.aspx.cs" Inherits="BanderaRotator"
5      EnableSessionState="False" %>
6
7  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
8      "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
9
10 <html xmlns="http://www.w3.org/1999/xhtml" >
11   <head id="Head1" runat="server">
12     <title>Rotador de banderas</title>
13   </head>
14   <body background="Imágenes/fondo.png">
15     <form id="form1" runat="server">
16       <div>
17         <h3>Ejemplo de AdRotator</h3>
18         <p>
19           <asp:AdRotator ID="paisRotator" runat="server"
20             DataSourceID="adXmlDataSource"
21             EnableViewState="False" />
22           <asp:XmlDataSource ID="adXmlDataSource" runat="server"
23             DataFile="~/App_Data/InformacionAdRotator.xml">
24             </asp:XmlDataSource>
25         </p>
26       </div>
27     </form>
28   </body>
29 </html>

```

Figura 21.18 | Formulario Web Forms que demuestra el uso de un control Web `AdRotator`. (Parte I de 2).

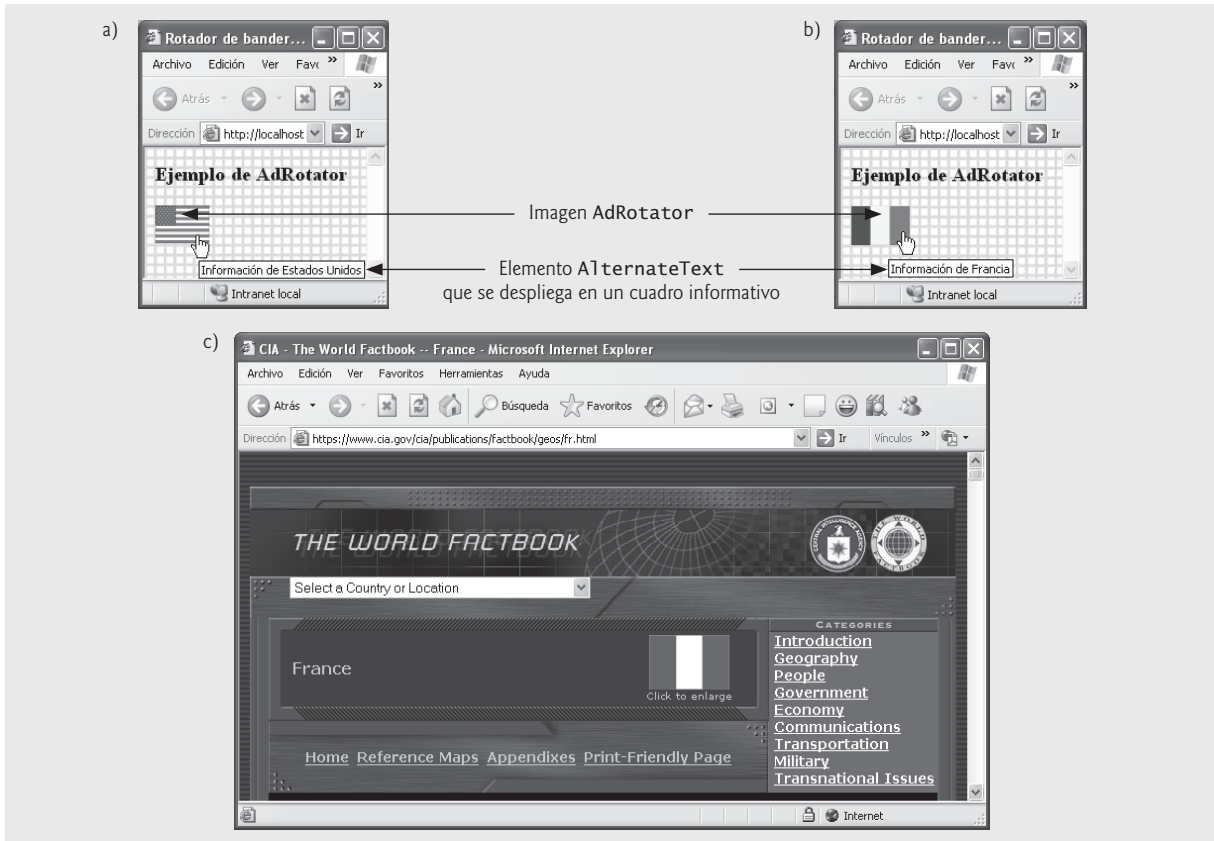


Figura 21.18 | Formulario Web Forms que demuestra el uso de un control Web AdRotator. (Parte 2 de 2).

Conectar datos a un control AdRotator

Un control AdRotator accede a un archivo de XML (que presentaremos en breve) para determinar de cuál anuncio se mostrará la imagen (es decir, bandera), el URL de hipervínculo y el texto alternativo, e incluirlos en la página. Para conectar el control AdRotator con el archivo de XML, creamos un control **XmlDataSource**; uno de varios controles de datos de ASP.NET (se encuentra en la sección **Datos** del **Cuadro de herramientas**) que encapsula orígenes de datos y hace que dichos datos estén disponibles para los controles Web. Un control XmlDataSource hace referencia a un archivo de XML, el cual contiene los datos que se utilizarán en una aplicación ASP.NET. Más adelante en este capítulo aprenderá más acerca de los controles Web enlazados a datos, así como sobre el control **SqlDataSource**, que recupera datos de una base de datos de SQL Server, y del control **ObjectDataSource**, que encapsula a un objeto que hace que los datos estén disponibles.

Para crear este ejemplo, primero hay que agregar el archivo de XML **InformacionAdRotator.xml** al proyecto. Cada proyecto que se crea en Visual Web Developer contiene una carpeta **App_Data**, cuya función es almacenar todos los datos utilizados por el proyecto. Haga clic con el botón derecho del ratón sobre esta carpeta en el **Explorador de soluciones** y seleccione la opción **Agregar elemento existente...**, después explore su computadora para buscar el archivo **InformacionAdRotator.xml**. (Nosotros incluimos este archivo en el directorio de ejemplos de este capítulo.)

Después de agregar el archivo de XML al proyecto, arrastre un control AdRotator del **Cuadro de herramientas** al formulario Web Forms. El menú de etiquetas inteligentes **Tareas de AdRotator** se abrirá en forma automática. De este menú, seleccione **<Nuevo origen de datos...>** de la lista desplegable **Elegir origen de datos** para iniciar el **Asistente para la configuración de orígenes de datos**. Seleccione **Archivo XML** como el tipo de origen de datos. Esto hace que el asistente cree un control **XmlDataSource** con el ID especificado en la mitad inferior del cuadro de diálogo del asistente. Cambie el nombre del ID del control a **adXmlDataSource**. Haga clic en el botón

Aceptar del cuadro de diálogo **Asistente para la configuración de orígenes de datos**. A continuación aparecerá el cuadro de diálogo **Configurar origen de datos – adXmlDataSource**. En la sección **Archivo de datos** de este cuadro de diálogo, haga clic en **Examinar...** y, en el cuadro de diálogo **Seleccionar archivo XML**, localice el archivo de XML que agregó a la carpeta `App_Data`. Haga clic en **Aceptar** para salir de este cuadro de diálogo y después haga clic en **Aceptar** para salir del cuadro de diálogo **Configurar origen de datos – adXmlDataSource**. Al completar estos pasos, el control `AdRotator` estará configurado para usar el archivo XML y determinar cuáles anuncios debe mostrar.

Examinar un archivo de XML que contiene información sobre anuncios

El documento de XML `InformacionAdRotator.xml` (figura 21.19) (o cualquier documento de XML que se utilice con un control `AdRotator`) debe contener un elemento raíz llamado **Advertisements** (líneas 4-94). Dentro de ese elemento puede haber varios elementos **Ad** (por ejemplo, las líneas 5-12), cada uno de los cuales proporciona información acerca de un anuncio distinto. El elemento **ImageUrl** (línea 6) especifica la ruta (ubicación) de la imagen del anuncio, y el elemento **NavigateUrl** (líneas 7-9) especifica el URL para la página Web que se carga cuando un usuario hace clic sobre el anuncio. Observe que cambiamos el formato de este archivo para fines de presentación. El archivo XML real no puede contener espacio en blanco antes o después del URL en el elemento `NavigateUrl`, ya que se consideraría parte del URL, y la página no se cargaría en forma apropiada.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <!-- Fig. 21.19: InformacionAdRotator.xml -->
3  <!-- Archivo de XML que contiene información sobre los anuncios. -->
4  <Advertisements>
5      <Ad>
6          <ImageUrl>Imagenes/francia.png</ImageUrl>
7          <NavigateUrl>
8              http://www.cia.gov/cia/publications/factbook/geos/fr.html
9          </NavigateUrl>
10         <AlternateText>Información de Francia</AlternateText>
11         <Impressions>1</Impressions>
12     </Ad>
13
14     <Ad>
15         <ImageUrl>Imagenes/alemania.png</ImageUrl>
16         <NavigateUrl>
17             http://www.cia.gov/cia/publications/factbook/geos/gm.html
18         </NavigateUrl>
19         <AlternateText>Información de Alemania</AlternateText>
20         <Impressions>1</Impressions>
21     </Ad>
22
23     <Ad>
24         <ImageUrl>Imagenes/italia.png</ImageUrl>
25         <NavigateUrl>
26             http://www.cia.gov/cia/publications/factbook/geos/it.html
27         </NavigateUrl>
28         <AlternateText>Información de Italia</AlternateText>
29         <Impressions>1</Impressions>
30     </Ad>
31
32     <Ad>
33         <ImageUrl>Imagenes/espania.png</ImageUrl>
34         <NavigateUrl>
35             http://www.cia.gov/cia/publications/factbook/geos/sp.html
36         </NavigateUrl>

```

Figura 21.19 | Archivo que contiene información sobre anuncios, para utilizarla en un ejemplo del control `AdRotator`. (Parte I de 2).

```

37     <AlternateText>Información de España</AlternateText>
38     <Impressions>1</Impressions>
39 </Ad>
40
41 <Ad>
42     <ImageUrl>Imagenes/latvia.png</ImageUrl>
43     <NavigateUrl>
44         http://www.cia.gov/cia/publications/factbook/geos/lg.html
45     </NavigateUrl>
46     <AlternateText>Información de Latvia</AlternateText>
47     <Impressions>1</Impressions>
48 </Ad>
49
50 <Ad>
51     <ImageUrl>Imagenes/peru.png</ImageUrl>
52     <NavigateUrl>
53         http://www.cia.gov/cia/publications/factbook/geos/pe.html
54     </NavigateUrl>
55     <AlternateText>Información de Perú</AlternateText>
56     <Impressions>1</Impressions>
57 </Ad>
58
59 <Ad>
60     <ImageUrl>Imagenes/senegal.png</ImageUrl>
61     <NavigateUrl>
62         http://www.cia.gov/cia/publications/factbook/geos/sg.html
63     </NavigateUrl>
64     <AlternateText>Información de Senegal</AlternateText>
65     <Impressions>1</Impressions>
66 </Ad>
67
68 <Ad>
69     <ImageUrl>Imagenes/suecia.png</ImageUrl>
70     <NavigateUrl>
71         http://www.cia.gov/cia/publications/factbook/geos/sw.html
72     </NavigateUrl>
73     <AlternateText>Información de Suecia</AlternateText>
74     <Impressions>1</Impressions>
75 </Ad>
76
77 <Ad>
78     <ImageUrl>Imagenes/tailandia.png</ImageUrl>
79     <NavigateUrl>
80         http://www.cia.gov/cia/publications/factbook/geos/th.html
81     </NavigateUrl>
82     <AlternateText>Información de Tailandia</AlternateText>
83     <Impressions>1</Impressions>
84 </Ad>
85
86 <Ad>
87     <ImageUrl>Imagenes/estadosunidos.png</ImageUrl>
88     <NavigateUrl>
89         http://www.cia.gov/cia/publications/factbook/geos/us.html
90     </NavigateUrl>
91     <AlternateText>Información de Estados Unidos</AlternateText>
92     <Impressions>1</Impressions>
93 </Ad>
94 </Advertisements>

```

Figura 21.19 | Archivo que contiene información sobre anuncios, para utilizarla en un ejemplo del control AdRotator. (Parte 2 de 2).

El elemento **AlternateText** (línea 10) anidado en cada elemento **Ad** contiene texto que se muestra en lugar de la imagen, cuando el explorador no puede localizar o representar la imagen por alguna razón (por ejemplo, si el archivo no existe, o si el explorador no es capaz de mostrarla). El texto del elemento **AlternateText** es también un cuadro de información sobre la herramienta (tooltip), que Internet Explorer muestra cuando un usuario coloca el puntero del ratón sobre la imagen (figura 21.18). El elemento **Impressions** (línea 56) especifica con qué frecuencia debe aparecer una imagen específica, en relación con las demás imágenes. Un anuncio que tenga un valor más alto en **Impressions** se mostrará con más frecuencia que un anuncio con un valor menor. En nuestro ejemplo, los anuncios se muestran con la misma probabilidad, ya que el valor de cada elemento **Impressions** es 1.

21.5.3 Controles de validación

Esta sección introduce un tipo distinto de control Web, conocido como *control de validación* (o *validador*), el cual determina si los datos en otro control Web tienen el formato apropiado. Por ejemplo, los validadores podrían determinar si un usuario ha proporcionado información en un campo requerido, o si un campo de código postal contiene exactamente cinco dígitos. Los validadores proporcionan un mecanismo para validar la entrada del usuario en el explorador cliente. Cuando se crea el XHTML para nuestra página, el validador se convierte en *ECMAScript*,¹ que realiza la validación. ECMAScript es un lenguaje de secuencias de comandos que mejora la funcionalidad y apariencia de las páginas Web. Por lo general, ECMAScript se ejecuta en el cliente. Algunos clientes no tienen soporte para secuencias de comandos o lo deshabilitan. No obstante, por cuestiones de seguridad, siempre es preferible realizar la validación en el servidor, se ejecute o no la secuencia de comandos en el cliente.

Validar la entrada en un formulario Web Forms

El ejemplo en esta sección pide al usuario que introduzca un nombre, dirección de correo electrónico y número telefónico. Un sitio Web podría utilizar un formulario como este para recolectar la información de contacto de sus visitantes. Después de que el usuario escribe datos, pero antes de que éstos se envíen al servidor Web, los validadores se aseguran que el usuario haya introducido un valor en cada campo, y que los valores de dirección de correo electrónico y número telefónico tengan un formato aceptable. En este ejemplo, (555) 123-4567, 555-123-4567 y 123-4567 se consideran todos números telefónicos válidos. Una vez que se envían los datos, el servidor Web responde mostrando un mensaje apropiado y una tabla de XHTML en la que se repite la información enviada. Por lo general, una aplicación comercial real almacena los datos enviados en una base de datos, o en un archivo en el servidor. Nosotros sólo enviaremos los datos de vuelta al formulario, para demostrar que el servidor los recibió.

La figura 21.20 presenta el archivo ASPX. Al igual que el formulario Web Forms en la figura 21.16, este formulario Web Forms utiliza un elemento **table** para organizar el contenido de la página. Las líneas 24-25, 36-37 y 56-57 definen controles **TextBox** para recuperar el nombre del usuario, la dirección de correo electrónico y el número telefónico, respectivamente, y la línea 75 define un botón **Enviar**. Las líneas 77-79 crean un control **Label** llamado **salidaLabel**, que muestra la respuesta del servidor cuando el usuario envía con éxito el formulario. Observe

```

1 <!-- Fig. 21.20: Validacion.aspx -->
2 <!-- Formulario que demuestra el uso de validadores para validar la entrada del
   usuario. -->
3 <%@ Page Language="C#" AutoEventWireup="true"
4     CodeFile="Validacion.aspx.cs" Inherits="Validacion" %>
5
6 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
```

Figura 21.20 | Validadores utilizados en un formulario Web Forms que recupera la información de contacto de un usuario. (Parte 1 de 4).

1. ECMAScript (comúnmente conocido como JavaScript) es un estándar de secuencias de comandos desarrollado por ECMA Internacional. Los lenguajes JavaScript de Netscape y JScript de Microsoft implementan el estándar ECMAScript, pero cada uno cuenta con características adicionales que están más allá de la especificación. Para obtener información acerca del estándar ECMAScript actual, visite www.ecma-international.org/publications/standards/Ecma-262.htm. En el sitio www.mozilla.org/js encontrará información acerca de JavaScript, y en msdn.microsoft.com/library/en-us/script56/html/1e9b3876-3d38-4fd8-8596-1bbfe2330aa9.asp encontrará información acerca de JScript.

```

7      "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
8
9      <html xmlns="http://www.w3.org/1999/xhtml" >
10     <head id="Head1" runat="server">
11         <title>Demostración de los controles de validación</title>
12     </head>
13     <body>
14         <form id="form1" runat="server">
15             <div>
16                 Por favor llene el siguiente formulario.<br />
17                 <em>Todos los campos son obligatorios y deben
18                 contener información válida.</em><br />
19             <br />
20             <table>
21                 <tr>
22                     <td style="width: 134px" valign="top">Nombre:</td>
23                     <td style="width: 450px" valign="top">
24                         <asp:TextBox ID="nombreTextBox" runat="server">
25                         </asp:TextBox><br />
26                         <asp:RequiredFieldValidator ID="nombreInputValidator"
27                         runat="server" ControlToValidate="nombreTextBox"
28                         ErrorMessage="Por favor escriba su nombre."
29                         Display="Dynamic"></asp:RequiredFieldValidator>
30                     </td>
31                 </tr>
32                 <tr>
33                     <td style="width: 134px; height: 64px;" valign="top">
34                         Dirección de E-mail:</td>
35                     <td style="width: 450px; height: 64px;" valign="top">
36                         <asp:TextBox ID="emailTextBox" runat="server">
37                         </asp:TextBox>
38                         &nbsp;ejemplo: usuario@dominio.com<br />
39                         <asp:RequiredFieldValidator ID="emailInputValidator"
40                         runat="server" ControlToValidate="emailTextBox"
41                         ErrorMessage="Por favor escriba su dirección de correo
42                         electrónico."
43                         Display="Dynamic"></asp:RequiredFieldValidator>
44                         <asp:RegularExpressionValidator
45                         ID="emailFormatValidator" runat="server"
46                         ControlToValidate="emailTextBox"
47                         ErrorMessage="Por favor escriba una dirección de correo en un
48                         formato válido." Display="Dynamic"
49                         ValidationExpression=
50                         "\w+([-+.']\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*">
51                         </asp:RegularExpressionValidator>
52                     </td>
53                 </tr>
54                 <tr>
55                     <td style="width: 134px" valign="top">Número telefónico:</td>
56                     <td style="width: 450px" valign="top">
57                         <asp:TextBox ID="telefonoTextBox" runat="server">
58                         </asp:TextBox>
59                         &nbsp;ejemplo: (555) 555-1234<br />
60                         <asp:RequiredFieldValidator ID="telefonoInputValidator"
61                         runat="server" ControlToValidate="telefonoTextBox"
62                         ErrorMessage="Por favor escriba su número telefónico."
63                         Display="Dynamic"></asp:RequiredFieldValidator>
64                         <asp:RegularExpressionValidator

```

Figura 21.20 | Validadores utilizados en un formulario Web Forms que recupera la información de contacto de un usuario. (Parte 2 de 4).

```

64         ID="telefonoFormatValidator" runat="server"
65         ControlToValidate="telefonoTextBox"
66         ErrorMessage="Por favor escriba un número telefónico en un
67         formato válido." Display="Dynamic"
68         ValidationExpression=
69         "((\d{3}\d{3}|\d{3}\d{3}-)\d{3})-\d{4}"
70     </asp:RegularExpressionValidator>
71 </td>
72 </tr>
73 </table>
74 <br />
75 <asp:Button ID="enviarButton" runat="server" Text="Enviar" />
76 <br /><br />
77 <asp:Label ID="salidaLabel" runat="server"
78     Text="Gracias por llenar el formulario y enviarlo."
79     Visible="False"></asp:Label>
80 </div>
81 </form>
82 </body>
83 </html>

```

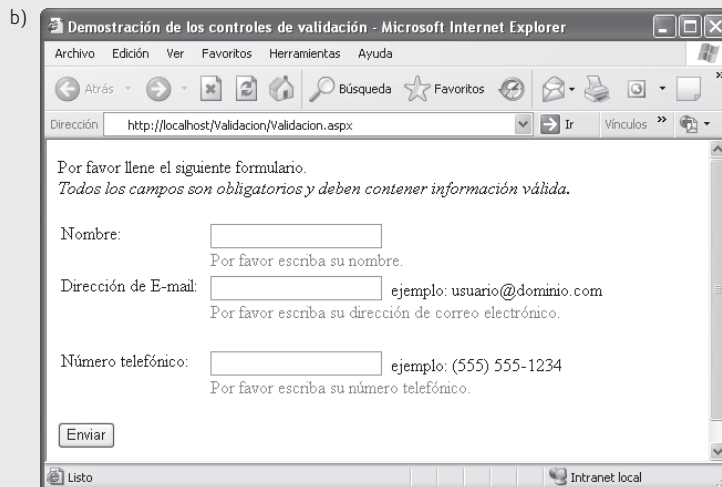
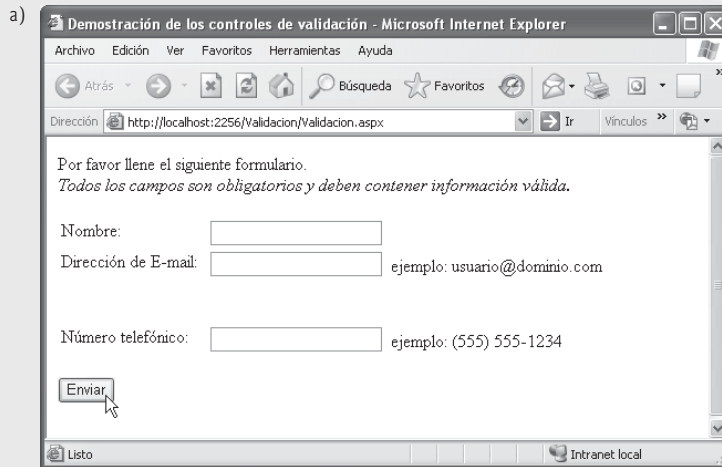


Figura 21.20 | Validadores utilizados en un formulario Web Forms que recupera la información de contacto de un usuario. (Parte 3 de 4).

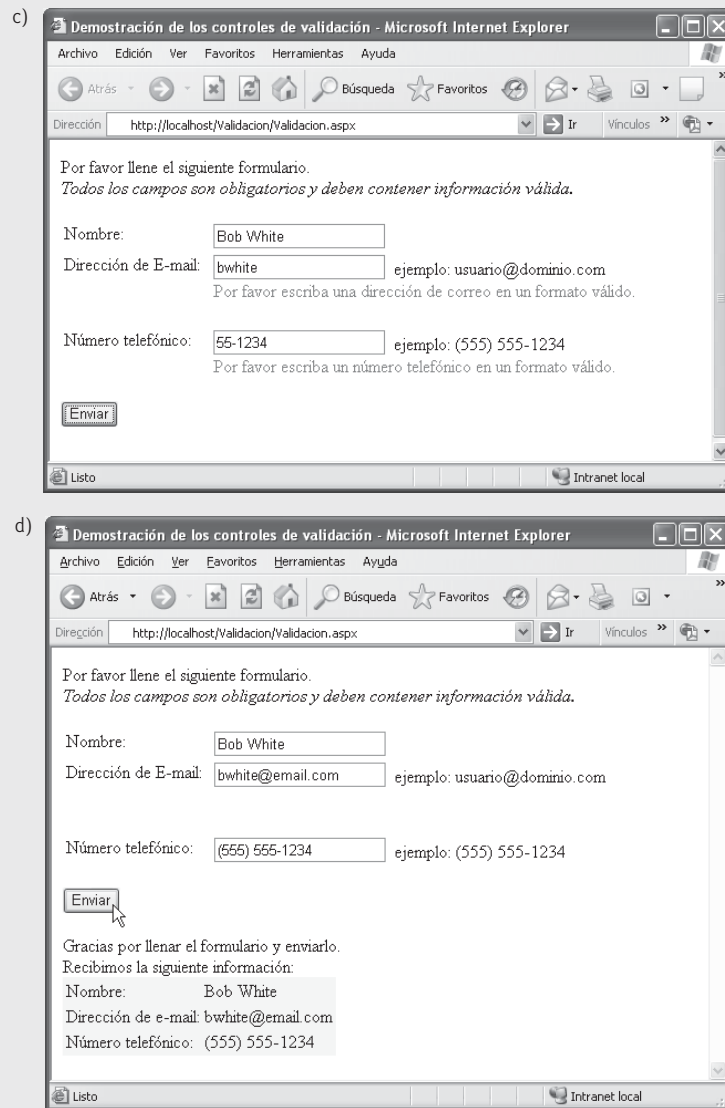


Figura 21.20 | Validadores utilizados en un formulario Web Forms que recupera la información de contacto de un usuario. (Parte 4 de 4).

que en un principio la propiedad **Visible** de `salidaLabel` se establece en `False`, para que el control `Label` no aparezca en el explorador del cliente cuando la página se carga por primera vez.

Uso de los controles `RequiredFieldValidator`

En este ejemplo utilizamos tres controles **`RequiredFieldValidator`** (se encuentran en la sección **Validación** del **Cuadro de herramientas**) para asegurar que los controles `TextBox` del nombre, la dirección de correo electrónico y el número telefónico no estén vacíos cuando se envíe el formulario. Un control `RequiredFieldValidator` convierte un control de entrada en un campo requerido. Si dicho campo está vacío, la validación falla. Por ejemplo, las líneas 26-29 definen el control `RequiredFieldValidator` llamado `nombreInputValidator`, el cual confirma que `nombreTextBox` no esté vacío. La línea 27 asocia a `nombreTextBox` con `nombreInputValidator` estableciendo la propiedad **`ControlToValidate`** del validador a `nombreTextBox`. Esto indica que `nombreInputValidator` verifica el contenido de `nombreTextBox`. El texto de la propiedad **`ErrorMessage`** (línea 28) se muestra en el formulario Web

Forms si falla la validación. Si el usuario no introduce datos en `nombreTextBox` y trata de enviar el formulario, el texto de `ErrorMessage` se muestra en rojo. Como establecimos la propiedad **Display** del control en `Dynamic` (línea 29), el validador ocupa espacio en el formulario Web Forms sólo cuando falla la validación; el espacio se asigna en forma dinámica cuando falla la validación, lo cual provoca que los controles que están debajo del validador se desplacen hacia abajo para dar cabida a `ErrorMessage`, como se puede ver en las figuras de la 21.20(a) a la 21.20(c).

Uso de controles *RegularExpressionValidator*

Este ejemplo utiliza también controles *RegularExpressionValidator* para comparar la dirección de correo electrónico y el número telefónico escritos por el usuario con expresiones regulares. (En el capítulo 16 se introdujeron las expresiones regulares.) Estos controles determinan si la dirección de correo electrónico y el número telefónico se introdujeron en un formato válido. Por ejemplo, las líneas 43-50 crean un control *RegularExpressionValidator* llamado `emailFormatValidator`. La línea 45 establece la propiedad `ControlToValidate` a `emailTextBox`, para indicar que `emailFormatValidator` verifica el contenido de `emailTextBox`.

La propiedad **ValidationExpression** de un control *RegularExpressionValidator* especifica la expresión regular que valida el contenido de `ControlToValidate`. Al hacer clic en la elipsis que está a un lado de la propiedad `ValidationExpression` en la ventana **Propiedades** se muestra el cuadro de diálogo **Editor de expresiones regulares**, el cual contiene una lista de **Expresiones estándar** para números telefónicos, códigos postales y otra información con formato. Usted también puede escribir su propia expresión personalizada. Para el control `emailFormatValidator`, elegimos la expresión estándar **Dirección de correo electrónico de Internet**, que utiliza la siguiente expresión de validación:

```
\w+([-+.' ]\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*
```

Esta expresión regular indica que una dirección de correo electrónico es válida si la parte de la dirección antes del símbolo @ contiene uno o más caracteres de palabra (es decir, caracteres alfanuméricos o guiones bajos), seguida de cero o más cadenas compuestas por un guión corto, signo de suma, punto o apóstrofe, y caracteres de palabra adicionales. Después del símbolo @, una dirección de correo electrónico válida debe contener uno o más grupos de caracteres de palabra, posiblemente separados por guiones cortos o puntos, seguidos de un punto obligatorio y otro grupo de uno o más caracteres de palabra, posiblemente separados por guiones cortos o puntos. Por ejemplo, `bob.white@email.com`, `bob.white@mi-email.com` y `bob-correo.personal@white.email.com` son todas direcciones de correo electrónico válidas. Si el usuario introduce texto en el control `emailTextBox` que no tenga el formato correcto, y hace clic en un cuadro de texto distinto o trata de enviar el formulario, el texto de `ErrorMessage` se mostrará en rojo.

También usamos el control *RegularExpressionValidator* llamado `telefonoFormatValidator` (líneas 63-70) para asegurarnos que el control `telefonoTextBox` contenga un número telefónico válido, antes de enviar el formulario. En el cuadro de diálogo **Editor de expresiones regulares**, seleccionamos **Número de teléfono en EE.UU.**, que asigna la expresión de validación

```
((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}
```

a la propiedad `ValidationExpression`. Esta expresión indica que un número telefónico puede contener un código de área de tres dígitos, ya sea entre paréntesis y seguido por un espacio opcional, o sin paréntesis y seguido por un guión corto requerido. Después de un código de área opcional, un teléfono debe contener tres dígitos, un guión corto y otros cuatro dígitos. Por ejemplo, (555) 123-4567, 555-123-4567 y 123-4567 son todos números telefónicos válidos.

Si los cinco validadores tienen éxito (es decir, que se llene cada uno de los controles `TextBox`, y que la dirección de correo electrónico y el número telefónico que se proporcionan sean válidos), al hacer clic en el botón **Enviar** se envían los datos del formulario al servidor. Como se muestra en la figura 21.20(d), el servidor responde mostrando los datos enviados en el control `salidaLabel` (líneas 77-79).

Análisis del archivo de código subyacente para un formulario Web Forms que recibe entrada del usuario

La figura 21.21 muestra el archivo de código subyacente para el archivo ASPX en la figura 21.20. Observe que este archivo de código subyacente no contiene ninguna implementación relacionada con los validadores. Pronto hablaremos más sobre esto.

```

1  // Fig. 21.21: Validacion.aspx.cs
2  // Archivo de código subyacente para el formulario que demuestra los controles de
   validación.
3  using System;
4  using System.Data;
5  using System.Configuration;
6  using System.Web;
7  using System.Web.Security;
8  using System.Web.UI;
9  using System.Web.UI.WebControls;
10 using System.Web.UI.WebControls.WebParts;
11 using System.Web.UI.HtmlControls;
12
13 public partial class Validacion : System.Web.UI.Page
14 {
15     // el manejador de eventos Page_Load se ejecuta cuando se carga la página
16     protected void Page_Load( object sender, EventArgs e )
17     {
18         // si no es la primera vez que se carga la página
19         // (es decir, si el usuario ya envió los datos del formulario)
20         if ( IsPostBack )
21         {
22             // recupera los valores enviados por el usuario
23             string nombre = Request.Form[ "nombreTextBox" ];
24             string email = Request.Form[ "emailTextBox" ];
25             string telefono = Request.Form[ "telefonoTextBox" ];
26
27             // crea una tabla, indicando los valores enviados
28             salidaLabel.Text +=
29                 "<br />Recibimos la siguiente información:" +
30                 "<table style=\"background-color: yellow\">" +
31                 "<tr><td>Nombre: </td><td>" + nombre + "</td></tr>" +
32                 "<tr><td>Dirección de e-mail: </td><td>" + email + "</td></tr>" +
33                 "<tr><td>Número telefónico: </td><td>" + telefono + "</td></tr>" +
34                 "<table>";
35
36             salidaLabel.Visible = true; // muestra el mensaje de salida
37         } // fin de if
38     } // fin del método Page_Load
39 } // fin de la clase Validation

```

Figura 21.21 | Código subyacente para un formulario Web Forms que obtiene la información de contacto de un usuario.

A menudo, los programadores de sitios Web que utilizan ASP.NET diseñan sus páginas Web de tal forma que la página actual se actualice cuando el usuario envía el formulario; esto permite al programa recibir la entrada, procesarla según sea necesario y mostrar los resultados en la misma página, cuando se carga la segunda vez. Por lo general, estas páginas contienen un formulario que, cuando se envía, envía los valores de todos los controles al servidor y hace que la página actual se solicite de nuevo. Este evento se conoce como *postback*. La línea 20 utiliza la propiedad **IsPostBack** de la clase Page para determinar si la página se cargará debido a un postback. La primera vez que se solicita la página Web, IsPostBack es false y la página sólo muestra el formulario para que el usuario introduzca datos. Cuando ocurre el postback (debido a que el usuario hizo clic en Enviar), IsPostBack cambia a true.

Las líneas 23-25 utilizan el objeto **Request** para recuperar los valores de nombreTextBox, emailTextBox y telefonoTextBox del objeto NameValueCollection **Form**. Cuando se publican datos al servidor Web, los datos del formulario de XHTML están accesibles para la aplicación Web a través del arreglo Form del objeto Request. Las líneas 28-34 adjuntan al valor de Text de salidaLabel un salto de línea, un mensaje adicional y una tabla de XHTML que contiene los datos enviados, de manera que el usuario sepa que el servidor recibió los datos en forma correcta. En una aplicación empresarial real, los datos se almacenarían en una base de datos o en un archivo, en

este punto de la aplicación. La línea 36 establece la propiedad `Visible` de `salidaLabel` a `true`, de manera que el usuario pueda ver el mensaje de agradecimiento y los datos que envió.

Análisis del XHTML del lado cliente para un formulario Web Forms con validación

La figura 21.22 muestra el XHTML y el ECMAScript que se envían al explorador Web cuando se carga `Validacion.aspx` después del evento `postback`. Para ver este código, seleccione **Ver > Código fuente** en Internet Explorer. Las líneas 25-36, 100-171 y 180-218 contienen el ECMAScript que proporciona la implementación para los controles de validación y para realizar el `postback`. ASP.NET genera este código ECMAScript. Usted no necesita crear ni tampoco entender el código ECMAScript; la funcionalidad definida para los controles en nuestra aplicación se convierte en ECMAScript funcional para nosotros.

En los archivos ASPX anteriores, establecimos en forma explícita el atributo `EnableViewState` de cada control Web a `false`. Este atributo determina si el valor de un control Web persiste (es decir, se retiene) cuando ocurre un `postback`. Este atributo es `true` de manera predeterminada, lo cual indica que el valor del control persiste. Observe en la figura 21.20(d) que los valores introducidos por el usuario siguen apareciendo en los cuadros de texto una vez que ocurre el `postback`. Una entrada `hidden` en el documento de XHTML (líneas 14-22 de la figura 21.22) contiene los datos de los controles en esta página. Este elemento siempre se llama `__VIEWSTATE` y almacena los datos de los controles en forma de una cadena codificada.



Tip de rendimiento 21.2

Al establecer `EnableViewState` a `false` se reduce la cantidad de datos que se pasan al servidor Web en cada solicitud.

```

1  <!-- Fig. 21.22: Validacion.html -->
2  <!-- El XHTML y ECMAScript generados para Validacion.aspx -->
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
4    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
5
6  <html xmlns="http://www.w3.org/1999/xhtml" >
7    <head>
8      <title>Demostración de los controles de validación</title>
9    </head>
10   <body>
11     <form method="post" action="Validacion.aspx"
12       onsubmit="javascript:return WebForm_OnSubmit();" id="form1">
13       <div>
14         <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
15           value="/wEPDwUJODcSMTEzMzA4D2QWAgIDD2QWAgITDw8WBB4EVGV4dAwWAKdyYW
16           NpYXMGcG9yIGxsZW5hcjB1bCBmb3JtdWxhcm1vIHkgZW52aWYyY2NpZ3J1bG9yIC8+UmV
17           jZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmVjZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmV
18           jZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmVjZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmV
19           jZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmVjZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmV
20           jZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmVjZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmV
21           jZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmVjZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmV
22           jZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmVjZW50ZSBBbmZvcmlhY2NpZ3J1bG9yIC8+UmV
23       </div>
24
25       <script src="/Validacion/WebResource.axd?d=L66EICyWjSFEYAKozJKp
26         XBboS9ZjYZ1S1S4TtTK3Pes1&t=632973466940620000"
27         type="text/javascript"></script>
28
29       <script type="text/javascript">
30         <!--
31         function WebForm_OnSubmit() {

```

Figura 21.22 | XHTML y ECMAScript generados por ASP.NET, que se envían al explorador cuando se solicita el archivo `Validacion.aspx`. (Parte I de 5).

```

32         if (typeof(ValidatorOnSubmit) == false) return false;
33         return true;
34     }
35     // -->
36 </script>
37
38 <div>
39     Por favor llene el siguiente formulario.<br />
40     <em>Todos los campos son obligatorios y deben
41     contener información válida.</em><br />
42 <br />
43 <table>
44     <tr>
45         <td style="width: 134px" valign="top">Nombre:</td>
46         <td style="width: 450px" valign="top">
47             <input name="nombreTextBox" type="text"
48             id="nombreTextBox" /><br />
49             <span id="nombreInputValidator" style="color:Red;
50             display:none;">Por favor escriba su nombre.</span>
51         </td>
52     </tr>
53     <tr>
54         <td style="width: 134px; valign="top">
55             Dirección de E-mail:</td>
56         <td style="width: 450px; valign="top">
57             <input name="emailTextBox" type="text"
58             id="emailTextBox" />
59             &nbsp;&nbsp;&nbsp;ejemplo: usuario@dominio.com<br />
60             <span id="emailInputValidator" style="color:Red;
61             display:none;">Por favor escriba su dirección de correo
62             electrónico.
63             </span>
64             <span id="emailFormatValidator" style="color:Red;
65             display:none;">Por favor escriba una dirección de correo en un
66             formato válido.</span>
67         </td>
68     </tr>
69     <tr>
70         <td style="width: 134px" valign="top">Número telefónico:</td>
71         <td style="width: 450px" valign="top">
72             <input name="telefonoTextBox" type="text"
73             id="telefonoTextBox" />
74             &nbsp;&nbsp;&nbsp;ejemplo: (555) 555-1234<br />
75             <span id="telefonoInputValidator" style="color:Red;
76             display:none;">Por favor escriba su número telefónico.
77             </span>
78             <span id="telefonoFormatValidator" style="color:Red;
79             display:none;">Por favor escriba un número telefónico en un
80             formato válido.</span>
81         </td>
82     </tr>
83 </table>
84 <br />
85 <input type="submit" name="enviarButton" value="Enviar"
86     onclick="javascript:WebForm_DoPostBackWithOptions(
87     new WebForm_PostBackOptions(&quot;enviarButton&quot;,
88     &quot;&quot;, true, &quot;&quot;, &quot;&quot;,

```

Figura 21.22 | XHTML y ECMAScript generados por ASP.NET, que se envían al explorador cuando se solicita el archivo Validacion.aspx. (Parte 2 de 5).


```

88         false, false))" id="enviarButton" />
89     <br /><br />
90     <span id="salidaLabel">Gracias por llenar el formulario y enviarlo.<br />
91     Recibimos la siguiente información:
92     <table style="background-color: yellow">
93     <tr><td>Nombre: </td><td>Bob</td></tr>
94     <tr><td>Dirección de e-mail: </td><td>bwhite@email.com</td></tr>
95     <tr><td>Número telefónico: </td><td>(555) 555-1234</td></tr>
96     <table>
97     </span>
98 </div>
99
100 <script type="text/javascript">
101 <!--
102     var Page_Validators = new Array(
103         document.getElementById("nombreInputValidator"),
104         document.getElementById("emailInputValidator"),
105         document.getElementById("emailFormatValidator"),
106         document.getElementById("telefonoInputValidator"),
107         document.getElementById("telefonoFormatValidator"));
108 // -->
109 </script>
110
111 <script type="text/javascript">
112 <!--
113     var nombreInputValidator =
114         document.all ? document.all["nombreInputValidator"] :
115         document.getElementById("nombreInputValidator");
116     nombreInputValidator.controltovalidate = "nombreTextBox";
117     nombreInputValidator.errorMessage = "Por favor escriba su nombre.";
118     nombreInputValidator.display = "Dynamic";
119     nombreInputValidator.evaluationfunction =
120         "RequiredFieldValidatorEvaluateIsValid";
121     nombreInputValidator.initialvalue = "";
122
123     var emailInputValidator =
124         document.all ? document.all["emailInputValidator"] :
125         document.getElementById("emailInputValidator");
126     emailInputValidator.controltovalidate = "emailTextBox";
127     emailInputValidator.errorMessage =
128         "Por favor escriba su dirección de correo electrónico.";
129     emailInputValidator.display = "Dynamic";
130     emailInputValidator.evaluationfunction =
131         "RequiredFieldValidatorEvaluateIsValid";
132     emailInputValidator.initialvalue = "";
133
134     var emailFormatValidator =
135         document.all ? document.all["emailFormatValidator"] :
136         document.getElementById("emailFormatValidator");
137     emailFormatValidator.controltovalidate = "emailTextBox";
138     emailFormatValidator.errorMessage =
139         "Por favor escriba una dirección de correo en un \r\n
140         "         formato válido.";
141     emailFormatValidator.display = "Dynamic";
142     emailFormatValidator.evaluationfunction =
143         "RegularExpressionValidatorEvaluateIsValid";
144     emailFormatValidator.validationexpression =

```

Figura 21.22 | XHTML y ECMAScript generados por ASP.NET, que se envían al explorador cuando se solicita el archivo Validacion.aspx. (Parte 3 de 5).

```

145         "\\w+([-+.'\\w+)*@\\w+([-.'\\w+)*\\.\\w+([-.'\\w+)*";
146
147         var telefonoInputValidator =
148             document.all ? document.all["telefonoInputValidator"] :
149             document.getElementById("telefonoInputValidator");
150         telefonoInputValidator.controltovalidate = "telefonoTextBox";
151         telefonoInputValidator.errormessage =
152             "Por favor escriba su número telefónico.";
153         telefonoInputValidator.display = "Dynamic";
154         telefonoInputValidator.evaluationfunction =
155             "RequiredFieldValidatorEvaluateIsValid";
156         telefonoInputValidator.initialvalue = "";
157
158         var telefonoFormatValidator =
159             document.all ? document.all["telefonoFormatValidator"] :
160             document.getElementById("telefonoFormatValidator");
161         telefonoFormatValidator.controltovalidate = "telefonoTextBox";
162         telefonoFormatValidator.errormessage =
163             "Por favor escriba un número telefónico en un \r\n      " +
164             "      formato válido.";
165         telefonoFormatValidator.display = "Dynamic";
166         telefonoFormatValidator.evaluationfunction =
167             "RegularExpressionValidatorEvaluateIsValid";
168         telefonoFormatValidator.validationexpression =
169             "((\\(\\d{3}\\) ?)|(\\d{3}-)?)\\d{3}-\\d{4}";
170     // -->
171 </script>
172
173 <div>
174     <input type="hidden" name="__EVENTTARGET"
175         id="__EVENTTARGET" value="" />
176     <input type="hidden" name="__EVENTARGUMENT"
177         id="__EVENTARGUMENT" value="" />
178 </div>
179
180 <script type="text/javascript">
181 <!--
182     var theForm = document.forms['form1'];
183
184     if (!theForm) {
185         theForm = document.form1;
186     }
187
188     function __doPostBack(eventTarget, eventArgument) {
189         if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
190             theForm.__EVENTTARGET.value = eventTarget;
191             theForm.__EVENTARGUMENT.value = eventArgument;
192             theForm.submit();
193         }
194     }
195 // -->
196 </script>
197
198 <script src="/Validacion/WebResource.axd?d=Ev2nWN869PXiNuJaCgEHp
199     A2&t=632973466940620000" type="text/javascript"></script>
200
201 <script type="text/javascript">

```

Figura 21.22 | XHTML y ECMAScript generados por ASP.NET, que se envían al explorador cuando se solicita el archivo Validacion.aspx. (Parte 4 de 5).

```

202      <!--
203      var Page_ValidationActive = false;
204
205      if (typeof(ValidatorOnLoad) == "function") {
206          ValidatorOnLoad();
207      }
208
209      function ValidatorOnSubmit() {
210          if (Page_ValidationActive) {
211              return ValidatorCommonOnSubmit();
212          }
213          else {
214              return true;
215          }
216      }
217      // -->
218      </script>
219      </form>
220  </body>
221 </html>

```

Figura 21.22 | XHTML y ECMAScript generados por ASP.NET, que se envían al explorador cuando se solicita el archivo Validacion.aspx. (Parte 5 de 5).

21.6 Rastreo de sesiones

En un principio, los críticos acusaron a Internet y al comercio electrónico de no proporcionar el tipo de servicio personalizado que se experimenta por lo general en las tiendas convencionales. Para tratar este problema, los comercios electrónicos empezaron a establecer mecanismos mediante los cuales pudieran personalizar las experiencias de navegación de los usuarios, ajustando el contenido para cada usuario y permitiéndoles, al mismo tiempo, evitar información irrelevante. Las empresas logran este nivel de servicio mediante el rastreo de los movimientos de cada cliente a través de Internet, y combinan los datos recolectados con la información que proporciona el consumidor, incluyendo información de facturación, preferencias personales, intereses y pasatiempos.

Personalización

La *personalización* hace posible para los comercios electrónicos comunicarse en forma efectiva con sus clientes, y también mejora la habilidad del usuario para localizar los productos y servicios que desea. Las compañías que proporcionan contenido de interés particular para los usuarios pueden establecer relaciones con los clientes y mejorar esas relaciones con el tiempo. Lo que es más, al dirigirse a los clientes con ofertas, recomendaciones, anuncios, promociones y servicios personales, los comercios electrónicos crean la lealtad del cliente. Los sitios Web pueden utilizar tecnología sofisticada para permitir a los visitantes personalizar sus páginas iniciales, para que se adapten a sus necesidades y preferencias individuales. De manera similar, los negocios de compras en línea a menudo almacenan información personal de los clientes, y con base en ello ajustan las notificaciones y ofertas especiales a sus intereses. Dichos servicios alientan a los clientes a visitar los sitios con más frecuencia, y a realizar compras con más regularidad.

Privacidad

No obstante, existe un sacrificio entre el servicio personalizado de los comercios electrónicos y la protección de la privacidad. Algunos clientes adoptan la idea del contenido personalizado, pero otros temen a las posibles consecuencias adversas si la información que proporcionan a los comercios electrónicos se libera o recolecta mediante tecnologías de rastreo. Los consumidores y los defensores de la privacidad se preguntan: ¿qué pasa si el comercio electrónico al que le proporcionamos datos personales vende o da esa información a otra organización, sin nuestro conocimiento? ¿Qué tal si no queremos que partes desconocidas rastreen nuestras acciones en Internet (un medio supuestamente anónimo)? ¿Qué pasa si partes no autorizadas obtienen acceso a datos privados delicados,

como los números de tarjetas de crédito o los historiales médicos? Los programadores, consumidores, comercios electrónicos y legisladores deben, por igual, debatir y tratar todas estas preguntas.

Reconocer clientes

Para proporcionar servicios personalizados a los consumidores, los comercios electrónicos deben tener la capacidad de reconocer a los clientes, cuando soliciten información de un sitio. Como hemos visto, HTTP facilita el sistema de solicitud/respuesta en el que opera la Web. Por desgracia, HTTP es un protocolo que carece de estado; no soporta conexiones persistentes que permitan a los servidores Web mantener información de estado en relación con clientes específicos. Esto significa que los servidores Web no pueden determinar si una solicitud viene de un cliente específico, o si el mismo o varios clientes generan una serie de solicitudes. Para sortear este problema, los sitios pueden proporcionar mecanismos mediante los cuales puedan identificar a los clientes individuales. Una sesión representa a un único cliente en un sitio Web. Si el cliente sale del sitio y regresa después, de todas formas se le reconocerá como el mismo usuario. Para ayudar al servidor a diferenciar los diversos clientes, cada cliente debe identificarse con el servidor. El rastreo de clientes individuales, conocido como *rastreo de sesiones*, puede realizarse de varias formas. Una técnica popular utiliza cookies (sección 21.6.1); otra utiliza el objeto `HttpSessionState` de ASP.NET (sección 21.6.2). Las demás técnicas de rastreo de sesiones incluyen el uso de elementos `input` de tipo oculto ("`hidden`") en el formulario, y reescritura de URLs. Mediante el uso de elementos "`hidden`", un formulario Web Forms puede escribir datos de rastreo de sesiones en un elemento `form` de la página Web que devuelve al cliente, en respuesta a una solicitud previa. Cuando el usuario envía el formulario en la nueva página Web, todos los datos (incluyendo los campos "`hidden`") se envían al manejador de formularios en el servidor Web. Cuando un sitio Web lleva a cabo la reescritura de URLs, el formulario Web Forms incrusta la información de rastreo de sesiones directamente en los URLs de los hipervínculos en los que el usuario hace clic, para enviar las solicitudes subsiguientes al servidor Web.

Observe que en nuestros ejemplos anteriores se estableció la propiedad `EnableSessionState` del formulario Web Forms a `false`. No obstante, y como deseamos utilizar el rastreo de sesiones en los siguientes ejemplos, mantendremos la configuración predeterminada de esta propiedad: `True`.

21.6.1 Cookies

Las *cookies* proporcionan a los desarrolladores Web una herramienta para personalizar páginas Web. Una cookie es una pieza de datos almacenados en un pequeño archivo, en la computadora del usuario. Una cookie mantiene información acerca del cliente durante, y entre, las sesiones en el explorador. La primera vez que un usuario visita un sitio Web, su computadora podría recibir una cookie; después, esta cookie se reactiva cada vez que el usuario vuelve a visitar ese sitio. La información recolectada tiene el propósito de ser un registro anónimo con datos que se utilizan para personalizar las futuras visitas que haga el usuario al sitio. Cuando un usuario agrega artículos a un carrito de compras en línea, o cuando realiza otra tarea que resulta en una solicitud al servidor Web, éste recibe una cookie que contiene el identificador único del usuario. Luego, el servidor utiliza ese identificador único para localizar el carrito de compras y realiza cualquier procesamiento requerido.

Además de identificar a los usuarios, las cookies también pueden indicar las preferencias de compra de los clientes. Cuando un formulario Web Forms recibe una solicitud de un cliente, puede examinar la(s) cookie(s) que envió al cliente durante las sesiones de comunicación anteriores, para identificar las preferencias del cliente y mostrar de inmediato productos de su interés.

Cada interacción basada en HTTP entre un cliente y un servidor incluye un encabezado, el cual contiene información acerca de la solicitud (cuando la comunicación es del cliente al servidor) o acerca de la respuesta (cuando la comunicación es del servidor al cliente). Cuando un formulario Web Forms recibe una solicitud, el encabezado incluye información como el tipo de solicitud (por ejemplo `Get`) y cualquier cookie que se haya enviado previamente del servidor para almacenarla en el equipo cliente. Cuando el servidor formula su respuesta, la información del encabezado contiene todas las cookies que el servidor desea almacenar en el equipo cliente y demás información, como el tipo MIME de la respuesta.

La *fecha de expiración* de una cookie determina cuánto tiempo permanece en la computadora del cliente. Si no se establece una fecha de expiración para una cookie, el explorador Web mantiene esa cookie el tiempo que dure la sesión de navegación. En caso contrario, el explorador Web mantiene la cookie hasta que llega la fecha de expiración. Cuando el explorador solicita un recurso de un servidor Web, las cookies que envió el servidor previamente al cliente se devuelven al servidor Web como parte de la solicitud formulada por el explorador. Las cookies se eliminan cuando *expiran*.



Tip de portabilidad 21.3

Los clientes pueden deshabilitar las cookies en sus exploradores Web, para asegurar que se proteja su privacidad. Dichos clientes tendrán dificultades al utilizar aplicaciones Web que dependan de las cookies para mantener la información de estado.

Uso de cookies para proporcionar recomendaciones de libros

La siguiente aplicación Web demuestra el uso de las cookies. El ejemplo contiene dos páginas. En la primera página (figuras 21.23-21.24), los usuarios elijen su lenguaje de programación favorito de un grupo de botones de opción, y envían el formulario de XHTML al servidor Web para procesarlo. El servidor Web responde creando una cookie que almacena un registro del lenguaje seleccionado, así como el número ISBN para un libro acerca de ese tema. Después, el servidor devuelve un documento de XHTML al explorador Web, con lo cual permite al usuario elegir otro lenguaje de programación favorito, o ver la segunda página en nuestra aplicación (figuras 21.25-21.26), que muestra una lista de libros recomendados, relacionados con el lenguaje de programación que el usuario seleccionó antes. Cuando el usuario hace clic en el hipervínculo, se leen las cookies que se habían almacenado antes en el cliente, y se utilizan para formar la lista de recomendaciones de libros.

El archivo ASPX en la figura 21.23 contiene cinco botones de opción (líneas 21-27) con los valores **Visual Basic 2005**, **Visual C# 2005**, **C**, **C++** y **Java**. Recuerde que puede establecer los valores de los botones de opción a través del **Editor de la colección ListItem**, que se abre haciendo clic en la propiedad **Items** de **RadioButtonList** en la ventana **Propiedades**, o haciendo clic en el vínculo **Editar elementos...** del menú de etiquetas inteligentes **Tareas de RadioButtonList**. Para seleccionar un lenguaje de programación, el usuario hace clic en uno de los botones de opción. La página contiene un botón **Enviar**, y cuando se hace clic en este botón se crea una cookie, la cual contiene un registro del lenguaje seleccionado. Una vez creada, esa cookie se agrega al encabezado de respuesta HTTP, y se produce un evento postback. Cada vez que el usuario selecciona un lenguaje y hace clic en **Enviar**, se escribe una cookie en el cliente.

Cuando ocurre el evento postback, ciertos controles se ocultan y otros se muestran. Los controles **Label**, **RadioButtonList** y **Button** que se utilizan para seleccionar un lenguaje, se ocultan. Cerca de la parte inferior de la página, se muestra un control **Label** y dos controles **HyperLink**. Uno de los vínculos solicita esta página (líneas 36-38) y el otro solicita **Recomendaciones.aspx** (líneas 41-43). Observe que al hacer clic en el primer hipervínculo (el que solicita la página actual) no se produce un postback. El archivo **Opciones.aspx** se especifica en la propiedad **NavigateUrl** del hipervínculo. Cuando se hace clic en el hipervínculo, esta página se solicita como una solicitud completamente nueva. Anteriormente en el capítulo establecimos **NavigateUrl** a un sitio

```

1  <!-- Fig. 21.23: Opciones.aspx -->
2  <!-- Permite al cliente seleccionar lenguajes de programación y acceder -->
3  <!-- a las recomendaciones de libros. -->
4  <%@ Page Language="C#" AutoEventWireup="true"
5     CodeFile="Opciones.aspx.cs" Inherits="Opciones" %>
6
7  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
8     "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
9
10 <html xmlns="http://www.w3.org/1999/xhtml" >
11   <head id="Head1" runat="server">
12     <title>Cookies</title>
13   </head>
14   <body>
15     <form id="form1" runat="server">
16       <div>
17         <asp:Label ID="indicadorLabel" runat="server" Font-Bold="True"
18           Font-Size="Large" Text="Seleccione un lenguaje de programación:">
19         </asp:Label>
20

```

Figura 21.23 | Archivo ASPX que presenta una lista de lenguajes de programación. (Parte 1 de 3).

```

21 <asp:RadioButtonList ID="lenguajesList" runat="server">
22 <asp:ListItem>Visual Basic 2005</asp:ListItem>
23 <asp:ListItem>Visual C# 2005</asp:ListItem>
24 <asp:ListItem>C</asp:ListItem>
25 <asp:ListItem>C++</asp:ListItem>
26 <asp:ListItem>Java</asp:ListItem>
27 </asp:RadioButtonList>
28
29 <asp:Button ID="enviarButton" runat="server" Text="Enviar"
  OnClick="enviarButton_Click" />
30
31 <asp:Label ID="respuestaLabel" runat="server" Font-Bold="True"
32   Font-Size="Large" Text="¡Bienvenido a las cookies!"
33   Visible="False"></asp:Label>
34 <br /><br />
35
36 <asp:HyperLink ID="lenguajeLink" runat="server"
37   Visible="False" NavigateUrl="~/Opciones.aspx">
38   Haga clic aquí para elegir otro lenguaje</asp:HyperLink>
39 <br /><br />
40
41 <asp:HyperLink ID="recomendacionesLink" runat="server"
42   Visible="False" NavigateUrl="~/Recomendaciones.aspx">
43   Haga clic aquí para obtener las recomendaciones de libros</asp:HyperLink>
44 </div>
45 </form>
46 </body>
47 </html>

```

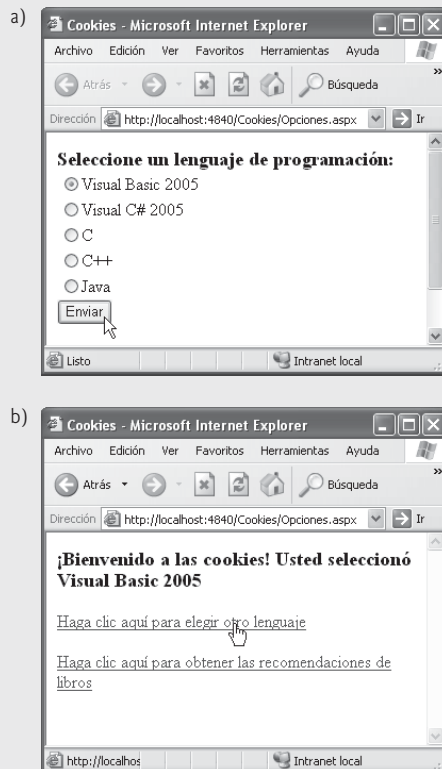


Figura 21.23 | Archivo ASPX que presenta una lista de lenguajes de programación. (Parte 2 de 3).

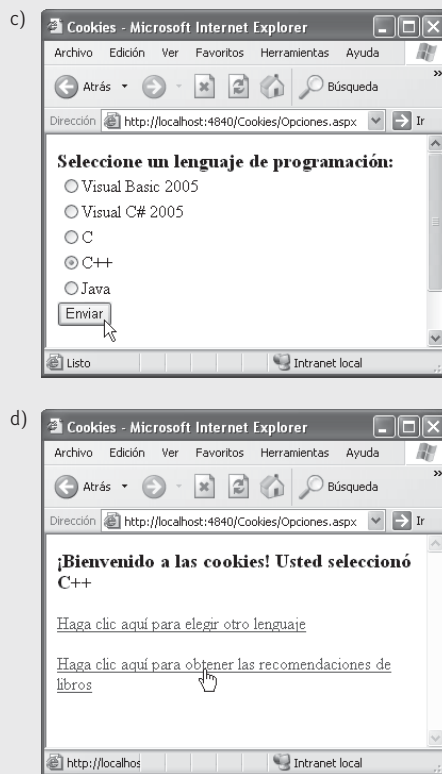


Figura 21.23 | Archivo ASPX que presenta una lista de lenguajes de programación. (Parte 3 de 3).

Web remoto (<http://www.deitel.com>). Para establecer esta propiedad a una página dentro de la misma aplicación ASP.NET, haga clic en el botón de elipsis cerca de la propiedad `NavigateUrl` en la ventana **Propiedades** para abrir el cuadro de diálogo **Seleccionar dirección URL**. Use este cuadro de diálogo para seleccionar una página dentro de su proyecto como el destino del control `HyperLink`.

Agregar y crear un vínculo hacia un nuevo formulario Web Forms

Para establecer la propiedad `NavigateUrl` a una página en la aplicación actual se requiere que la página de destino exista de antemano. Por ende, para establecer la propiedad `NavigateUrl` del segundo vínculo (el que solicita la página con las recomendaciones de libros) a `Recomendaciones.aspx`, primero debe crear este archivo; para ello, haga clic con el botón derecho en la ubicación del proyecto dentro del **Explorador de soluciones** y seleccione **Agregar nuevo elemento...** del menú que aparezca. En el cuadro de diálogo **Agregar nuevo elemento**, seleccione **Web Forms** del panel **Plantillas** y cambie el nombre del archivo a `Recomendaciones.aspx`. Por último, seleccione la casilla de verificación **Colocar el código en un archivo independiente** para indicar que el IDE debe crear un archivo de código subyacente para este archivo ASPX. Haga clic en **Agregar** para crear el archivo. (En breve hablaremos sobre el contenido de este archivo ASPX y del archivo de código subyacente.) Una vez que exista el archivo `Recomendaciones.aspx`, podrá seleccionarlo como valor para la propiedad `NavigateUrl` de un control `HyperLink` en el cuadro de diálogo **Seleccionar dirección URL**.

Escribir cookies en un archivo de código subyacente

La figura 21.24 presenta el archivo de código subyacente para `Opciones.aspx` (figura 21.23). Este archivo contiene el código que escribe una cookie en el equipo cliente cuando el usuario selecciona un lenguaje de programación. El archivo de código subyacente también modifica la apariencia de la página, en respuesta a un evento postback.


```

1  // Fig. 21.24: Opciones.aspx.cs
2  // Procesa la elección que hizo el usuario sobre un lenguaje de programación,
3  // mostrando vínculos y escribiendo una cookie en el equipo del usuario.
4  using System;
5  using System.Data;
6  using System.Configuration;
7  using System.Web;
8  using System.Web.Security;
9  using System.Web.UI;
10 using System.Web.UI.WebControls;
11 using System.Web.UI.WebControls.WebParts;
12 using System.Web.UI.HtmlControls;
13 public partial class Opciones : System.Web.UI.Page
14 {
15     // almacena valores para representar los libros como cookies
16     private System.Collections.Hashtable libros =
17         new System.Collections.Hashtable();
18
19     // inicializa la tabla Hash de los valores que se van a almacenar como cookies
20     protected void Page_Init( object sender, EventArgs e )
21     {
22         libros.Add( "Visual Basic 2005", "0-13-186900-0" );
23         libros.Add( "Visual C# 2005", "0-13-152523-9" );
24         libros.Add( "C", "0-13-142644-3" );
25         libros.Add( "C++", "0-13-185757-6" );
26         libros.Add( "Java", "0-13-148398-6" );
27     } // fin del método Page_Init
28
29     // si ocurrió un postback, oculta el formulario y muestra los vínculos para realizar
30     // selecciones adicionales o ver las recomendaciones
31     protected void Page_Load( object sender, EventArgs e )
32     {
33         if ( IsPostBack )
34         {
35             // el usuario envió información, por lo que se muestra el mensaje
36             // y los hipervínculos apropiados
37             respuestaLabel.Visible = true;
38             lenguajeLink.Visible = true;
39             recomendacionesLink.Visible = true;
40
41             // oculta los demás controles que se usan para seleccionar lenguajes
42             indicadorLabel.Visible = false;
43             lenguajesList.Visible = false;
44             enviarButton.Visible = false;
45
46             // si el usuario hizo una selección, la muestra en respuestaLabel
47             if ( lenguajesList.SelectedItem != null )
48                 respuestaLabel.Text += " Usted seleccionó " +
49                     lenguajesList.SelectedItem.Text.ToString();
50             else
51                 respuestaLabel.Text += " No seleccionó un lenguaje.";
52         } // fin de if
53     } // fin del método Page_Load
54
55     // escribe una cookie para registrar la selección del usuario
56     protected void enviarButton_Click( object sender, EventArgs e )
57     {
58         // si el usuario hizo una selección
59         if ( lenguajesList.SelectedItem != null )

```

Figura 21.24 | Archivo de código subyacente que escribe una cookie en el cliente. (Parte 1 de 2).


```

60     {
61         string lenguaje = lenguajesList.SelectedItem.ToString();
62
63         // obtiene el número ISBN del libro sobre el lenguaje dado
64         string ISBN = libros[ lenguaje ].ToString();
65
66         // crea cookie usando el par nombre-valor lenguaje-ISBN
67         HttpCookie cookie = new HttpCookie( lenguaje, ISBN );
68
69         // agrega cookie a la respuesta, para colocarla en el equipo del usuario
70         Response.Cookies.Add( cookie );
71     } // fin de if
72 } // fin del método submitButton_Click
73 } // fin de la clase Opciones

```

Figura 21.24 | Archivo de código subyacente que escribe una cookie en el cliente. (Parte 2 de 2).

Las líneas 16-17 crean libros como un objeto `Hashtable` (espacio de nombres `System.Collections`): una estructura de datos que almacena *pares clave-valor*. Un programa utiliza la clave para almacenar y recuperar el valor asociado en el objeto `Hashtable`. En este ejemplo, las claves son objetos `string` que contienen los nombres de los lenguajes de programación, y los valores son objetos `string` que contienen los números ISBN para los libros recomendados. La clase `Hashtable` proporciona el método **Add**, que recibe una clave y un valor como argumentos. Un valor que se agrega mediante el método `Add` se coloca en el objeto `Hashtable`, en una ubicación determinada por la clave. El valor para una entrada específica en el objeto `Hashtable` puede determinarse mediante la indexación del objeto `Hashtable` con la clave de ese valor. La expresión

`NombreHashtable[ nombreClave ]`

devuelve el valor en el par clave-valor, en el que *nombreClave* es la clave. Por ejemplo, la expresión `libros[ lenguaje ]` en la línea 64 devuelve el valor que corresponde a la clave contenida en `lenguaje`. En el capítulo 24, Estructuras de datos, hablaremos con detalle sobre la clase `Hashtable`.

Al hacer clic en el botón **Enviar** se crea una cookie si hay un lenguaje seleccionado, y se produce un postback. En el manejador de eventos `enviarButton_Click` (líneas 56-72), se crea un nuevo objeto cookie (de tipo **HttpCookie**) para almacenar el lenguaje y su correspondiente número ISBN (línea 67). Después, esta cookie se agrega (mediante `Add`) a la colección **Cookies** que se envía como parte del encabezado de respuesta HTTP (línea 70). El postback hace que la condición en la instrucción `if` de `Page_Load` (línea 33) se evalúe a `true`, y las líneas 37-51 se ejecutan. Las líneas 37-39 revelan los controles `respuestaLabel`, `lenguajeLink` y `recomendacionesList`, que en un principio estaban ocultos. Las líneas 42-44 ocultan los controles usados para obtener el lenguaje que seleccionó el usuario. La línea 47 determina si el usuario seleccionó un lenguaje. De ser así, el lenguaje se muestra en `respuestaLabel` (líneas 48-49). En caso contrario, se muestra texto en `respuestaLabel` indicando que no se seleccionó un lenguaje (línea 51).

Mostrar recomendaciones de libros con base en los valores de una cookie

Después del evento postback de `Opciones.aspx`, el usuario puede solicitar una recomendación de un libro. El hipervínculo de recomendación de libros lleva al usuario a la página `Recomendaciones.aspx` (figura 21.25) para mostrar las recomendaciones, con base en los lenguajes que seleccionó el usuario.

```

1  <!-- Fig. 21.25: Recomendaciones.aspx -->
2  <!-- Muestra recomendaciones de libros mediante el uso de cookies. -->
3  <%@ Page Language="C#" AutoEventWireup="true"
4      CodeFile="Recomendaciones.aspx.cs" Inherits="Recomendaciones" %>
5

```

Figura 21.25 | Archivo ASPX que muestra las recomendaciones de libros, con base en las cookies. (Parte 1 de 2).

```

6  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
7    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
8
9  <html xmlns="http://www.w3.org/1999/xhtml" >
10   <head runat="server">
11     <title>Recomendaciones de libros</title>
12   </head>
13   <body>
14     <form id="form1" runat="server">
15       <div>
16         <asp:Label ID="recomendacionesLabel"
17           runat="server" Text="Recomendaciones"
18           Font-Bold="True" Font-Size="X-Large">
19         </asp:Label><br /><br />
20
21         <asp:ListBox ID="librosListBox" runat="server" Height="125px"
22           Width="450px"></asp:ListBox><br /><br />
23
24         <asp:HyperLink ID="lenguajeLink" runat="server"
25           NavigateUrl="~/Opciones.aspx">
26           Haga clic aquí para elegir otro lenguaje
27         </asp:HyperLink>
28       </div>
29     </form>
30   </body>
31 </html>

```

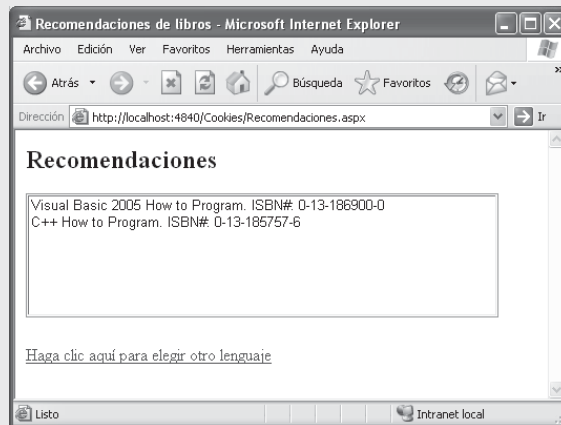


Figura 21.25 | Archivo ASPX que muestra las recomendaciones de libros, con base en las cookies. (Parte 2 de 2).

Recomendaciones.aspx contiene un control `Label` (líneas 16-19), un control `ListBox` (líneas 21-22) y un control `HyperLink` (líneas 24-27). El control `Label` muestra el texto **Recomendaciones** si el usuario seleccionó uno o más lenguajes; en caso contrario, muestra **No hay recomendaciones**. El control `ListBox` muestra la recomendación creada por el archivo de código subyacente, que se muestra en la figura 21.26. El control `HyperLink` permite al usuario regresar a `Opciones.aspx` para seleccionar lenguajes adicionales.

Archivo de código subyacente que crea las recomendaciones a partir de cookies

En el archivo de código subyacente `Recomendaciones.aspx.cs` (figura 21.26), el método `Page_Init` (líneas 17-40) recupera las cookies del cliente, usando la propiedad **Cookies** del objeto `Request` (línea 20). Esto devuelve una colección de tipo **HttpCookieCollection**, que contiene las cookies que se escribieron previamente en el cliente. Una aplicación puede leer cookies sólo si éstas se crearon en el dominio en el que se ejecuta la aplicación; un servidor Web nunca podrá acceder a las cookies creadas fuera del dominio asociado con ese servidor. Por ejemplo,

```

1  // Fig. 21.26: Recomendaciones.aspx.cs
2  // Crea recomendaciones de libros, con base en las cookies.
3  using System;
4  using System.Data;
5  using System.Configuration;
6  using System.Collections;
7  using System.Web;
8  using System.Web.Security;
9  using System.Web.UI;
10 using System.Web.UI.WebControls;
11 using System.Web.UI.WebControls.WebParts;
12 using System.Web.UI.HtmlControls;
13
14 public partial class Recomendaciones : System.Web.UI.Page
15 {
16     // lee cookies y llena el control ListBox con recomendaciones de libros
17     protected void Page_Init( object sender, EventArgs e )
18     {
19         // recupera las cookies del cliente
20         HttpCookieCollection cookies = Request.Cookies;
21
22         // si hay cookies, lista los libros apropiados, junto con los números ISBN
23         if ( cookies.Count != 0 )
24         {
25             for ( int i = 0; i < cookies.Count; i++ )
26                 librosListBox.Items.Add( cookies[ i ].Name +
27                                         " How to Program. ISBN#: " + cookies[ i ].Value );
28         } // fin de if
29         else
30         {
31             // si no hay cookies, significa que no se eligió un lenguaje, por
32             // lo que se muestra el mensaje apropiado y se oculta librosListBox
33             recomendacionesLabel.Text = "No hay recomendaciones";
34             librosListBox.Items.Clear();
35             librosListBox.Visible = false;
36
37             // modifica lenguajeLink porque no se seleccionó un lenguaje
38             lenguajeLink.Text = "Haga clic aquí para elegir un lenguaje";
39         } // fin de else
40     } // fin de Page_Init
41 } // fin de la clase Recomendaciones

```

Figura 21.26 | Leer cookies de un cliente para determinar las recomendaciones de libros.

una cookie creada por un servidor Web en el dominio deitel.com no puede ser leída por un servidor Web en cualquier otro dominio.

La línea 23 determina si existe cuando menos una cookie. Las líneas 25-27 agregan la información en la(s) cookie(s) al objeto `librosListBox`. La instrucción `for` recupera el nombre y el valor de cada cookie usando `i`, la variable de control de la instrucción, para determinar el valor actual en la colección de cookies. Las propiedades **Name** y **Value** de la clase `HttpCookie`, que contienen el lenguaje y el correspondiente ISBN, respectivamente, se concatenan con " How to Program. ISBN# " y se agregan al control `ListBox`. Las líneas 33-38 se ejecutan si no se seleccionó un lenguaje. En la figura 21.27 se muestra un resumen de algunas propiedades de `HttpCookie` de uso común.

21.6.2 Rastreo de sesiones con `HttpSessionState`

C# proporciona herramientas para rastreo de sesiones en la clase **`HttpSessionState`** de la Biblioteca de clases del .NET Framework. Para demostrar las técnicas básicas de rastreo de sesiones, modificamos la figura 21.26 de manera que utilice objetos `HttpSessionState`. La figura 21.28 presenta el archivo ASPX, y la figura 21.29 presenta el archivo de código subyacente. El archivo ASPX es similar al que presentamos en la figura 21.23, sólo que la figura 21.28 contiene dos controles `Label` adicionales (líneas 35-36 y 38-39), que veremos en breve.

Todo formulario Web Forms incluye un objeto `HttpSessionState`, el cual es accesible a través de la propiedad **Session** de la clase `Page`. A lo largo de esta sección utilizaremos la propiedad `Session` para manipular el objeto `HttpSessionState` de nuestra página. Cuando se solicita la página Web, se crea un objeto `HttpSessionState` y se asigna a la propiedad `Session` del objeto `Page`. Como resultado, a menudo haremos referencia a la propiedad `Session` como el objeto `Session`.

Propiedades	Descripción
Domain	Devuelve un objeto <code>string</code> que contiene el dominio de la cookie (es decir, el dominio del servidor Web que ejecuta la aplicación que escribió la cookie). Esto determina cuáles servidores Web pueden recibir la cookie. De manera predeterminada, las cookies se envían al servidor Web que envió originalmente la cookie al cliente. Si se modifica la propiedad <code>Domain</code> , la cookie se devuelve a un servidor Web distinto del que la escribió originalmente.
Expires	Devuelve un objeto <code>DateTime</code> que indica cuándo puede el explorador eliminar la cookie.
Name	Devuelve un objeto <code>string</code> que contiene el nombre de la cookie.
Path	Devuelve un objeto <code>string</code> que contiene la ruta hacia un directorio en el servidor (es decir, <code>Domain</code>) en el que se aplica la cookie. Las cookies pueden “dirigirse” a directorios específicos en el servidor Web. De manera predeterminada, una cookie se devuelve sólo a las aplicaciones que operan en el mismo directorio que la aplicación que envió la cookie, o a un subdirectorio de ese directorio. Si se modifica la propiedad <code>Path</code> , la cookie se devuelve a un directorio distinto del directorio en el que se escribió originalmente.
Secure	Devuelve un valor <code>bool</code> que indica si la cookie debe transmitirse a través de un protocolo seguro. El valor <code>true</code> hace que se utilice un protocolo seguro.
Value	Devuelve un objeto <code>string</code> que contiene el valor de la cookie.

Figura 21.27 | Propiedades de `HttpCookie`.

```

1  <!-- Fig. 21.28: Opciones.aspx -->
2  <!-- Permite al cliente seleccionar lenguajes de programación y -->
3  <!-- acceder a las recomendaciones de libros. -->
4  <%@ Page Language="C#" AutoEventWireup="true"
5     CodeFile="Opciones.aspx.cs" Inherits="Opciones" %>
6
7  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
8     "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
9
10 <html xmlns="http://www.w3.org/1999/xhtml" >
11   <head runat="server">
12     <title>Sessions</title>
13   </head>
14   <body>
15     <form id="form1" runat="server">
16       <div>
17         <asp:Label ID="indicadorLabel" runat="server" Font-Bold="True"
18           Font-Size="Large" Text="Seleccione un lenguaje de programación:">
19         </asp:Label>
20
21         <asp:RadioButtonList ID="lenguajesList" runat="server">
22           <asp:ListItem>Visual Basic 2005</asp:ListItem>
23           <asp:ListItem>Visual C# 2005</asp:ListItem>
24           <asp:ListItem>C</asp:ListItem>
25           <asp:ListItem>C++</asp:ListItem>

```

Figura 21.28 | Archivo de ASPX que presenta una lista de lenguajes de programación. (Parte 1 de 3).

```

26         <asp:ListItem>Java</asp:ListItem>
27     </asp:RadioButtonList>
28
29     <asp:Button ID="enviarButton" runat="server" Text="Enviar"
        OnClick="enviarButton_Click" />
30
31     <asp:Label ID="respuestaLabel" runat="server" Font-Bold="True"
32         Font-Size="Large" Text="¡Bienvenido a las sesiones!"
33         Visible="False"></asp:Label><br /><br />
34
35     <asp:Label ID="idLabel" runat="server" Visible="False">
36 </asp:Label><br /><br />
37
38     <asp:Label ID="tiempoLimiteLabel" runat="server" Visible="False">
39 </asp:Label><br /><br />
40
41     <asp:HyperLink ID="lenguajeLink" runat="server"
42         Visible="False" NavigateUrl="~/Opciones.aspx">
43         Haga clic aquí para elegir otro lenguaje
44     </asp:HyperLink><br /><br />
45
46     <asp:HyperLink ID="recomendacionesLink" runat="server"
47         Visible="False" NavigateUrl="~/Recomendaciones.aspx">
48         Haga clic aquí para obtener recomendaciones de libros</asp:HyperLink>
49 </div>
50 </form>
51 </body>
52 </html>

```

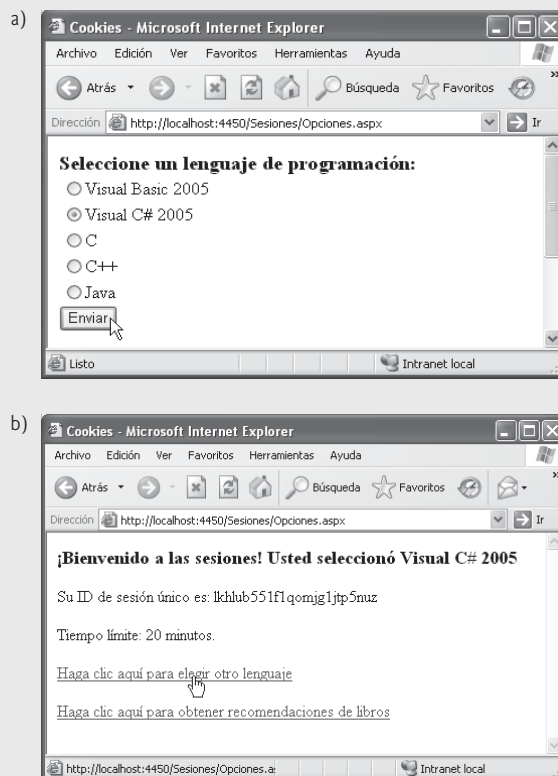


Figura 21.28 | Archivo de ASPX que presenta una lista de lenguajes de programación. (Parte 2 de 3).

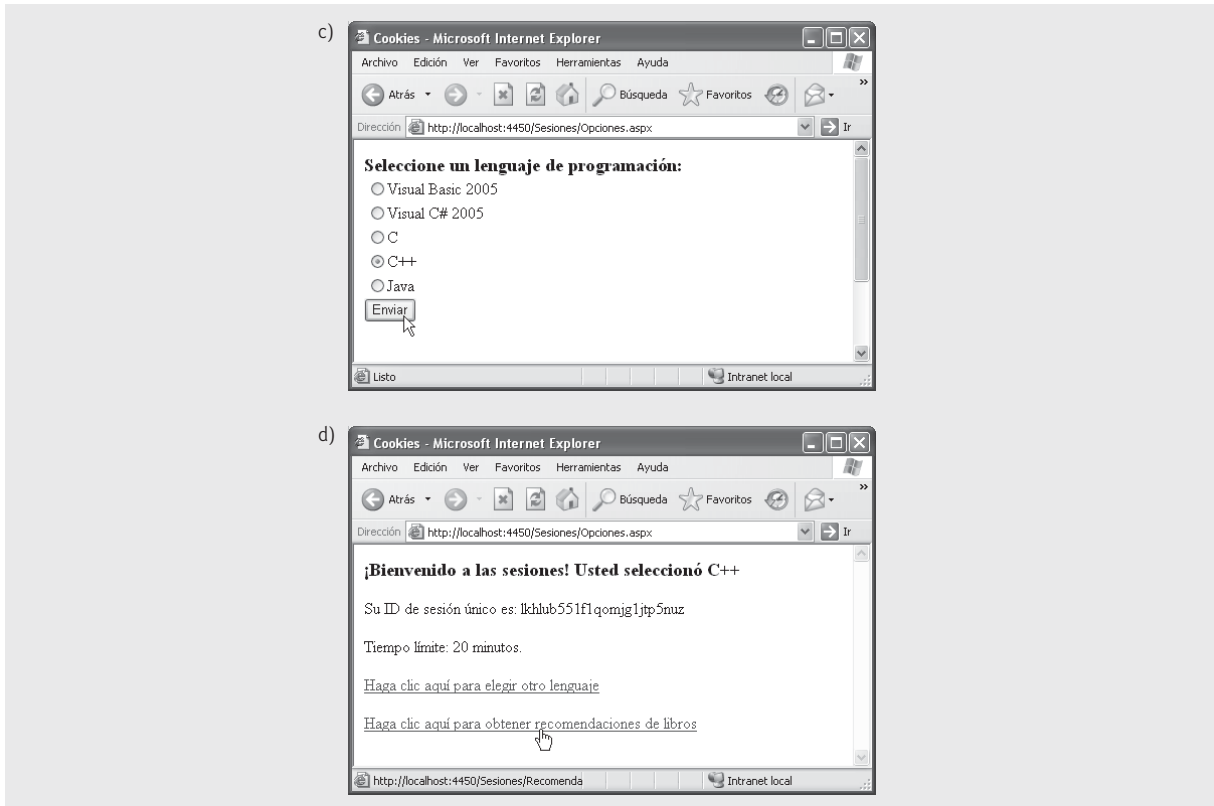


Figura 21.28 | Archivo de ASPX que presenta una lista de lenguajes de programación. (Parte 3 de 3).

Agregar elementos de sesión

Cuando el usuario oprime el botón **Enviar** en el formulario Web Forms, se invoca el método `enviarButton_Click` en el archivo de código subyacente (figura 21.29). El método `enviarButton_Click` responde agregando un par clave-valor a nuestro objeto `Session`, especificando el lenguaje elegido y el número ISBN para un libro sobre ese lenguaje. Estos pares clave-valor se conocen comúnmente como *elementos de sesión*. A continuación, se produce un evento postback. Cada vez que el usuario hace clic en **Enviar**, `enviarButton_Click` agrega un nuevo elemento de sesión al objeto `HttpSessionState`. Como la mayor parte de este ejemplo es idéntica al ejemplo anterior, nos concentraremos en las nuevas características.



Observación de ingeniería de software 21.1

Un formulario Web Forms no debe utilizar variables de instancia para mantener la información de estado del cliente, ya que cada nueva solicitud o postback se maneja mediante una nueva instancia de la página. Los formularios Web Forms deben mantener la información de estado del cliente en objetos `HttpSessionState`, ya que dichos objetos son específicos para cada cliente.

```

1 // Fig. 21.29: Opciones.aspx.cs
2 // Procesa el lenguaje de programación seleccionado por el usuario,
3 // mostrando vínculos y escribiendo una cookie en el equipo del usuario.
4 using System;
5 using System.Data;
6 using System.Configuration;

```

Figura 21.29 | Crea un elemento de sesión para cada lenguaje de programación seleccionado por el usuario en la página ASPX. (Parte 1 de 3).

```

7  using System.Web;
8  using System.Web.Security;
9  using System.Web.UI;
10 using System.Web.UI.WebControls;
11 using System.Web.UI.WebControls.WebParts;
12 using System.Web.UI.HtmlControls;
13 public partial class Opciones : System.Web.UI.Page
14 {
15     // almacena valores para representar los libros como cookies
16     private System.Collections.Hashtable libros =
17         new System.Collections.Hashtable();
18
19     //inicializa el objeto Hashtable de valores para almacenarlos como cookies
20     protected void Page_Init( object sender, EventArgs e )
21     {
22         libros.Add( "Visual Basic 2005", "0-13-186900-0" );
23         libros.Add( "Visual C# 2005", "0-13-152523-9" );
24         libros.Add( "C", "0-13-142644-3" );
25         libros.Add( "C++", "0-13-185757-6" );
26         libros.Add( "Java", "0-13-148398-6" );
27     } // fin del método Page_Init
28
29     // si ocurre el evento postback, oculta el formulario y muestra vínculos
30     // para realizar selecciones adicionales o ver recomendaciones
31     protected void Page_Load( object sender, EventArgs e )
32     {
33         if ( IsPostBack )
34         {
35             // el usuario envió información, por lo que muestra las etiquetas
36             // e hipervínculos apropiados
37             respuestaLabel.Visible = true;
38             idLabel.Visible = true;
39             tiempolimitelabel.Visible = true;
40             lenguajeLink.Visible = true;
41             recomendacionesLink.Visible = true;
42
43             // oculta los otros controles que se usan para seleccionar los lenguajes
44             indicadorLabel.Visible = false;
45             lenguajesList.Visible = false;
46             enviarButton.Visible = false;
47
48             // si el usuario hizo una selección, la muestra en respuestaLabel
49             if ( lenguajesList.SelectedItem != null )
50                 respuestaLabel.Text += " Usted seleccionó " +
51                     lenguajesList.SelectedItem.Text.ToString();
52             else
53                 respuestaLabel.Text += " No seleccionó un lenguaje.";
54
55             // muestra el ID de sesión
56             idLabel.Text = "Su ID de sesión único es: " + Session.SessionID;
57
58             // muestra el tiempo límite
59             tiempolimitelabel.Text = "Tiempo límite: " + Session.Timeout + " minutos.";
60         } // fin de if
61     } // fin del método Page_Load
62
63     // escribe una cookie para registrar la selección del usuario
64     protected void enviarButton_Click( object sender, EventArgs e )

```

Figura 21.29 | Crea un elemento de sesión para cada lenguaje de programación seleccionado por el usuario en la página ASPX. (Parte 2 de 3).


```

65     {
66         // si el usuario hizo una selección
67         if ( lenguajesList.SelectedItem != null )
68         {
69             string lenguaje = lenguajesList.SelectedItem.ToString();
70
71             // obtiene el número ISBN del libro para el lenguaje elegido
72             string ISBN = libros[ lenguaje ].ToString();
73
74             Session.Add( lenguaje, ISBN ); // agrega el par nombre/valor a Session
75         } // fin de if
76     } // fin del método enviarButton_Click
77 } // fin de la clase Opciones

```

Figura 21.29 | Crea un elemento de sesión para cada lenguaje de programación seleccionado por el usuario en la página ASPX. (Parte 3 de 3).

Al igual que una cookie, un objeto `HttpSessionState` puede almacenar pares nombre-valor. Estos elementos de sesión se colocan en un objeto `HttpSessionState` mediante una llamada al método **Add**. La línea 74 llama a **Add** para colocar el lenguaje y su correspondiente número ISBN del libro recomendado en el objeto `HttpSessionState`. Si la aplicación llama al método **Add** para agregar un atributo que tenga el mismo nombre que un atributo guardado en una sesión previa, se sustituye el objeto asociado con el atributo.



Observación de ingeniería de software 21.2

Uno de los principales beneficios del uso de objetos `HttpSessionState` (en vez de cookies) es que los objetos `HttpSessionState` pueden almacenar cualquier tipo de objeto (no sólo objetos `string`) como valores de los atributos. Esto nos proporciona una mayor flexibilidad para determinar el tipo de información de estado que se debe mantener para los clientes.

La aplicación maneja el evento postback (líneas 33-60) en el método `Page_Load`. Aquí recuperamos de las propiedades del objeto `Session` la información acerca de la sesión actual del cliente, y mostramos esta información en la página Web. La aplicación ASP.NET contiene información acerca del objeto `HttpSessionState` para el cliente actual. La propiedad **SessionID** (línea 56) contiene el **ID de sesión único**: una secuencia de letras y números al azar. La primera vez que se conecta un cliente al servidor Web, se crea un ID de sesión único para ese cliente. Cuando el cliente realiza solicitudes adicionales, el ID de sesión del cliente se compara para ese cliente. Cuando el cliente realiza solicitudes adicionales, el ID de sesión del cliente se compara con los IDs de sesión almacenados en la memoria del servidor Web, para recuperar el objeto `HttpSessionState` para ese cliente. La propiedad **Timeout** (línea 59) especifica la máxima cantidad de tiempo que un objeto `HttpSessionState` puede estar inactivo antes de descartarlo. La figura 21.30 lista algunas propiedades comunes de `HttpSessionState`.

Propiedades	Descripción
Count	Especifica el número de pares clave-valor en el objeto <code>Session</code> .
IsNewSession	Indica si ésta es una nueva sesión (es decir, si la sesión se creó al momento de cargar esta página).
IsReadOnly	Indica si el objeto <code>Session</code> es de sólo lectura.
Keys	Devuelve una colección que contiene las claves del objeto <code>Session</code> .
SessionID	Devuelve el ID de sesión único.
Timeout	Especifica el número máximo de minutos durante los que una sesión puede estar inactiva (es decir, que no se realicen solicitudes) antes de que expire. Esta propiedad se establece a 20 minutos de manera predeterminada.

Figura 21.30 | Propiedades de `HttpSessionState`.

Mostrar las recomendaciones con base en los valores de la sesión

Al igual que en el ejemplo de las cookies, esta aplicación proporciona un vínculo a Recomendaciones.aspx (figura 21.31), que muestra una lista de recomendaciones de libros con base en los lenguajes que seleccionó el usuario. Las líneas 21-22 definen un control Web ListBox que se utiliza para presentar las recomendaciones al usuario.

```

1  <!-- Fig. 21.31: Recomendaciones.aspx -->
2  <!-- Muestra las recomendaciones de libros usando sesiones. -->
3  <%@ Page Language="C#" AutoEventWireup="true"
4      CodeFile="Recomendaciones.aspx.cs" Inherits="Recomendaciones" %>
5
6  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
7      "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
8
9  <html xmlns="http://www.w3.org/1999/xhtml" >
10     <head id="Head1" runat="server">
11         <title>Recomendaciones de libros</title>
12     </head>
13     <body>
14         <form id="form1" runat="server">
15             <div>
16                 <asp:Label ID="recomendacionesLabel"
17                     runat="server" Text="Recomendaciones"
18                     Font-Bold="True" Font-Size="X-Large">
19                 </asp:Label><br /><br />
20
21                 <asp:ListBox ID="librosListBox" runat="server" Height="125px"
22                     Width="450px"></asp:ListBox><br /><br />
23
24                 <asp:HyperLink ID="lenguajeLink" runat="server"
25                     NavigateUrl="~/Opciones.aspx">
26                     Haga clic aquí para elegir otro lenguaje
27                 </asp:HyperLink>
28             </div>
29         </form>
30     </body>
31 </html>

```



Figura 21.31 | Las recomendaciones de libros basadas en sesiones se muestran en un control ListBox.

Archivo de código subyacente que crea recomendaciones de libros a partir de una sesión

La figura 21.32 presenta el archivo de código subyacente para Recomendaciones.aspx. El manejador de eventos Page_Init (líneas 17-47) recupera la información de la sesión. Si un usuario no ha seleccionado un lenguaje en Opciones.aspx, la propiedad **Count** de nuestro objeto Session será 0. Esta propiedad proporciona el número de elementos de sesión contenidos en un objeto Session. Si la propiedad Count de un objeto Session es 0 (es decir, no se seleccionó ningún lenguaje), entonces mostramos el texto **No hay recomendaciones** y actualizamos la propiedad Text del control HyperLink de vuelta a Opciones.aspx.

```

1  // Fig. 21.32: Recomendaciones.aspx.cs
2  // Crea recomendaciones de libros con base en un objeto session.
3  using System;
4  using System.Data;
5  using System.Configuration;
6  using System.Collections;
7  using System.Web;
8  using System.Web.Security;
9  using System.Web.UI;
10 using System.Web.UI.WebControls;
11 using System.Web.UI.WebControls.WebParts;
12 using System.Web.UI.HtmlControls;
13
14 public partial class Recomendaciones : System.Web.UI.Page
15 {
16     // lee las cookies y llena el control ListBox con cualquier recomendación de libro
17     protected void Page_Init( object sender, EventArgs e )
18     {
19         // almacena el nombre de una clave que se encuentre en el objeto Session
20         string nombreClave;
21
22         // determina si Session contiene información
23         if ( Session.Count != 0 )
24         {
25             for ( int i = 0; i < Session.Count; i++ )
26             {
27                 nombreClave = Session.Keys[ i ]; // almacena el nombre de la clave actual
28
29                 // usa la clave actual para mostrar uno
30                 // de los pares nombre-valor de la sesión
31                 librosListBox.Items.Add( nombreClave +
32                     " How to Program. ISBN#: " +
33                     Session[ nombreClave ].ToString() );
34             } // fin de for
35         } // fin de if
36         else
37         {
38             // si no hay elementos de sesión, no se eligió ningún lenguaje, por lo
39             // que se muestra el mensaje apropiado y se borra y oculta librosListBox
40             recomendacionesLabel.Text = "No hay recomendaciones";
41             librosListBox.Items.Clear();
42             librosListBox.Visible = false;
43
44             // modifica lenguajeLink, ya que no se seleccionó un lenguaje
45             lenguajeLink.Text = "Haga clic aquí para elegir un lenguaje";
46         } // fin de else
47     } // fin del método Page_Init
48 } // fin de la clase Recomendaciones

```

Figura 21.32 | Datos de sesión utilizados para proporcionar recomendaciones de libros al usuario.

Si el usuario eligió un lenguaje, la instrucción `for` (líneas 25-34) itera a través de los elementos de sesión de nuestro objeto `Session` y almacena en forma temporal el nombre de cada clave (línea 27). El valor en un par clave-valor se recupera del objeto `Session` mediante una indexación del objeto `Session` con el nombre de la clave, usando el mismo proceso mediante el cual recuperamos un valor de nuestro objeto `Hashtable` en la sección anterior.

La línea 27 accede a la propiedad **Keys** de la clase `HttpSessionState`, la cual devuelve una colección que contiene todas las claves en la sesión. La línea 27 indexa esta colección para recuperar la clave actual. Las líneas 31-33 concatenan el valor de `nombreClave` con el objeto `string` "How to Program. ISBN#: " y el valor del objeto `Session` para el que `nombreClave` es la clave. Este objeto `string` es la recomendación que aparece en el control `ListBox`.

21.7 Caso de estudio: conexión a una base de datos en ASP.NET

Muchos sitios Web permiten a los usuarios proporcionar retroalimentación acerca del sitio Web en un *libro de visitantes*. Por lo general, los usuarios hacen clic en un vínculo en la página inicial del sitio Web para solicitar la página del libro de visitantes. Esta página consiste comúnmente en un formulario de XHTML que contiene campos para el nombre del usuario, la dirección de correo electrónico, el mensaje/retroalimentación y así, en lo sucesivo. Después, los datos que se envían en el formulario del libro de visitantes se almacenan en una base de datos ubicada en el equipo servidor Web.

En esta sección creamos una aplicación Web Forms de libro de visitantes. La GUI de este ejemplo es un poco más compleja que la de los ejemplos anteriores. Contiene un control de datos de ASP.NET llamado **GridView**, como se muestra en la figura 21.33, el cual muestra todas las entradas en el libro de visitantes, en formato tabular. En breve le explicaremos cómo crear y configurar este control de datos. Observe que el control `GridView` muestra **abc** en modo **Diseño** para indicar los datos tipo `string` que recuperará de un origen de datos en tiempo de ejecución.

El formulario de XHTML que se presenta al usuario consiste en un campo para el nombre, un campo para la dirección de correo electrónico y un campo para el mensaje. El formulario contiene también un botón **Enviar**, para enviar los datos al servidor, y un botón **Borrar** para restablecer cada uno de los campos en el formulario. La aplicación almacena la información del libro de visitantes en una base de datos de SQL Server llamada `Libro-Visitantes.mdf`, que se encuentra en el servidor Web. (En el directorio de ejemplos de este capítulo incluimos esta base de datos. Puede descargar los ejemplos de www.deitel.com/books/csharpforprogrammers2.) Debajo del formulario de XHTML, el control `GridView` muestra los datos (es decir, las entradas en el libro de visitantes) en la tabla `Mensajes` de la base de datos.



Figura 21.33 | GUI de la aplicación Libro de visitantes en modo **Diseño**.

21.7.1 Creación de un formulario Web Forms que muestra datos de una base de datos

Ahora le explicaremos cómo crear esta GUI y establecer el enlace de datos entre el control GridView y la base de datos. Muchos de estos pasos son similares a los que realizamos en el capítulo 20 para acceder a una base de datos e interactuar con ella en una aplicación para Windows. Más adelante en esta sección presentaremos el archivo ASPX generado mediante la GUI, y hablaremos sobre el archivo de código subyacente relacionado en la siguiente sección. Para crear la aplicación de libro de visitantes, realice los siguientes pasos:

Paso 1: crear el proyecto

Cree un **Sitio Web ASP.NET** llamado **LibroVisitantes** y cambie el nombre del archivo ASPX a **LibroVisitantes.aspx**. Cambie el nombre de la clase en el archivo de código subyacente **LibroVisitantes**, y actualice la directiva **Page** en el archivo ASPX de manera acorde.

Paso 2: crear el formulario para la entrada del usuario

En modo **Diseño** para el archivo ASPX, agregue el texto **Por favor deje un mensaje en nuestro libro de visitantes:** y aplique el formato de encabezado h2, color azul marino. Como vimos en la sección 21.5.1, inserte una tabla de XHTML con dos columnas y cuatro filas, configurada de manera que el texto en cada celda se alinee con la parte superior de la celda. Coloque el texto apropiado (vea la figura 21.33) en las tres celdas superiores de la columna izquierda de la tabla. Después, coloque los controles **TextBox** llamados **nombreTextBox**, **emailTextBox** y **mensajeTextBox** en las tres celdas superiores de la columna derecha. Establezca **mensajeTextBox** de manera que sea un control **TextBox** multilínea. Por último, agregue los controles **Button** llamados **enviarButton** y **borrarButton** a la celda inferior derecha de la tabla. Establezca las leyendas de los botones a **Enviar** y **Borrar**, respectivamente. Hablaremos sobre los manejadores de eventos para estos botones cuando presentemos el archivo de código subyacente.

Paso 3: agregar un control GridView al formulario Web Forms

Agregue un control **GridView** llamado **mensajesGridView** para mostrar las entradas en el libro de visitantes. Este control aparece en la sección **Datos** del **Cuadro de herramientas**. Los colores para el control **GridView** se especifican a través del vínculo **Formato automático...** en el menú de etiquetas inteligentes **Tareas de GridView**, que aparece cuando se coloca el control **GridView** en la página. Al hacer clic en este vínculo se abre un cuadro de diálogo llamado **Formato automático** con varias opciones. En este ejemplo, elegimos **Simple**. En unos instantes, le mostraremos cómo establecer el origen de datos del control **GridView** (es decir, de donde obtiene los datos que mostrará en sus filas y columnas).

Paso 4: agregar una base de datos a una aplicación Web ASP.NET

Para usar una base de datos en una aplicación Web ASP.NET, primero debe agregarla a la carpeta **App_Data** del proyecto. Haga clic con el botón derecho del ratón sobre esta carpeta en el **Explorador de soluciones** y seleccione **Agregar elemento existente...** Localice el archivo **LibroVisitantes.mdf** en el directorio de ejemplos del capítulo, y después haga clic en **Agregar**.

Paso 5: enlazar el control GridView a la tabla Mensajes de la base de datos LibroVisitantes

Ahora que la base de datos forma parte del proyecto, podemos configurar el control **GridView** para mostrar sus datos. Abra el menú de etiquetas inteligentes **Tareas de GridView** y seleccione **<Nuevo origen de datos...>** de la lista desplegable **Elegir origen de datos**. En el **Asistente para la configuración de orígenes de datos** que aparezca, seleccione **Base de datos**. En este ejemplo, utilizamos un control **SqlDataSource**, que permite a la aplicación interactuar con la base de datos **LibroVisitantes**. Escriba **mensajesSqlDataSource** para el ID del origen de datos y haga clic en **Aceptar** para iniciar el asistente **Configurar origen de datos**. En la pantalla **Elegir la conexión de datos**, seleccione **LibroVisitantes.mdf** de la lista desplegable (figura 21.34) y después haga clic en **Siguiente >** dos veces, para continuar con la pantalla **Configurar la instrucción Select**.

La pantalla **Configurar la instrucción Select** (figura 21.35) le permite especificar cuáles datos debe recuperar el objeto **SqlDataSource** de la base de datos. Sus elecciones en esta página diseñan una instrucción **SELECT**, que se muestra en el panel inferior del cuadro de diálogo. La lista desplegable **Nombre** identifica a una tabla en la base de datos. La base de datos **LibroVisitantes** sólo contiene una tabla llamada **Mensajes**, que se selecciona de manera predeterminada. En el panel **Columnas**, haga clic en la casilla de verificación marcada con un

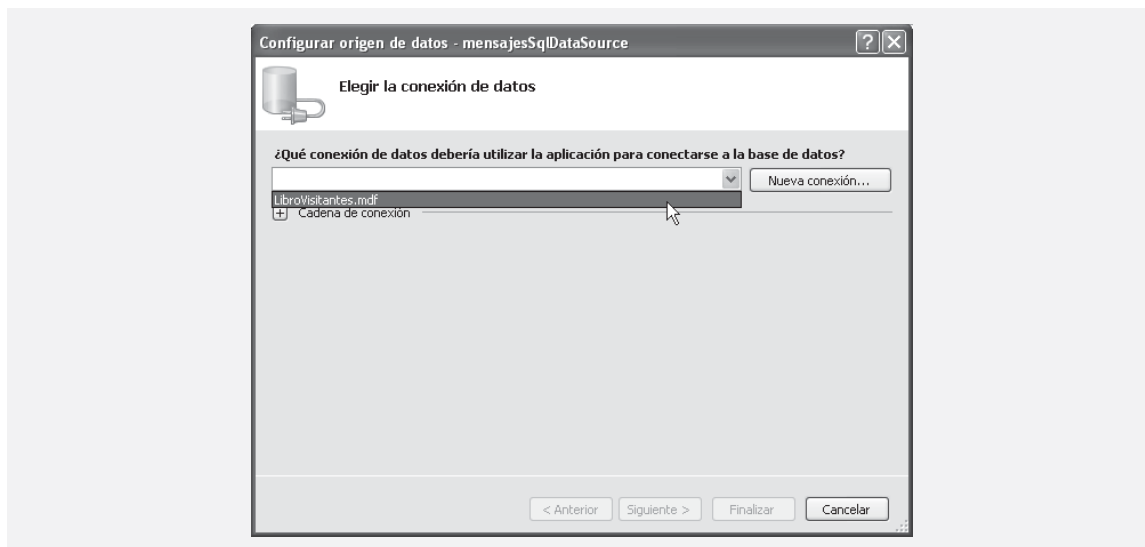


Figura 21.34 | El cuadro de diálogo **Configurar origen de datos** en Visual Web Developer.

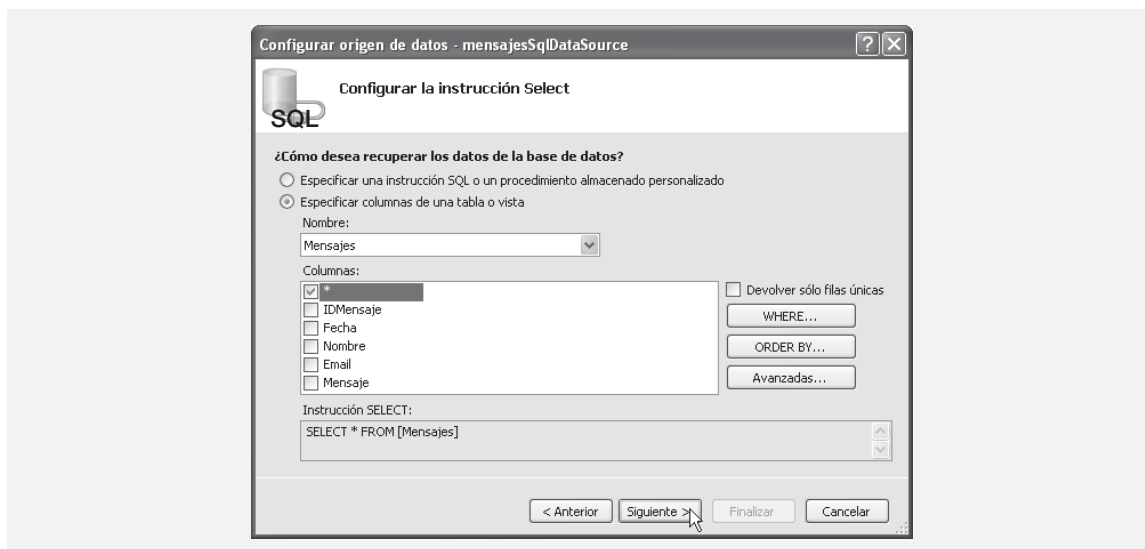


Figura 21.35 | Configuración de la instrucción SELECT utilizada por el control `SqlDataSource` para obtener datos.

asterisco (*), para indicar que desea recuperar los datos de todas las columnas en la tabla **Mensajes**. Haga clic en el botón **Avanzadas...**, y después seleccione la casilla de verificación que está a un lado de **Generar instrucciones Insert, Update y Delete**. Esto configura el control `SqlDataSource` para permitirnos insertar nuevos datos en la base de datos. En breve hablaremos sobre cómo insertar nuevas entradas en el libro de visitantes, con base en los datos que envíe el usuario mediante el formulario. Haga clic en **Aceptar** y después en **Siguiete >** para continuar con el asistente **Configurar origen de datos**.

La siguiente pantalla del asistente le permite probar la consulta que acaba de diseñar. Haga clic en **Consulta de prueba** para obtener la vista previa de los datos que devolverá el control `SqlDataSource` (mostrado en la figura 21.36).

Por último, haga clic en **Finalizar** para completar el asistente. Observe que ahora aparece un control llamado `mensajesSqlDataSource` en el formulario Web Forms, justo debajo del control `GridView` (figura 21.37).

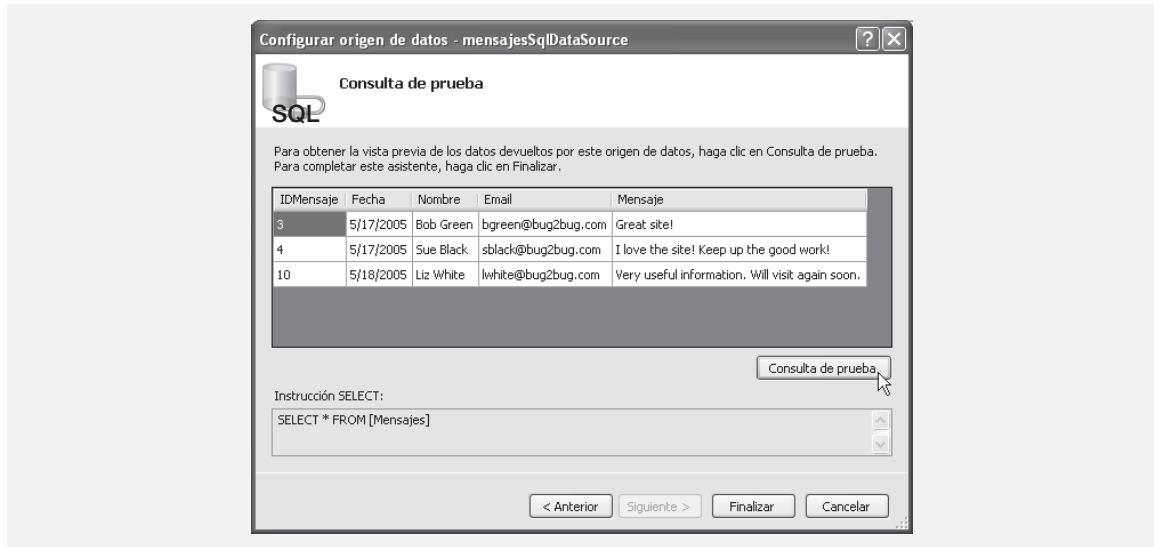


Figura 21.36 | Vista previa de los datos obtenidos por el control SqlDataSource.



Figura 21.37 | El modo **Diseño** muestra un control SqlDataSource para un control GridView.

Este control está representado en el modo **Diseño** como una caja gris que contiene su tipo y su nombre. Este control *no* aparecerá en la página Web; el cuadro gris sólo proporciona un medio para manipular el control en forma visual, a través del modo **Diseño**. Observe además que el control GridView ahora tiene encabezados de columna que corresponden a las columnas en la tabla *Mensajes*, y que cada fila contiene un número (que representa a una columna con autoincremento) o *abc* (que indica datos tipo cadena de caracteres). Los verdaderos datos del archivo de base de datos *LibroVisi tantes* aparecerán en estas filas cuando se ejecute el archivo ASPX y se vea en un explorador Web.

Paso 6: modificar las columnas del origen de datos mostrado en el control GridView

No es necesario que los visitantes a los sitios Web vean la columna `IDMensaje` cuando estén viendo las entradas anteriores en el libro de visitantes; esta columna es tan sólo una clave primaria única, requerida por la tabla `Mensajes` dentro de la base de datos. Por lo tanto, modificaremos el control `GridView` para que esta columna no aparezca en el formulario Web Forms. En el menú de etiquetas inteligentes **Tareas de GridView**, haga clic en **Editar columnas**. En el cuadro de diálogo **Campos** que aparezca (figura 21.38), seleccione `IDMensaje` en el panel **Campos seleccionados** y haga clic en la **X**. Esto eliminará la columna `IDMensaje` del control `GridView`. Haga clic en **Aceptar** para regresar a la ventana principal del IDE. Ahora, el control `GridView` deberá aparecer como en la figura 21.33.

Paso 7: modificar la forma en que el control SqlDataSource inserta los datos

Cuando se crea un control `SqlDataSource` de la manera aquí descrita, se configura para permitir operaciones SQL INSERT en la tabla de la base de datos de la que recopila datos. Hay que especificar los valores a insertar, ya sea mediante programación o a través de otros controles en el formulario Web Forms. En este ejemplo, deseamos insertar los datos introducidos por el usuario en los controles `nombreTextBox`, `emailTextBox` y `mensajeTextBox`. También queremos insertar la fecha actual; en el archivo de código subyacente especificaremos la fecha a insertar mediante programación, lo cual presentaremos más adelante.

Para configurar el control `SqlDataSource` de manera que permita una inserción de ese tipo, haga clic en el botón de elipsis que está a un lado de la propiedad **InsertQuery** del control `mensajesSqlDataSource` en la ventana **Propiedades**. A continuación aparecerá el **Editor de parámetros y comandos** (figura 21.39), el cual muestra el comando INSERT utilizado por el control `SqlDataSource`. Este comando contiene los parámetros `@Fecha`, `@Nombre`, `@Email` y `@Mensaje`. Usted debe proporcionar valores para estos parámetros, antes de insertarlos en la base de datos. Cada parámetro se lista en la sección **Parámetros** del **Editor de parámetros y comandos**. Como estableceremos el parámetro **Fecha** mediante programación, no lo modificaremos aquí. Para cada uno de los tres parámetros restantes, seleccione el parámetro y después seleccione **Control** en la lista desplegable **Origen del parámetro**. Esto indica que el valor del parámetro debe tomarse de un control. La lista desplegable **ControlID** contiene todos los controles en el formulario Web Forms. Seleccione el control apropiado para cada parámetro y después haga clic en **Aceptar**. Ahora el control `SqlDataSource` estará configurado para insertar el nombre del usuario, su dirección de correo electrónico y el mensaje en la tabla `Mensajes` de la base de datos `LibroVisi`. En breve le mostraremos cómo establecer el parámetro de la fecha, e iniciar la operación de inserción cuando el usuario haga clic en **Enviar**.

Archivo ASPX para un formulario Web Forms que interactúa con una base de datos

El archivo ASPX generado por la GUI del libro de visitantes (y por el control `mensajesSqlDataSource`) se muestra en la figura 21.40. Este archivo contiene una gran cantidad de marcado generado. Sólo hablaremos de las

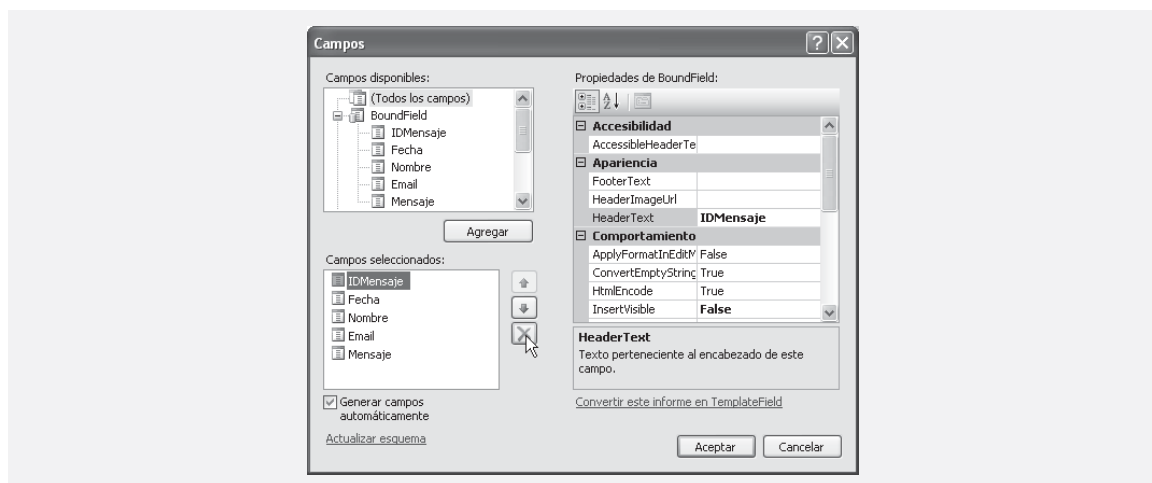


Figura 21.38 | Eliminar la columna `IDMensaje` del control `GridView`

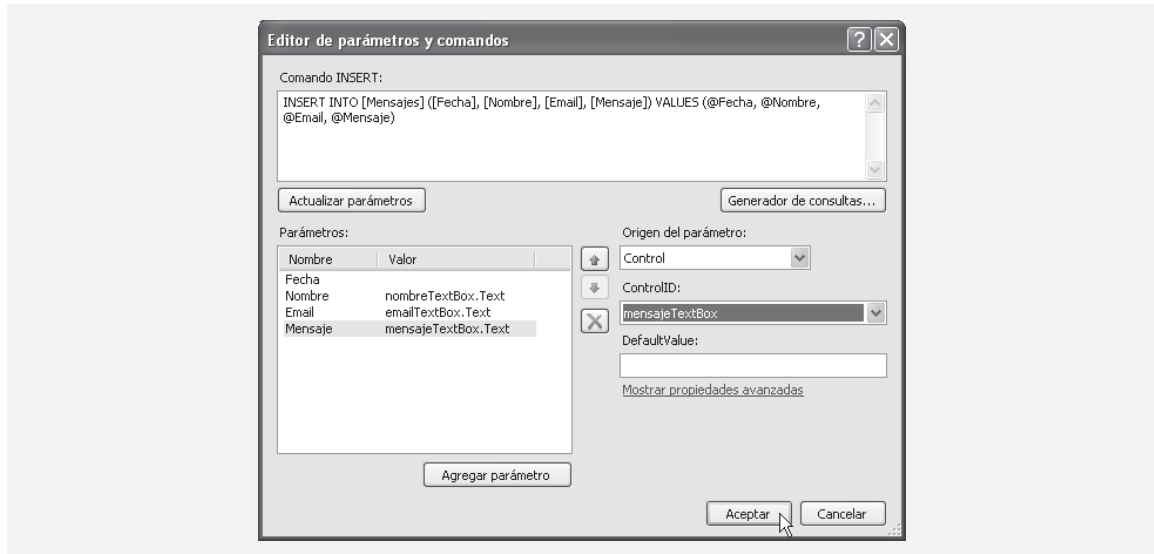


Figura 21.39 | Establecer los parámetros de INSERT con base en valores de control.

partes que son nuevas, o que tienen importancia para el ejemplo actual. Las líneas 20-58 contienen los elementos de XHTML y ASP.NET que componen el formulario para recopilar la entrada del usuario. El control `GridView` aparece en las líneas 61-87. La etiqueta inicial `<asp:GridView>` (líneas 61-65) contiene propiedades que establecen varios aspectos de la apariencia y comportamiento del control `GridView`; por ejemplo, si deben mostrarse líneas de cuadrícula entre las filas y columnas. La propiedad **`DataSourceID`** identifica al origen de datos que se utiliza para llenar el control `GridView` con datos en tiempo de ejecución. Las líneas 66-76 contienen elementos anidados que definen los estilos utilizados para dar formato a las filas del control `GridView`. El IDE configuró estos estilos, con base en el estilo **Simple** que usted seleccionó en el cuadro de diálogo **Formato automático** para el control `GridView`.

Las líneas 77-86 definen las columnas que aparecen en el control `GridView`. Cada columna se representa como un objeto **`BoundField`**, ya que los valores en las columnas están enlazados a los valores que se obtienen del origen de datos (es decir, la tabla `Mensajes` de la base de datos `LibroVisitantes`). La propiedad `DataSourceField` de cada objeto `BoundField` identifica la columna en el origen de datos con la que se enlaza la columna en el control `GridView`. La propiedad `HeaderText` indica el texto que aparece como encabezado para la columna. De manera predeterminada, éste es el nombre de la columna en el origen de datos, pero puede modificar esta propiedad según lo desee.

El control `mensajesSqlDataSource` se define mediante el marcado en las líneas 89-120 de la figura 21.40. Las líneas 90-91 contienen una propiedad **`ConnectionString`**, la cual indica la conexión a través de la cual el control `SqlDataSource` interactúa con la base de datos. El valor de esta propiedad utiliza una *expresión ASP.NET*,

```

1 <!-- Fig. 21.40: LibroVisitantes.aspx -->
2 <!-- Aplicación Web Libro de visitantes, con un formulario para que los usuarios envíen
3 <!-- entradas al libro de visitantes, y un control GridView para ver las entradas
4 <!-- existentes. -->
5 <%@ Page Language="C#" AutoEventWireup="true"
6   CodeFile="LibroVisitantes.aspx.cs" Inherits="LibroVisitantes" %>
7 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
8   "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

```

Figura 21.40 | Archivo ASPX para la aplicación libro de visitantes. (Parte I de 4).


```

9
10 <html xmlns="http://www.w3.org/1999/xhtml" >
11   <head idb"Head1" runat="server">
12     <title>Libro de visitantes</title>
13   </head>
14   <body>
15     <form id="form1" runat="server">
16       <div>
17         <h2><span style="color: navy">
18           Por favor deje un mensaje en nuestro libro de visitantes:</span></h2>
19
20         <table>
21           <tr>
22             <td style="width: 130px; height: 21px;" valign="top">
23               Su nombre:<br />
24             </td>
25             <td style="width: 300px; height: 21px;" valign="top">
26               <asp:TextBox ID="nombreTextBox" runat="server"
27                 Width="300px"></asp:TextBox>
28             </td>
29           </tr>
30           <tr>
31             <td style="width: 130px;" valign="top">
32               Su dirección de e-mail:<br />
33             </td>
34             <td style="width: 300px;" valign="top">
35               <asp:TextBox ID="emailTextBox" runat="server"
36                 Width="300px"></asp:TextBox></td>
37           </tr>
38           <tr>
39             <td style="width: 130px;" valign="top">
40               Dígalo a todo el mundo:<br />
41             </td>
42             <td style="width: 300px;" valign="top">
43               <asp:TextBox ID="mensajeTextBox" runat="server"
44                 Height="100px" Rows="8" Width="300px">
45               </asp:TextBox>
46             </td>
47           </tr>
48           <tr>
49             <td style="width: 130px;" valign="top">
50             </td>
51             <td style="width: 300px;" valign="top">
52               <asp:Button ID="enviarButton" runat="server"
53                 Text="Enviar" OnClick="enviarButton_Click" />
54               <asp:Button ID="borrarButton" runat="server"
55                 Text="Borrar" OnClick="borrarButton_Click" />
56             </td>
57           </tr>
58         </table>
59         <br />
60
61         <asp:GridView ID="mensajesGridView" runat="server"
62           AutoGenerateColumns="False" CellPadding="4"
63           ForeColor="#333333" GridLines="None"
64           DataSourceID="mensajesSqlDataSource" Width="597px"
65           DataKeyNames="IDMensaje">
66           <FooterStyle BackColor="#1C5E55" Font-Bold="True"

```

Figura 21.40 | Archivo ASPX para la aplicación libro de visitantes. (Parte 2 de 4).

```

67         ForeColor="White" />
68     <RowStyle BackColor="#E3EAE8" />
69     <PagerStyle BackColor="#666666" ForeColor="White"
70         HorizontalAlign="Center" />
71     <SelectedRowStyle BackColor="#C5BBAF" Font-Bold="True"
72         ForeColor="#333333" />
73     <HeaderStyle BackColor="#1C5E55" Font-Bold="True"
74         ForeColor="White" />
75     <EditRowStyle BackColor="#7C6F57" />
76     <AlternatingRowStyle BackColor="White" />
77     <Columns>
78         <asp:BoundField DataField="Fecha" HeaderText="Fecha"
79             SortExpression="Fecha" />
80         <asp:BoundField DataField="Nombre" HeaderText="Nombre"
81             SortExpression="Nombre" />
82         <asp:BoundField DataField="Email" HeaderText="Email"
83             SortExpression="Email" />
84         <asp:BoundField DataField="Mensaje" HeaderText="Mensaje"
85             SortExpression="Mensaje" />
86     </Columns>
87 </asp:GridView>
88
89 <asp:SqlDataSource ID="mensajesSqlDataSource" runat="server"
90     ConnectionString=
91         "<%= $ConnectionStrings:LibroVisitantesConnectionString %>"
92     SelectCommand="SELECT * FROM [Mensajes]"
93     DeleteCommand="DELETE FROM [Mensajes] WHERE
94         [IDMensaje] = @IDMensaje"
95     InsertCommand="INSERT INTO [Mensajes]
96         ([Fecha], [Nombre], [Email], [Mensaje]) VALUES
97         (@Fecha, @Nombre, @Email, @Mensaje)"
98     UpdateCommand="UPDATE [Mensajes] SET [Fecha] = @Fecha,
99         [Nombre] = @Nombre, [Email] = @Email, [Mensaje] = @Mensaje
100     WHERE [IDMensaje] = @IDMensaje">
101     <DeleteParameters>
102         <asp:Parameter Name="IDMensaje" Type="Int32" />
103     </DeleteParameters>
104     <UpdateParameters>
105         <asp:Parameter Name="Fecha" Type="String" />
106         <asp:Parameter Name="Nombre" Type="String" />
107         <asp:Parameter Name="Email" Type="String" />
108         <asp:Parameter Name="Mensaje" Type="String" />
109         <asp:Parameter Name="IDMensaje" Type="Int32" />
110     </UpdateParameters>
111     <InsertParameters>
112         <asp:Parameter Name="Fecha" Type="String" />
113         <asp:ControlParameter ControlID="nombreTextBox"
114             Name="Nombre" PropertyName="Text" Type="String" />
115         <asp:ControlParameter ControlID="emailTextBox"
116             Name="Email" PropertyName="Text" Type="String" />
117         <asp:ControlParameter ControlID="mensajeTextBox"
118             Name="Mensaje" PropertyName="Text" Type="String" />
119     </InsertParameters>
120 </asp:SqlDataSource>
121 </div>
122 </form>
123 </body>
124 </html>

```

Figura 21.40 | Archivo ASPX para la aplicación libro de visitantes. (Parte 3 de 4).

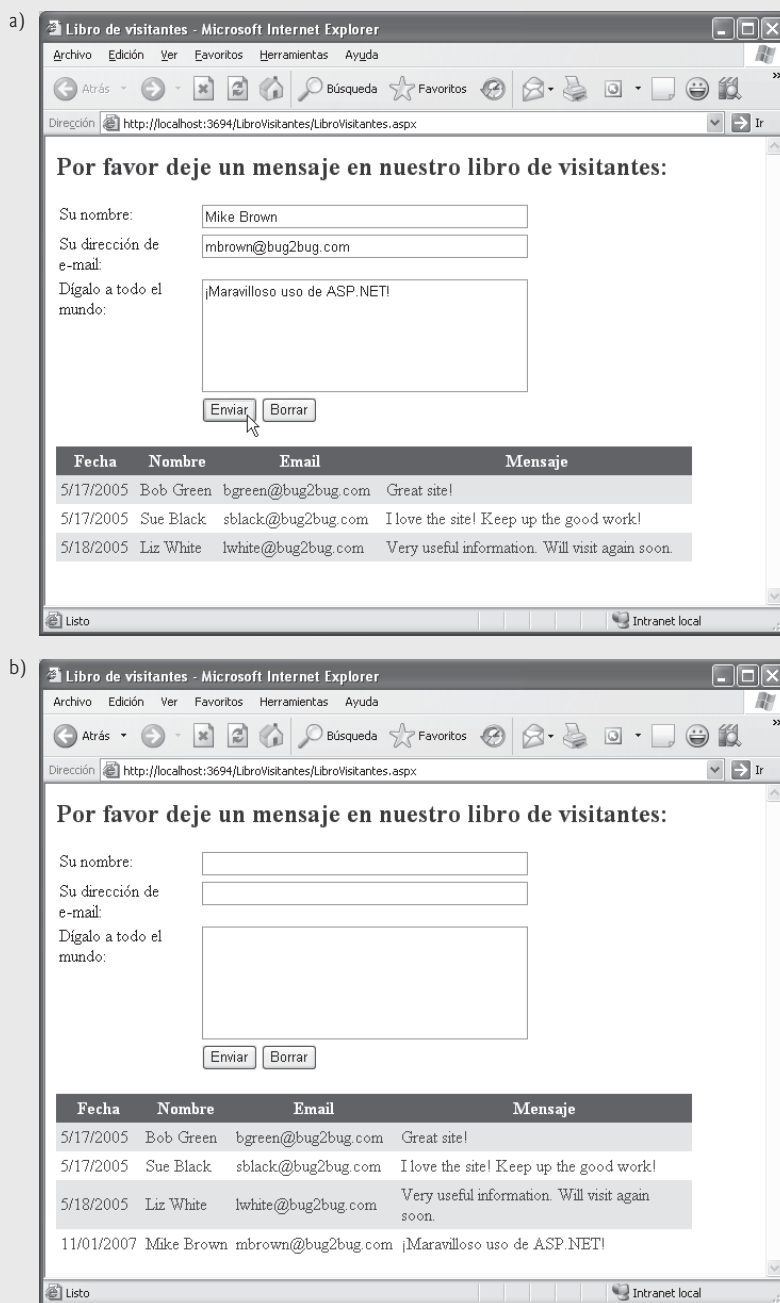


Figura 21.40 | Archivo ASPX para la aplicación libro de visitantes. (Parte 4 de 4).

delimitada por `<%%$ y %>`, para acceder a la conexión `LibroVisitantesConnectionString` almacenada en la sección `ConnectionStrings` del archivo de configuración `Web.config`. Recuerde que creamos esta cadena de conexión anteriormente en esta sección, usando el asistente **Configurar origen de datos**.

La línea 92 define la propiedad **SelectCommand** de `SqlDataSource`, la cual contiene la instrucción SQL `SELECT` que se utiliza para obtener los datos de la base de datos. Con base en nuestras acciones en el asistente **Configurar origen de datos**, esta instrucción obtiene los datos en todas las columnas de todas las filas de la tabla `Mensajes`. Las líneas 93-100 definen las propiedades **DeleteCommand**, **InsertCommand** y **UpdateCommand**, que

contienen las instrucciones SQL DELETE, INSERT y UPDATE, respectivamente. Estas propiedades también las generó el asistente **Configurar origen de datos**. En este ejemplo sólo utilizamos InsertCommand. Más adelante hablaremos sobre cómo invocar este comando.

Observe que los comandos SQL utilizados por el control `SqlDataSource` contienen varios parámetros (con el prefijo @). Las líneas 101-119 contienen elementos que definen el nombre, el tipo y, para algunos parámetros, el origen. Los parámetros que se establecen mediante programación se definen mediante elementos **Parameter** que contienen propiedades `Name` y `Type`. Por ejemplo, la línea 112 define el parámetro `Fecha` de tipo `String`. Éste corresponde al parámetro `@Fecha` en `InsertCommand` (línea 97). Los parámetros que obtienen sus valores de los controles se definen mediante elementos **ControlParameter**. Las líneas 113-118 contienen marcado que establece las relaciones entre los parámetros de INSERT y los controles `TextBox` del formulario Web Forms. En el **Editor de parámetros y comandos** (figura 21.39) establecimos estas relaciones. Cada elemento `ControlParameter` contiene una propiedad `ControlID`, la cual indica el control a partir del cual obtiene el parámetro su valor. `PropertyName` especifica la propiedad que contiene el valor real que se usará como valor para el parámetro. El IDE establece `PropertyName` con base en el tipo de control especificado por `ControlID` (de manera indirecta, a través del **Editor de parámetros y comandos**). En este caso sólo utilizamos controles `TextBox`, por lo que el valor de `PropertyName` para cada `ControlParameter` es `Text` (por ejemplo, el valor del parámetro `@Nombre` viene de `nombreTextBox.Text`). No obstante, si utilizáramos un control `DropDownList`, por ejemplo, el valor de `PropertyName` sería `SelectedValue`.

21.7.2 Modificación del archivo de código subyacente para la aplicación Libro de visitantes

Después de crear el formulario Web Forms y configurar los controles de datos utilizados en este ejemplo, haga doble clic en los botones **Enviar** y **Borrar** para crear los correspondientes manejadores de los eventos `Click` en el archivo de código subyacente `LibroVisitantes.aspx.cs` (figura 21.41). El IDE genera manejadores de eventos vacíos, por lo que debemos agregar el código apropiado para hacer que estos botones funcionen en forma apropiada. El manejador de eventos para `borrarButton` (líneas 43-48) borra cada uno de los controles `TextBox`, estableciendo su propiedad `Text` a una cadena vacía. Esto restablece el formulario para un nuevo envío de datos al libro de visitantes.

Las líneas 17-40 contienen el código para manejo de eventos de `enviarButton`, que agrega la información del usuario a la tabla `Mensajes` de la base de datos `LibroVisitantes`. Recuerde que configuramos el comando `INSERT` de `mensajesSqlDataSource` para utilizar los valores de los controles `TextBox` en el formulario Web Forms como los valores de los parámetros que se insertan en la base de datos. Aún no hemos especificado el valor de fecha a insertar. La línea 20-22 asigna una representación `string` de la fecha actual (por ejemplo, "5/27/05") a un nuevo

```

1  // Fig. 21.41: LibroVisitantes.aspx.cs
2  // Archivo de código subyacente que define los manejadores de eventos para el libro de
   visitantes.
3  using System;
4  using System.Data;
5  using System.Configuration;
6  using System.Web;
7  using System.Web.Security;
8  using System.Web.UI;
9  using System.Web.UI.WebControls;
10 using System.Web.UI.WebControls.WebParts;
11 using System.Web.UI.HtmlControls;
12
13 public partial class LibroVisitantes : System.Web.UI.Page
14 {
15     // El botón Enviar agrega una nueva entrada del libro de visitantes a la base de datos,
16     // borra el formulario y muestra la lista actualizada de entradas en el libro de
   visitantes

```

Figura 21.41 | Archivo de código subyacente para la aplicación del libro de visitantes. (Parte I de 2).

```

17 protected void enviarButton_Click( object sender, EventArgs e )
18 {
19     // crea un parámetro de fecha para almacenar la fecha actual
20     System.Web.UI.WebControls.Parameter fecha =
21         new System.Web.UI.WebControls.Parameter(
22         "Fecha", TypeCode.String, DateTime.Now.ToShortDateString() );
23
24     // establece el parámetro @Fecha con el parámetro de fecha
25     mensajesSqlDataSource.InsertParameters.RemoveAt( 0 );
26     mensajesSqlDataSource.InsertParameters.Add( fecha );
27
28     // ejecuta una instrucción SQL INSERT para agregar una nueva fila a la
29     // tabla Mensajes en la base de datos Libro de visitantes que contiene
30     // la fecha actual y el nombre, e-mail y mensaje del usuario
31     mensajesSqlDataSource.Insert();
32
33     // borra los controles TextBox
34     nombreTextBox.Text = "";
35     emailTextBox.Text = "";
36     mensajeTextBox.Text = "";
37
38     // actualiza el control GridView con el nuevo contenido de la tabla de la base de
39     // datos
40     mensajesGridView.DataBind();
41 } // fin del método enviarButton_Click
42
43 // El botón Borrar borra los controles TextBox del formulario Web Forms
44 protected void borrarButton_Click( object sender, EventArgs e )
45 {
46     nombreTextBox.Text = "";
47     emailTextBox.Text = "";
48     mensajeTextBox.Text = "";
49 } // fin del método borrarButton_Click
50 } // fin de la clase LibroVisitantes

```

Figura 21.41 | Archivo de código subyacente para la aplicación del libro de visitantes. (Parte 2 de 2).

objeto de tipo `Parameter`. Este objeto `Parameter` se identifica como "Fecha" y recibe la fecha actual como valor predeterminado. La colección `InsertParameters` de `SqlDataSource` contiene un elemento llamado `Fecha`, el cual eliminamos (`Remove`) en la línea 25 y sustituimos en la línea 26, agregando (`Add`) nuestro parámetro `fecha`. Al invocar al método `Insert` de `SqlDataSource` en la línea 31 se ejecuta el comando `INSERT` en la base de datos, con lo cual se agrega una fila a la tabla `Mensajes`. Después de insertar los datos en la base de datos, las líneas 34-36 borran los controles `TextBox` y la línea 39 invoca al método `DataBind` de `mensajesGridView` para actualizar los datos que muestra el control `GridView`. Esto hace que `mensajesSqlDataSource` (el origen de datos de `GridView`) ejecute su comando `SELECT` para obtener los datos recién actualizados de la tabla `Mensajes`.

21.8 Caso de estudio: aplicación segura de la base de datos Libros

Este caso de estudio presenta una aplicación Web en la que un usuario inicia sesión en un sitio Web, para ver una lista de publicaciones de un autor que el usuario seleccione. La aplicación consiste en varios archivos ASPX. La sección 21.8.1 presenta la aplicación funcional y explica el propósito de cada una de sus páginas Web. La sección 21.8.2 proporciona instrucciones paso a paso para guiarlo a través del proceso de crear la aplicación, y presenta el marcado en los archivos ASPX a medida que se van creando.

21.8.1 Análisis de la aplicación segura de la base de datos Libros completa

Este ejemplo utiliza una técnica conocida como *autenticación de formularios* para proteger una página, de manera que sólo los usuarios que el sitio Web reconozca puedan acceder a él. A dichos usuarios se les conoce como miembros

del sitio Web. La autenticación es una herramienta imprescindible para los sitios que sólo permiten que los miembros entren al sitio completo, o a una parte de éste. En esta aplicación, los visitantes del sitio Web deben iniciar sesión antes de que puedan ver las publicaciones en la base de datos Libros. Por lo general, la primera página que solicita un usuario es `Login.aspx` (figura 21.42). Pronto aprenderá a crear esta página utilizando un control `Login`, uno de varios *controles ASP.NET de inicio de sesión* que ayudan a crear aplicaciones seguras, mediante el uso de la autenticación. Estos controles se encuentran en la sección **Inicio de sesión** del **Cuadro de herramientas**.

La página `Login.aspx` permite que un visitante del sitio introduzca un nombre de usuario y contraseñas existentes, para iniciar sesión en el sitio Web. Un usuario que visite el sitio por primera vez debe hacer clic en el vínculo debajo del botón **Iniciar sesión** para crear un nuevo usuario, antes de intentar iniciar sesión. Así el visitante será redirigido a la página `CrearNuevoUsuario.aspx` (figura 21.43), la cual contiene un control `CreateUserWizard` que presenta al visitante un formulario de registro de usuarios. En la sección 21.8.2 hablaremos con detalle sobre el control

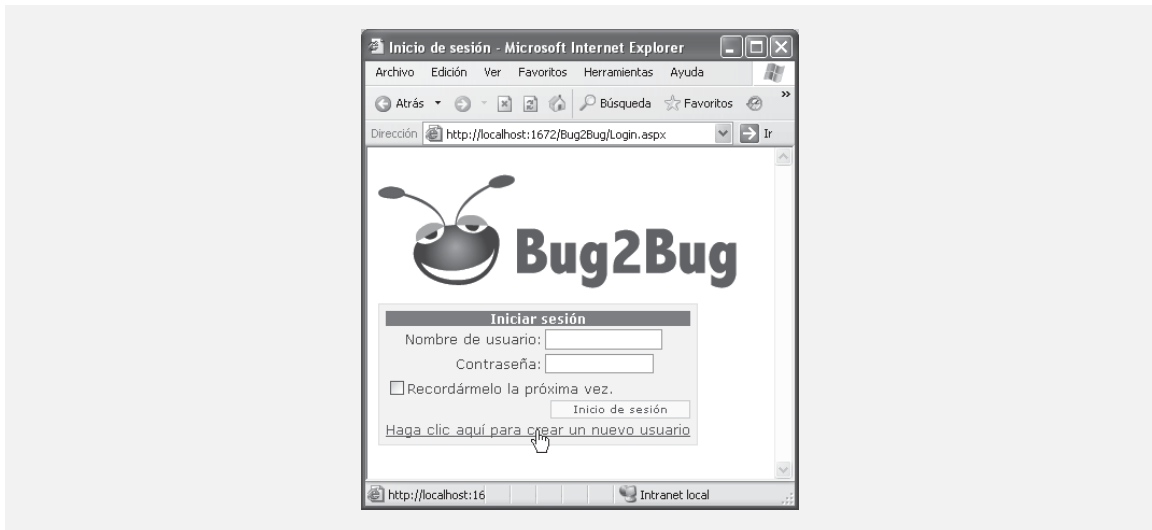


Figura 21.42 | La página `Login.aspx`, de la aplicación segura de la base de datos Libros.



Figura 21.43 | La página `CrearNuevoUsuario.aspx` de la aplicación segura de base de datos de libros.

CreateUserWizard. En la figura 21.43, utilizamos la contraseña pa\$\$word para fines de prueba; como aprenderá, el control CreateUserWizard requiere que la contraseña contenga caracteres especiales por cuestiones de seguridad. Al hacer clic en **Crear usuario** se establece una nueva cuenta de usuario. Una vez creada su cuenta, el usuario inicia sesión en forma automática y aparece un mensaje indicando que la cuenta se creó con éxito (figura 21.44).

Al hacer clic en el botón **Continuar** en la página de confirmación, el usuario es enviado a la página Libros.aspx (figura 21.45), la cual proporciona una lista desplegable de autores y una tabla que contiene los números ISBN, títulos, números de edición y años de copyright de los libros en la base de datos. De manera predeterminada se muestran todos los libros de Harvey Deitel. Aparecen vínculos en la parte inferior de la tabla, que permiten al usuario acceder a páginas adicionales de datos. Cuando el usuario elije un autor, se produce un evento postback



Figura 21.44 | Mensaje que se muestra para indicar que se creó una cuenta de usuario con éxito.

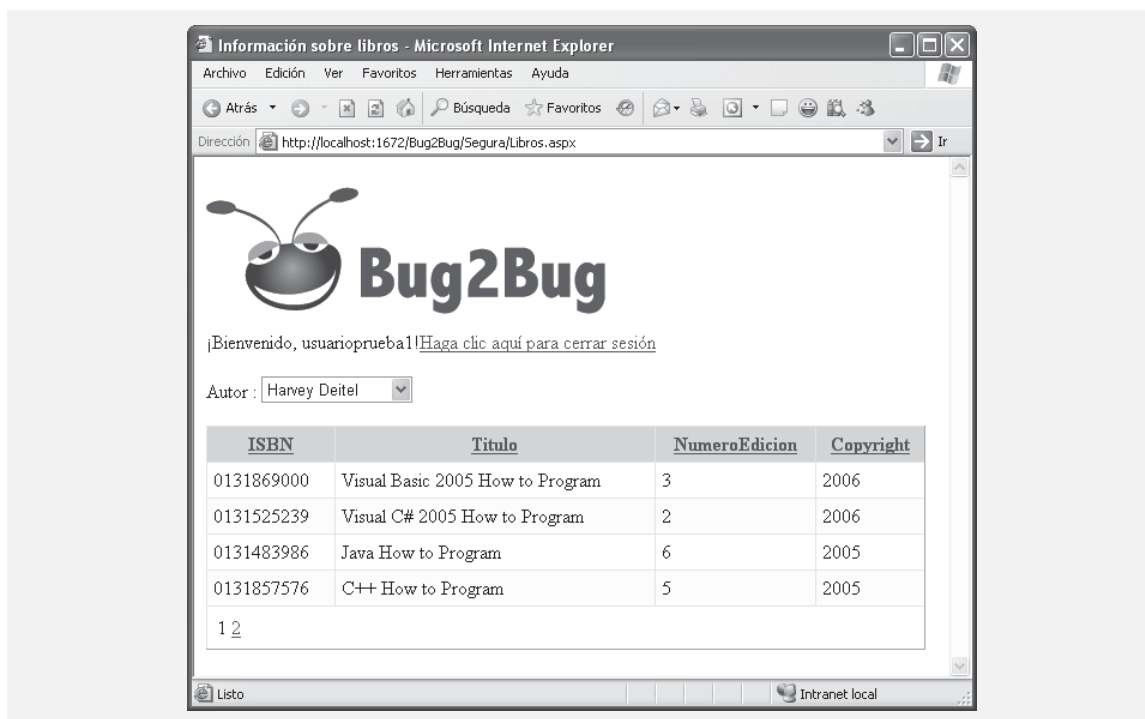


Figura 21.45 | La página Libros.aspx muestra los libros escritos por Harvey Deitel (de manera predeterminada).

y la página se actualiza para mostrar información acerca de los libros escritos por el autor seleccionado (figura 21.46).

Observe que, una vez que el usuario crea una cuenta e inicia sesión, `Libros.aspx` muestra un mensaje de bienvenida personalizado para cada usuario específico que inicia sesión. Como veremos pronto, hay un control llamado `LoginName` que proporciona esta funcionalidad. Una vez que usted agrega el control a la página, ASP.NET se encarga de los detalles acerca de cómo determinar el nombre del usuario.

Si el usuario hace clic en el vínculo **Haga clic para cerrar sesión**, se cierra su sesión y después regresa a la página `Login.aspx` (figura 21.47). Como aprenderá, este vínculo lo crea un control `LoginStatus`, el cual se encarga de los detalles relacionados con el cierre de sesión de un usuario. Para ver el listado de libros otra vez, el usuario debe iniciar sesión a través de `Login.aspx`. El control `Login` en esta página recibe el nombre de usuario y la contraseña escritas por un visitante. Después, ASP.NET compara estos valores con los nombres de usuarios y contraseñas almacenados

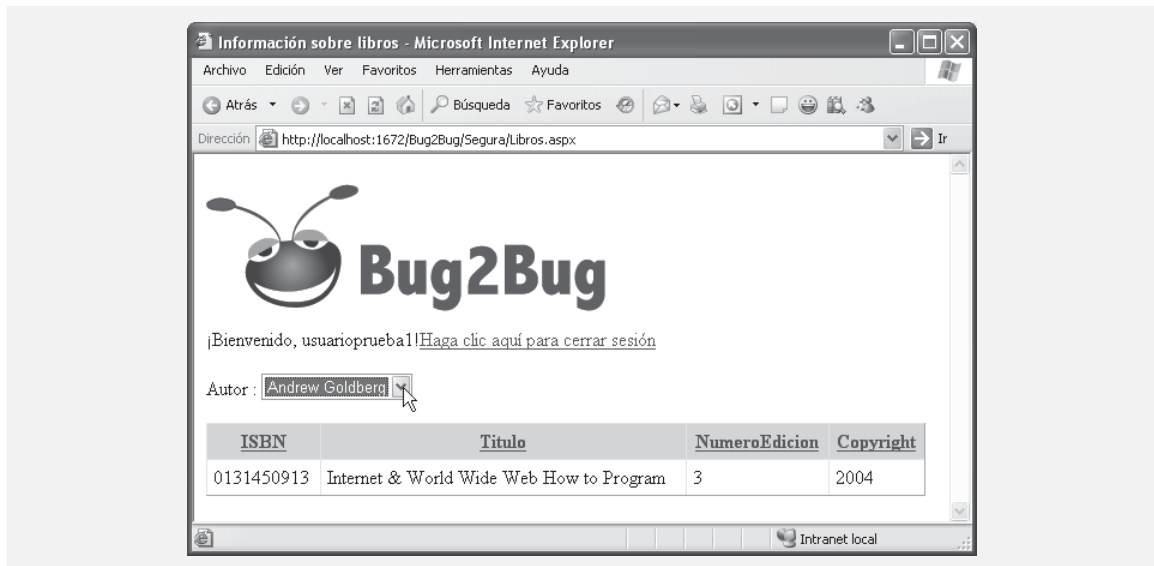


Figura 21.46 | La página `Libros.aspx` muestra los libros escritos por Andrew Goldberg.



Figura 21.47 | Inicio de sesión mediante el uso del control `Login`.

en una base de datos en el servidor. Si hay una coincidencia, el visitante es *autenticado* (es decir, se confirma la identidad del usuario). En la sección 21.8.2 explicaremos con detalle el proceso de autenticación. Cuando un usuario existente se autentica con éxito, `Login.aspx` redirige al usuario a `Libros.aspx` (figura 21.45). Si falla el intento de iniciar sesión del usuario, se muestra un mensaje de error apropiado (figura 21.48).

Observe que `Login.aspx`, `CrearNuevoUsuario.aspx` y `Libros.aspx` comparten el mismo encabezado de página, que contiene la imagen del logotipo Bug2Bug. En vez de colocar esta imagen en la parte superior de cada página, utilizamos una *página principal* para lograr esto. Como demostraremos en breve, una página principal define los elementos comunes de la GUI que hereda cada página en un conjunto de *páginas de contenido*. Así como las clases en C# pueden heredar variables de instancia y métodos de las clases existentes, las páginas de contenido heredan elementos de las páginas principales; a esto se le conoce como *herencia visual*.

21.8.2 Creación de la aplicación segura de la base de datos Libros

Ahora que está familiarizado en cuanto al comportamiento de esta aplicación, le demostraremos cómo crearla desde cero. Gracias al extenso conjunto de controles de inicio de sesión y de datos que proporciona ASP.NET, no tendrá que escribir *nada* de código para crear esta aplicación. De hecho, la aplicación no contiene archivos de código subyacente. Toda la funcionalidad se especifica a través de las propiedades de los controles, muchas de las cuales se establecen mediante asistentes y demás herramientas de programación visual. ASP.NET oculta los detalles de autenticar a los usuarios en una base de datos de nombres de usuarios y contraseñas; muestra mensajes de éxito o de error apropiados y redirige al usuario a la página correcta, con base en los resultados de la autenticación. Ahora veremos los pasos que debe realizar para crear la aplicación segura de base de datos de libros.

Paso 1: crear el sitio Web

Cree un **Sitio Web ASP.NET** en `http://localhost/Bug2Bug`, como en los ejemplos anteriores. Crearemos de manera explícita cada uno de los archivos ASPX que necesitamos en esta aplicación, así que puede eliminar el archivo `Default.aspx` generado por el IDE (y su correspondiente archivo de código subyacente); para ello seleccione la página `Default.aspx` en el **Explorador de soluciones** y oprima *Supr.* Haga clic en el botón **Aceptar** del cuadro de diálogo de confirmación para eliminar estos archivos.

Paso 2: establecer las carpetas del sitio Web

Antes de crear cualquiera de las páginas en el sitio Web, crearemos carpetas para organizar su contenido. Primero cree una carpeta llamada `Imágenes` y agregue el archivo `bug2bug.png` a esta carpeta. Encontrará esta imagen en el directorio de ejemplos para este capítulo. A continuación, agregue el archivo de base de datos `Libros.mdf` (que también encontrará en el directorio de ejemplos) a la carpeta `App_Data` del proyecto. Más adelante en esta sección le mostraremos cómo obtener datos de esta base de datos.



Figura 21.48 | Mensaje de error que se muestra para un intento de inicio de sesión sin éxito, usando control `Login`.

Paso 3: configurar las opciones de seguridad de la aplicación

En esta aplicación, queremos asegurarnos que sólo los usuarios autenticados puedan tener acceso a `Libros.aspx` (que crearemos en los *pasos 9 y 10*) para ver la información en la base de datos. Anteriormente, creamos todas nuestras páginas ASPX en el directorio raíz de la aplicación Web (por ejemplo, `http://localhost/Nombre-Proyecto`). De manera predeterminada, cualquier visitante del sitio Web (sin importar que esté autenticado o no) puede ver las páginas del directorio raíz. ASP.NET nos permite restringir el acceso a determinadas carpetas de un sitio Web. No queremos restringir el acceso a la raíz del sitio Web, ya que todos los usuarios deben poder ver `Login.aspx` y `CrearNuevoUsuario.aspx` para iniciar sesión y crear cuentas de usuario, respectivamente. Por lo tanto, si queremos restringir el acceso a `Libros.aspx`, debe residir en un directorio que no sea el directorio raíz. Cree una carpeta llamada *Segura*. Más adelante en esta sección, crearemos la página `Libros.aspx` en esta carpeta. Primero, habilitaremos la autenticación de formularios en nuestra aplicación y configuraremos la carpeta *Segura* para restringir el acceso sólo a usuarios autenticados.

Seleccione **Sitio Web > Configuración de ASP.NET** para abrir la **Herramienta Administración de sitios Web** en un explorador Web (figura 21.49). Esta herramienta le permite configurar varias opciones que determinan la manera en que se comportará la aplicación. Haga clic en el vínculo **Seguridad** o en la ficha **Seguridad** para abrir una página Web en la que puede establecer las opciones de seguridad (figura 21.50), como el tipo de autenticación que debe utilizar la aplicación. En la columna **Usuarios**, haga clic en **Seleccionar tipo de autenticación**. En la página resultante (figura 21.51), seleccione el botón de opción que está a un lado de **Desde Internet** para indicar que los usuarios iniciarán sesión a través de un formulario en el sitio Web, en el que el usuario puede escribir un nombre de usuario y contraseña (es decir, la aplicación utilizará autenticación de formularios). La opción predeterminada (**Desde una red local**) se basa en los nombres y contraseñas de usuarios de Windows para fines de autenticación. Haga clic en el botón **Listo** para guardar este cambio.

Ahora que está habilitada la autenticación de formularios, la columna **Usuarios** en la página principal de la **Herramienta Administración de sitios Web** (figura 21.52) proporciona vínculos para crear y administrar usuarios. Como vio en la sección 21.8.1, nuestra aplicación incluye la página `CrearNuevoUsuario.aspx`, en la que los usuarios pueden crear sus propias cuentas. Por lo tanto, aunque es posible crear usuarios a través de la **Herramienta Administración de sitios Web**, no lo haremos aquí.

Aun cuando no existen usuarios en este momento, configuramos la carpeta *Segura* para otorgar el acceso sólo a los usuarios autenticados (es decir, denegar el acceso a todos los usuarios no autenticados). Haga clic en el vínculo **Crear reglas de acceso** que está en la columna **Reglas de acceso** de la **Herramienta Administración de sitios**

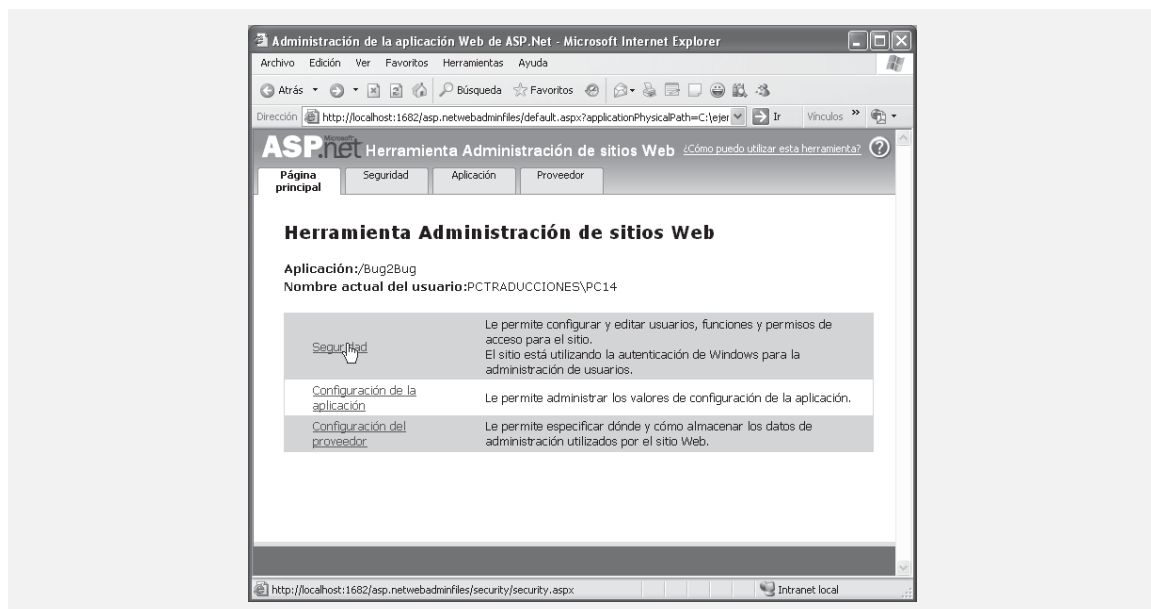


Figura 21.49 | La **Herramienta Administración de sitios Web** para configurar una aplicación Web.

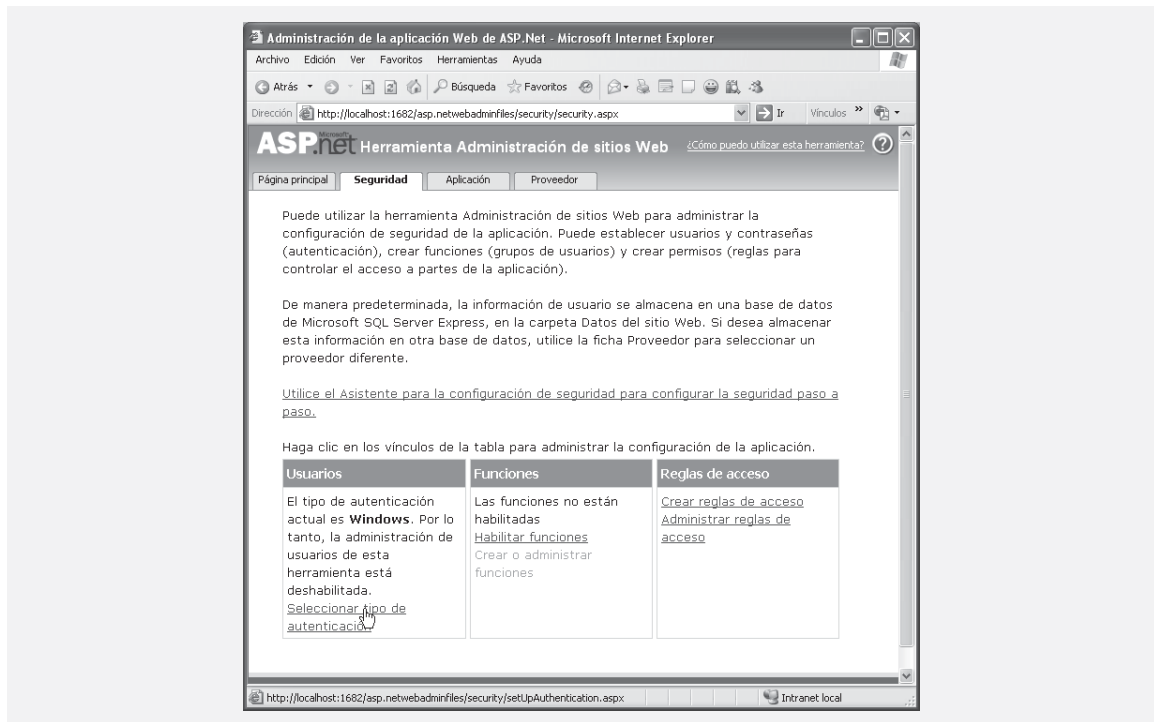


Figura 21.50 | La página **Seguridad** de la **Herramienta Administración de sitios Web**.

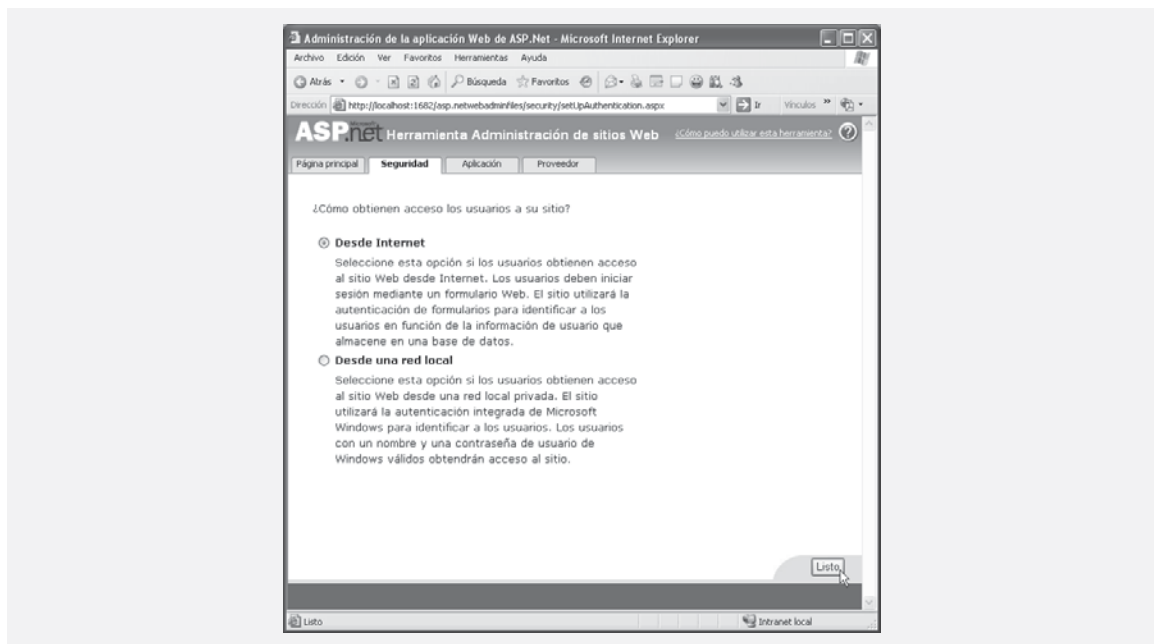


Figura 21.51 | Elegir el tipo de autenticación usada por una aplicación Web ASP.NET.

Web (figura 21.52) para ver la página **Agregar nueva regla de acceso** (figura 21.53). Esta página se utiliza para crear una *regla de acceso*: una regla que otorga o deniega el acceso a un directorio específico de la aplicación Web, para un usuario o grupo de usuarios específico. Haga clic en el directorio **Segura** en la columna izquierda de la

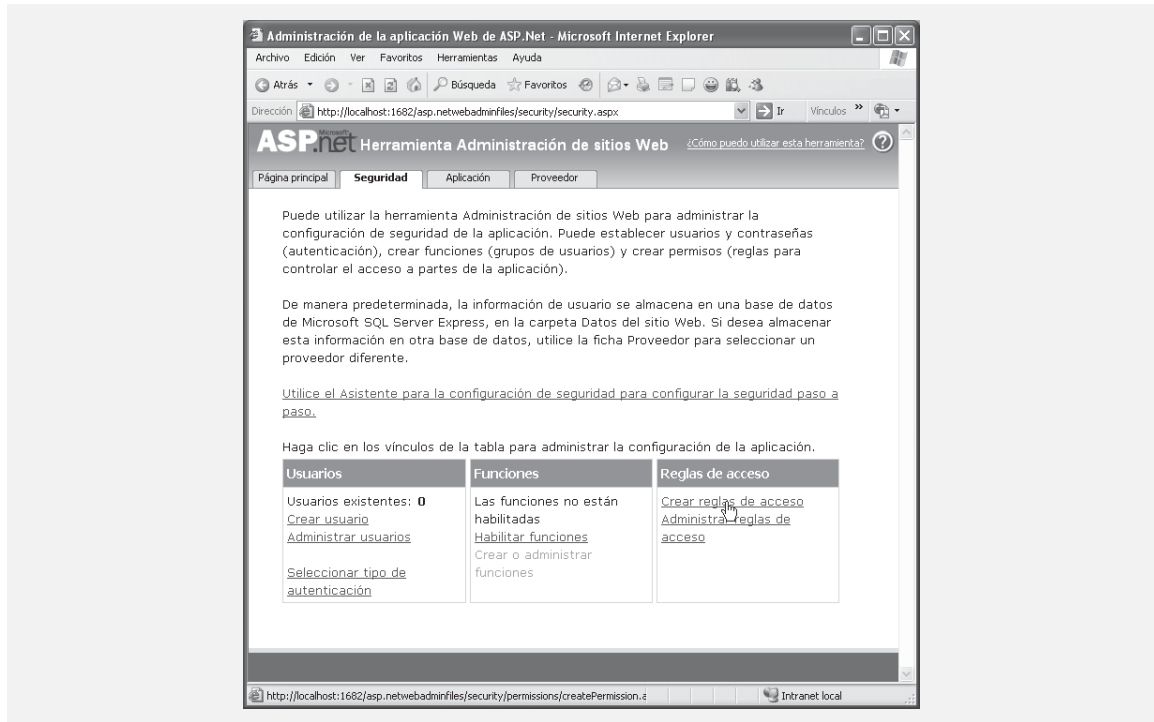


Figura 21.52 | Página principal de la **Herramienta Administración de sitios Web**, después de habilitar la autenticación de formularios.

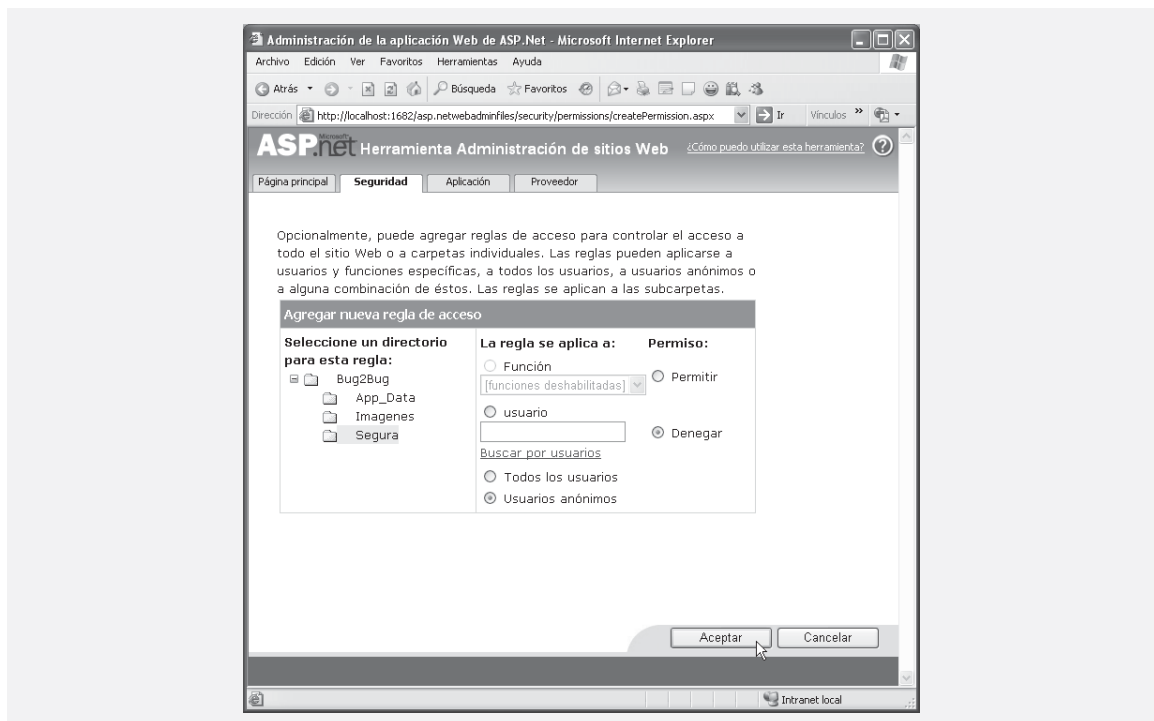


Figura 21.53 | La página **Agregar nueva regla de acceso** se utiliza para configurar el acceso a los directorios.

página para identificar el directorio al que se aplica nuestra regla de acceso. En la columna de en medio, seleccione el botón de opción identificado como **Usuarios anónimos** para especificar que la regla se aplica a los usuarios que no se han autenticado. Por último, seleccione **Denegar** en la columna derecha etiquetada como **Permiso** y después haga clic en **Aceptar**. Esta regla indica que se debe denegar el acceso a los *usuarios anónimos* (es decir, los usuarios que no se hayan identificado a sí mismos mediante el inicio de sesión) a cualquier página dentro del directorio Segura (por ejemplo, Libros.aspx). De manera predeterminada, los usuarios anónimos que intenten cargar una página en el directorio Segura se redirigen a la página Login.aspx, para que puedan identificarse a sí mismos. Observe que, como no establecimos reglas de acceso para el directorio raíz Bug2Bug, los usuarios anónimos pueden acceder a las páginas de ese directorio (por ejemplo, Login.aspx, CrearNuevoUsuario.aspx). En unos momentos crearemos estas páginas.

Paso 4: examinar los archivos Web.config generados en forma automática

Ahora hemos configurado la aplicación para usar la autenticación de formularios, y creamos una regla de acceso para asegurar que sólo los usuarios autenticados puedan acceder a la carpeta Segura. Antes de crear el contenido del sitio Web, vamos a examinar cómo aparecen los cambios realizados a través de la **Herramienta Administración de sitios Web** en el IDE. Recuerde que Web.config es un archivo de XML que se utiliza para la configuración de aplicaciones, como habilitar la depuración o almacenar cadenas de conexión de bases de datos. Visual Web Developer genera dos archivos Web.config en respuesta a nuestras acciones al usar la **Herramienta Administración de sitios Web**; uno en el directorio raíz de la aplicación y otro en la carpeta Segura. [Nota: Tal vez necesite hacer clic en el botón **Actualizar** del **Explorador de soluciones** para ver estos archivos.] En una aplicación ASP.NET, las opciones de configuración de una página se determinan con base en el archivo Web.config del directorio actual. Las opciones en este archivo tienen precedencia sobre las opciones en el archivo Web.config del directorio raíz.

Después de establecer el tipo de autenticación para la aplicación Web, el IDE genera un archivo Web.config en `http://localhost/Bug2Bug/Web.config`, el cual contiene un elemento **authentication**:

```
<authentication mode="Forms" />
```

Este elemento aparece en el archivo Web.config del directorio raíz, por lo que la opción se aplica a todo el sitio Web. El valor "Forms" del atributo **mode** especifica que deseamos utilizar la autenticación de formularios. Si hubiéramos dejado el tipo de autenticación establecido en **Desde una red local** en la **Herramienta Administración de sitios Web**, el atributo mode se establecería a "Windows". Hay que tener en cuenta que "Forms" es el modo predeterminado en un archivo Web.config generado para otro fin, como para guardar una cadena de conexión.

Después de crear la regla de acceso para la carpeta Segura, el IDE genera un segundo archivo Web.config en esa carpeta. Este archivo contiene un elemento **authorization** que indica quién (y quién no) está autorizado para acceder a esta carpeta a través de la Web. En esta aplicación, sólo queremos permitir acceso a los usuarios autenticados al contenido de la carpeta Segura, por lo que el elemento authorization aparece de la siguiente manera:

```
<authorization>
  <deny users="?" />
</authorization>
```

En vez de otorgar permiso a cada usuario autenticado por separado, denegamos el acceso a todos los que no estén autenticados (es decir, los que no hayan iniciado sesión). El elemento **deny** dentro del elemento authorization especifica los usuarios a los que deseamos denegar el acceso. Cuando el valor del atributo users se establece en "?", se deniega el acceso a la carpeta a todos los usuarios anónimos (es decir, no autenticados). Por ende, un usuario no autenticado no podrá cargar `http://localhost/Bug2Bug/Segura/Libros.aspx`, sino que será redirigido a la página Login.aspx; cuando a un usuario se le deniega el acceso a una parte de un sitio, ASP.NET envía de manera predeterminada al usuario a una página llamada Login.aspx en el directorio raíz de la aplicación.

Paso 5: creación de una página principal

Ahora que ha establecido las opciones de seguridad de la aplicación, puede crear las páginas Web de la misma. Comenzaremos con la página principal, que define los elementos que queremos que aparezcan en cada página. Una página principal es como una clase base en una jerarquía de herencia visual, y las páginas de contenido son

como clases derivadas. La página principal contiene receptáculos para el contenido personalizado creado en cada página de contenido. Las páginas de contenido heredan en forma visual el contenido de la página principal y después agregan contenido en lugar de los receptáculos de la página principal.

Por ejemplo, tal vez sea conveniente incluir una *barra de navegación* (es decir, una serie de botones para navegar por un sitio Web) en todas las páginas de un sitio. Si el sitio comprende un número extenso de páginas, agregar marcado para crear la barra de navegación para cada página puede ser un proceso que ocupe mucho tiempo. Lo que es más, si modifica después la barra de navegación, todas las páginas del sitio que la utilicen deben actualizarse. Al crear una página principal, puede especificar el marcado de la barra de navegación en un archivo y hacer que aparezca en todas las páginas de contenido, con sólo unas cuantas líneas de marcado. Si la barra de navegación cambia, sólo cambia la página principal; cualquier página de contenido que la utilice se actualiza la próxima vez que se solicite.

En este ejemplo, queremos que el logotipo Bug2Bug aparezca como encabezado en la parte superior de todas las páginas, por lo que colocaremos un control *Image* en la página principal. Cada página subsiguiente que creemos será una página de contenido basada en la página principal y, por lo tanto, incluirá el encabezado. Para crear una página principal, haga clic con el botón derecho en la ubicación del sitio Web en el **Explorador de soluciones** y seleccione la opción **Agregar nuevo elemento...** En el cuadro de diálogo **Agregar nuevo elemento**, seleccione **Página principal** de la lista de plantillas y especifique **Bug2Bug.master** como el nombre de archivo. Las páginas principales tienen la extensión de archivo **.master** y, al igual que los formularios Web Forms, pueden utilizar de manera opcional un archivo de código subyacente para definir una funcionalidad adicional. En este ejemplo no necesitamos especificar código para la página principal, por lo que deje la casilla de verificación **Colocar el código en un archivo independiente** en blanco. Haga clic en **Agregar** para crear la página.

El IDE abrirá la página principal en modo **Código** (figura 21.54) al momento de crear el archivo. [Nota: Agregamos un retorno de línea en el elemento DOCTYPE para fines de presentación.] El marcado para una página principal es casi idéntico al de un formulario Web Forms. Una diferencia es que la página principal contiene una directiva **Master** (línea 1 en la figura 21.54), la cual especifica que este archivo define a una página maestra, usando el lenguaje (Language) indicado para cualquier código. Como elegimos no utilizar un archivo de código subyacente, la página maestra también contiene un elemento **script** (líneas 6-8). El código que se coloca comúnmente en un archivo de código subyacente puede colocarse en un elemento **script**. No obstante, eliminaremos el elemento **script** de esta página, ya que no necesitamos escribir código adicional. Después de eliminar este bloque de marcado, establezca el título (**title**) de la página a **Bug2Bug**. Por último, observe que la página principal contiene un

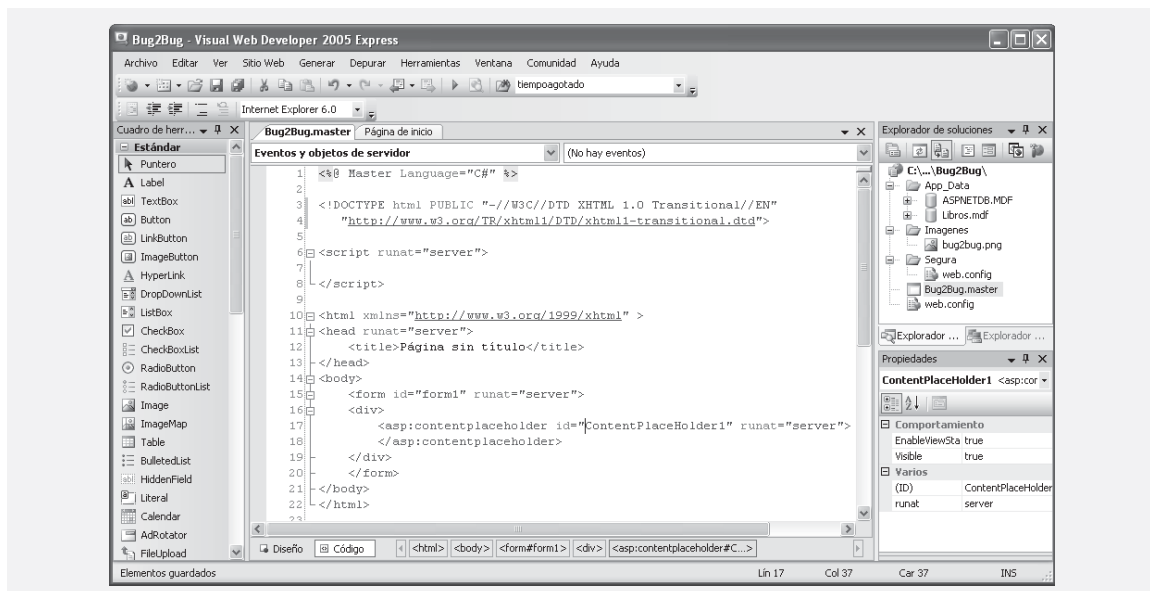


Figura 21.54 | Página principal en modo **Código**.

control **ContentPlaceholder** en las líneas 17-18. Este control sirve como un receptáculo para el contenido que se definirá mediante una página de contenido. En breve veremos cómo definir contenido para sustituir el control ContentPlaceholder.

En este punto, puede editar la página principal en modo **Diseño** (figura 21.55) como si fuera un archivo ASPX. Observe que el control ContentPlaceholder aparece como un rectángulo grande, con una barra color gris que indica el tipo y el ID del control. Utilice la ventana **Propiedades** para cambiar el ID de este control a cuerpoContent.

Para crear un encabezado en la página principal que aparezca en la parte superior de todas las páginas de contenido, insertaremos una tabla en la página principal. Coloque el cursor a la izquierda del control ContentPlaceholder y seleccione **Diseño > Insertar tabla**. En el cuadro de diálogo **Insertar tabla**, haga clic en el botón de opción **Plantilla** y seleccione **Encabezado** de la lista desplegable de plantillas de tabla disponibles. Haga clic en **Aceptar** para crear una tabla que llene la página y contenga dos filas. Arrastre y suelte el control ContentPlaceholder en la celda inferior de la tabla. Cambie la propiedad **align** de esta celda a **top**, para que el control ContentPlaceholder se alinee en forma vertical con la parte superior de la celda. A continuación, establezca la propiedad **Height** de la celda superior a 130. Agregue a esta celda un control **Image** llamado encabezadoImage, y establezca su propiedad **ImageUrl** al archivo **bug2bug.png** que se encuentra en la carpeta **Imágenes** del proyecto. (También puede arrastrar la imagen del **Explorador de soluciones** hacia la celda superior). La figura 21.56 muestra el marcado y la vista **Diseño** de la página principal completa. Como verá en el *paso 6*, una página de contenido basada en esta página principal muestra la imagen del logotipo que se define aquí, así como el contenido diseñado para esa página específica (en lugar del control ContentPlaceholder).

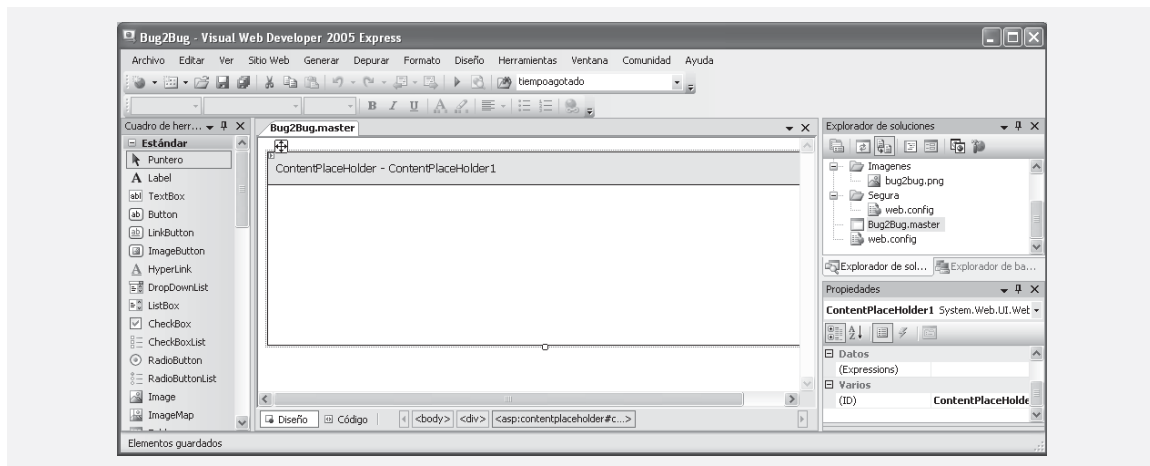


Figura 21.55 | Página principal en modo **Diseño**.

```

1  <!-- Fig. 21.56: Bug2Bug.master -->
2  <!-- Página principal que define las características comunes de todas las -->
3  <!-- páginas en la aplicación segura de la base de datos Libros -->
4  <%@ Master Language="C#" %>
5
6  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
7    "http://www.w3.org/TR/xhtml11/DTD/xhtml11-transitional.dtd">
8
9  <html xmlns="http://www.w3.org/1999/xhtml" >
10   <head runat="server">
11     <title>Bug2Bug</title>
12   </head>
13   <body>

```

Figura 21.56 | La página Bug2Bug.master define un encabezado con una imagen de logotipo para todas las páginas en la aplicación segura de la base de datos Libros. (Parte 1 de 2).


```

14     <form id="form1" runat="server">
15         <div>
16             <table border="0" cellpadding="0" cellspacing="0"
17                 style="width: 100%; height: 100%">
18                 <tr>
19                     <td height="130">
20                         <asp:Image ID="encabezadoImage" runat="server"
21                             ImageUrl="~/Imagenes/bug2bug.png" />
22                     </td>
23                 </tr>
24                 <tr>
25                     <td valign="top">
26                         <asp:contentplaceholder id="cuerpoContent"
27                             runat="server">
28                             </asp:contentplaceholder>
29                     </td>
30                 </tr>
31             </table>
32         </div>
33     </form>
34 </body>
35 </html>

```



Figura 21.56 | La página Bug2Bug.master define un encabezado con una imagen de logotipo para todas las páginas en la aplicación segura de la base de datos Libros. (Parte 2 de 2).

Paso 6: crear una página de contenido

Ahora crearemos una página de contenido basada en Bug2Bug.master. Empezaremos por crear la página CrearNuevoUsuario.aspx. Para crear este archivo, haga clic con el botón derecho del ratón en la página maestra dentro del **Explorador de soluciones**, y seleccione **Agregar página de contenido**. Esta acción hace que se agregue al proyecto un archivo Default.aspx, configurado para usar la página principal. Cambie el nombre de este archivo a CrearNuevoUsuario.aspx y después ábralo en modo Código (figura 21.57). Observe que este archivo contiene una directiva Page con una propiedad Language, una propiedad MasterPageFile y una propiedad Title. La directiva Page indica el archivo de página maestra (**MasterPageFile**) en el que se basa la página de contenido. En este caso, la propiedad MasterPageFile es "~/Bug2Bug.master" para indicar que el archivo actual se basa en la página maestra que acabamos de crear. La propiedad **Title** especifica el título que aparecerá en la barra de título del explorador Web cuando se cargue la página de contenido. Este valor, que establecimos a Crear un nuevo usuario, sustituye el valor (es decir, Bug2Bug) establecido en el elemento title de la página principal.

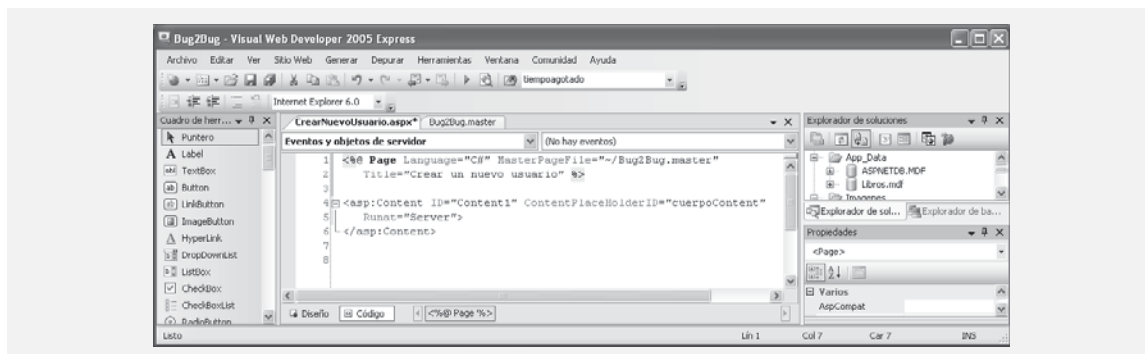


Figura 21.57 | La página de contenido CrearNuevoUsuario.aspx en modo **Código**.

Como la directiva Page de CrearNuevoUsuario.aspx especifica a Bug2Bug.master como el archivo de página principal (MasterPageFile) de la página, la página de contenido contiene implícitamente el contenido de la página principal, como los elementos DOCTYPE, html y body. El archivo de la página de contenido no duplica los elementos de XHTML que se encuentran en la página principal. En vez de ello, la página de contenido contiene un control **Content** (líneas 4-6 en la figura 21.57), en el que colocaremos el contenido específico de la página que sustituirá al control ContentPlaceholder de la página principal cuando se solicite la página de contenido. La propiedad ContentPlaceholderID del control Control identifica el elemento ContentPlaceholder en la página principal que el control debe sustituir; en este caso, cuerpoContent.

La relación entre una página de contenido y su página principal es más evidente en el modo **Diseño** (figura 21.58). La región sombreada guarda el contenido de la página principal Bug2Bug.master que aparecerá en CrearNuevoUsuario.aspx cuando se represente en un explorador Web. La única parte modificable de esta página es el control Content, que aparece en lugar del elemento ContentPlaceholder de la página principal.

Paso 7: agregar un control CreateUserWizard a una página de contenido

En la sección 21.8.1 vimos que CrearNuevoUsuario.aspx es la página en nuestro sitio Web que permite a los usuarios que nos visitan por primera vez crear cuentas de usuario. Para proporcionar esta funcionalidad, utilizamos un control **CreateUserWizard**. Coloque el cursor dentro del control Content en modo **Diseño** y haga doble clic en el control CreateUserWizard que está en la sección **Inicio de sesión** del **Cuadro de herramientas** para

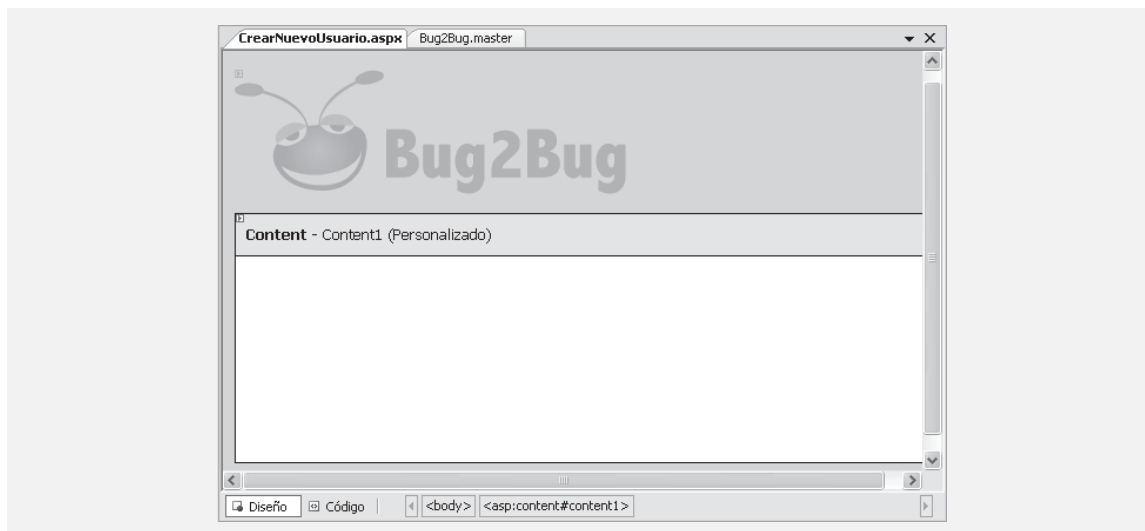


Figura 21.58 | La página de contenido CrearNuevoUsuario.aspx en modo **Diseño**.

agregarlo a la página, en la posición actual del cursor. También puede arrastrar y soltar el control en la página. Para modificar la apariencia de `CreateUserWizard`, abra el menú de etiquetas inteligentes **Tareas de CreateUserWizard** y haga clic en **Formato automático**. Seleccione el esquema de color **Profesional**.

Como vimos antes, un control `CreateUserWizard` proporciona un formulario de registro que los visitantes del sitio pueden usar para crear una cuenta de usuario. ASP.NET maneja los detalles de crear una base de datos de SQL Server (llamada `ASPNETDB.MDF` y ubicada en la carpeta `App_Data`) para almacenar los nombres de usuario, contraseñas y demás información de los usuarios de la aplicación. ASP.NET también impone un conjunto predeterminado de requerimientos para llenar el formulario. Cada campo en el formulario es requerido, la contraseña debe contener por lo menos siete caracteres, incluyendo al menos un carácter no alfanumérico, y las dos contraseñas introducidas deben coincidir. El formulario también pide una pregunta de seguridad y una respuesta, que pueden usarse para identificar a un usuario, en caso de que éste necesite restablecer o recuperar su contraseña.

Una vez que el usuario llena los campos del formulario y hace clic en el botón **Crear usuario** para enviar la información de la cuenta, ASP.NET verifica que se hayan cumplido todos los requerimientos del formulario e intenta crear la cuenta del usuario. Si ocurre un error (por ejemplo, si ya existe el nombre del usuario), el control `CreateUserWizard` muestra un mensaje debajo del formulario. Si la cuenta se crea con éxito, el formulario se sustituye por un mensaje de confirmación y un botón que permite al usuario continuar. Para ver este mensaje de confirmación en modo **Diseño**, seleccione la opción **Completar** de la lista desplegable **Paso** en el menú de etiquetas inteligente **Tareas de CreateUserWizard**.

Cuando se crea una cuenta de usuario, ASP.NET inicia la sesión del usuario en el sitio, de forma automática (en breve hablaremos más acerca del proceso de inicio de sesión). En este punto, el usuario es autenticado y se le permite acceder a la carpeta Segura. Después de crear la página `Libros.aspx` más adelante en esta sección, estableceremos la propiedad `ContinueDestinationPageUrl` de `CreateUserWizard` a `~/Segura/Libros.aspx` para indicar que el usuario debe ser redirigido a `Libros.aspx` después de hacer clic en el botón **Continuar** en la página de confirmación.

La figura 21.59 presenta el archivo `CrearNuevoUsuario.aspx` completo (se cambió su formato para mejorar la legibilidad). Dentro del control `Content`, el control `CreateUserWizard` se define mediante el marcado en las líneas 9-40. La etiqueta inicial (líneas 9-12) contiene varias propiedades que especifican estilos de formato para el control, así como la propiedad `ContinueDestinationPageUrl`, que estableceremos más adelante en este capítulo. Las líneas 14-32 contienen elementos que definen los estilos adicionales utilizados para dar formato a partes específicas del control. Por último, las líneas 34-39 especifican los dos pasos del asistente (`CreateUserWizardStep` y `CompleteWizardStep`) en un elemento `WizardSteps`. `CreateUserWizardStep` y `CompleteWizardStep` son clases que encapsulan los detalles de crear un usuario y emitir un mensaje de confirmación.

Los resultados de ejemplo en las figuras 21.59(a) y 21.59(b) muestran la creación exitosa de una cuenta de usuario con `CrearNuevoUsuario.aspx`. Utilizamos la contraseña `pa$$word` para fines de prueba. Esta contraseña cumple con los requerimientos de longitud mínima y caracteres especiales impuestos por ASP.NET, pero en una aplicación real es conveniente que utilice una contraseña que sea más difícil de adivinar. La figura 21.59(c) muestra el mensaje de error que aparece si intenta crear una segunda cuenta de usuario con el mismo nombre; ASP.NET requiere que cada nombre de usuario sea único.

```

1  <!-- %> Fig. 21.59: CrearNuevoUsuario.aspx -->
2  <!-- Página de contenido que utiliza un control CreateUserWizard para registrar
   usuarios -->
3  <%@ Page Language="C#" MasterPageFile="~/Bug2Bug.master"
4     Title="Crear un nuevo usuario" %>
5
6  <asp:Content ID="Content1" ContentPlaceHolderID="cuerpoContent"
7     Runat="Server">
8
9     <asp:CreateUserWizard ID="CreateUserWizard1" runat="server"

```

Figura 21.59 | La página de contenido `CrearNuevoUsuario.aspx` proporciona un formulario de registro de usuarios. (Parte I de 3).

```

10 BackColor="#F7F6F3" BorderColor="#E6E2D8" BorderStyle="Solid"
11 BorderWidth="1px" Font-Names="Verdana" Font-Size="0.8em"
12 ContinueDestinationPageUrl="~/Segura/Libros.aspx">
13
14 <SideBarStyle BackColor="#5D7B9D" BorderWidth="0px"
15 Font-Size="0.9em" VerticalAlign="Top" />
16 <SideBarButtonStyle BorderWidth="0px" Font-Names="Verdana"
17 ForeColor="White" />
18 <NavigationButtonStyle BackColor="#FFFBFF" BorderColor="#CCCCCC"
19 BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
20 ForeColor="#284775" />
21 <HeaderStyle BackColor="#5D7B9D" BorderStyle="Solid"
22 Font-Bold="True" Font-Size="0.9em"
23 ForeColor="White" HorizontalAlign="Center" />
24 <CreateUserButtonStyle BackColor="#FFFBFF" BorderColor="#CCCCCC"
25 BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
26 ForeColor="#284775" />
27 <ContinueButtonStyle BackColor="#FFFBFF" BorderColor="#CCCCCC"
28 BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
29 ForeColor="#284775" />
30 <StepStyle BorderWidth="0px" />
31 <TitleTextStyle BackColor="#5D7B9D" Font-Bold="True"
32 ForeColor="White" />
33
34 <WizardSteps>
35 <asp:CreateUserWizardStep runat="server">
36 </asp:CreateUserWizardStep>
37 <asp:CompleteWizardStep runat="server">
38 </asp:CompleteWizardStep>
39 </WizardSteps>
40 </asp:CreateUserWizard>
41 </asp:Content>

```



Figura 21.59 | La página de contenido CrearNuevoUsuario.aspx proporciona un formulario de registro de usuarios. (Parte 2 de 3).

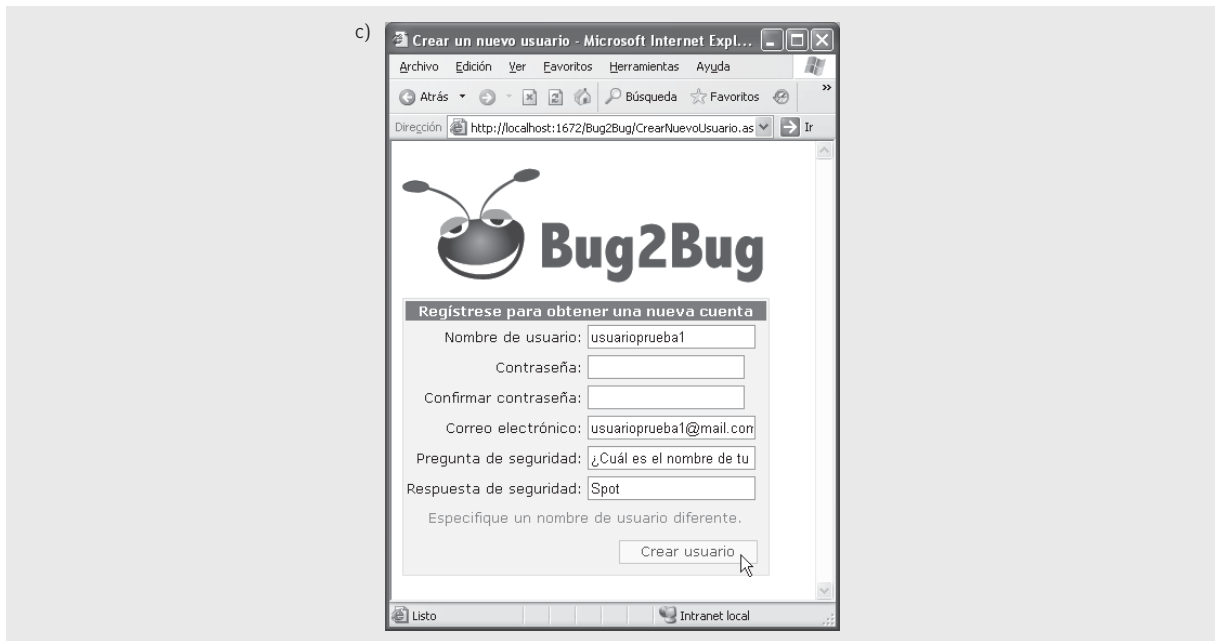


Figura 21.59 | La página de contenido `CrearNuevoUsuario.aspx` proporciona un formulario de registro de usuarios. (Parte 3 de 3).

Paso 8: crear una página de inicio de sesión

En la sección 21.8.1 vimos que `Login.aspx` es la página en nuestro sitio Web que permite que los visitantes que regresan puedan iniciar sesión en sus cuentas de usuario. Para crear esta funcionalidad, agregue otra página de contenido llamada `Login.aspx` y establezca su título a *Inicio de sesión*. En modo **Diseño**, arrastre un control **Login** (ubicado en la sección *Inicio de sesión* del **Cuadro de herramientas**) al control **Content** de la página. Abra el cuadro de diálogo **Formato automático** del menú de etiquetas inteligentes **Tareas de Login** y establezca el esquema de colores del control a **Profesional**.

A continuación, configure el control `Login` para mostrar un vínculo a la página para crear nuevos usuarios. Establezca la propiedad `CreateUserUrl` del control `Login` a `CrearNuevoUsuario.aspx`, haciendo clic en el botón de elipsis a la derecha de esta propiedad y seleccionando el archivo `CrearNuevoUsuario.aspx` en el cuadro de diálogo resultante. Después, establezca la propiedad `CreateUserText` a *Haga clic aquí para crear un nuevo usuario*. Estos valores de las propiedades hacen que aparezca un vínculo en el control `Login`.

Por último, cambiaremos el valor de la propiedad `DisplayRememberMe` del control `Login` a `False`. De manera predeterminada, el control muestra una casilla de verificación con el texto *Recordármelo la próxima vez*. Esta casilla puede usarse para permitir que un usuario permanezca autenticado más allá de una sola sesión del explorador, en la computadora actual del usuario. No obstante, deseamos requerir que los usuarios inicien sesión cada vez que visiten el sitio, por lo que deshabilitaremos esta opción.

El control `Login` encapsula los detalles relacionados con el inicio de sesión de un usuario en una aplicación Web (es decir, la autenticación de un usuario). Cuando un usuario introduce un nombre de usuario y contraseña, y después hace clic en el botón **Inicio de sesión**, ASP.NET determina si la información proporcionada coincide con la de una cuenta en la base de datos de los miembros (es decir, el archivo `ASPNETDB.MDF` creado por ASP.NET). Si coinciden, el usuario es autenticado (es decir, se confirma la identidad del usuario) y el explorador se redirige a la página especificada por la propiedad `DestinationPageUrl` del control `Login`. En la siguiente sección estableceremos esta propiedad a la página `Libros.aspx`, después de crearla. Si no se puede confirmar la identidad del usuario (es decir, si el usuario no está autenticado), el control `Login` muestra un mensaje de error (vea la figura 21.60) y el usuario puede intentar iniciar sesión de nuevo.

La figura 21.60 presenta el archivo `Login.aspx` completo. Observe que, al igual que en `CrearNuevoUsuario.aspx`, la directiva `Page` indica que esta página hereda contenido de `Bug2Bug.master`. En el control `Content`

```

1 <!-- Fig. 21.60 Login.aspx -->
2 <!-- Página de contenido que usa un control Login para autenticar usuarios. -->
3 <%@ Page Language="C#" MasterPageFile="~/Bug2Bug.master" Title="Inicio de sesión" %>
4
5 <asp:Content ID="Content1" ContentPlaceHolderID="cuerpoContent"
6   Runat="Server">
7
8   <asp:Login ID="Login1" runat="server" BackColor="#F7F6F3"
9     BorderColor="#E6E2D8" BorderPadding="4" BorderStyle="Solid"
10    BorderWidth="1px" CreateUserText="Haga clic aquí para crear un nuevo usuario"
11    CreateUserUrl="~/CrearNuevoUsuario.aspx"
12    DestinationPageUrl="~/Segura/Libros.aspx" DisplayRememberMe="False"
13    Font-Names="Verdana" Font-Size="0.8em" ForeColor="#333333" >
14
15    <LoginButtonStyle BackColor="#FFFBFF" BorderColor="#CCCCCC"
16      BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
17      Font-Size="0.8em" ForeColor="#284775" />
18    <TextBoxStyle Font-Size="0.8em" />
19    <TitleTextStyle BackColor="#5D7B9D" Font-Bold="True"
20      Font-Size="0.9em" ForeColor="White" />
21    <InstructionTextStyle Font-Italic="True" ForeColor="Black" />
22  </asp:Login>
23 </asp:Content>

```

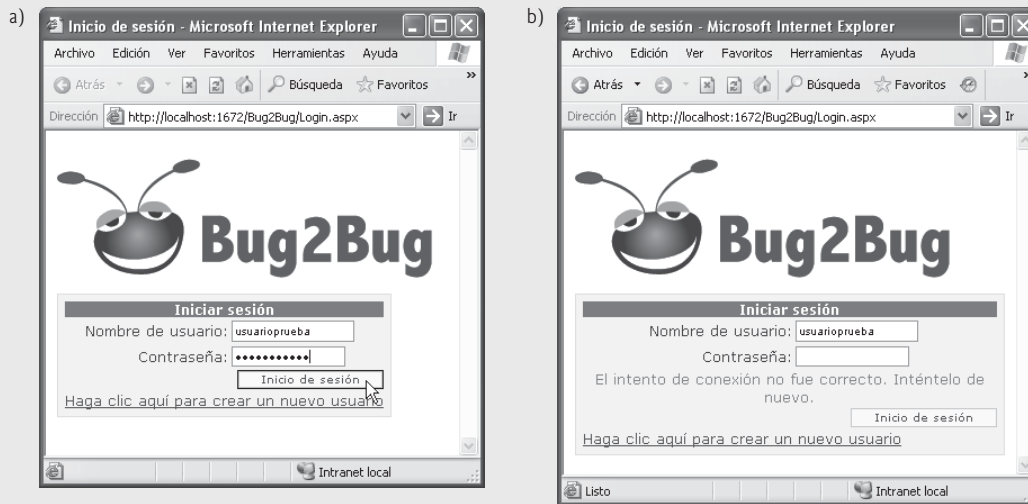


Figura 21.60 | La página de contenido Login.aspx utiliza un control Login.

que sustituye al control ContentPlaceHolder de la página inicial con el ID cuerpoContenido, las líneas 8-22 crean un control Login. Observe las propiedades CreateUserText y CreateUserUrl (líneas 10-11) que establecimos mediante la ventana **Propiedades**. La línea 12 en la etiqueta inicial para el control Login contiene las propiedades DestinationPageUrl (en el siguiente paso establecerá esta propiedad) y DisplayRememberMe, que establecimos en False. Los elementos en las líneas 15-21 definen varios estilos de formato que se aplican a ciertas partes del control. Observe que toda la funcionalidad relacionada con el verdadero inicio de sesión del usuario, o con el proceso de mostrar errores, se oculta completamente de usted.

Cuando un usuario introduce el nombre de usuario y contraseña de una cuenta de usuario existente, ASP.NET autentica al usuario y escribe en el cliente una cookie *cifrada*, que contiene información acerca del usuario autenticado. Los datos cifrados se traducen en un código que sólo el emisor y el receptor pueden entender; por lo tanto, se mantiene privado. La cookie cifrada contiene un nombre de usuario string y un valor bool que especifica si esta cookie debe persistir (es decir, permanecer en la computadora del cliente) más allá de la sesión actual. Nuestra aplicación autentica al usuario sólo para la sesión actual.

Paso 9: crear una página de contenido a la cual sólo pueden acceder los usuarios autenticados

Un usuario que haya sido autenticado se redirigirá a `Libros.aspx`. Ahora crearemos el archivo `Libros.aspx` en la carpeta `Segura`, para la cual establecimos una regla de acceso, que deniega el acceso a los usuarios anónimos. Si un usuario no autenticado solicita este archivo, será redirigido a `Login.aspx`. De aquí, el usuario puede iniciar sesión o crear una nueva cuenta; cualquiera de estas dos acciones autenticará al usuario, con lo que se le permitirá regresar a `Libros.aspx`.

Para crear la página `Libros.aspx`, haga clic con el botón derecho del ratón en la carpeta `Segura` dentro del **Explorador de soluciones**, y seleccione la opción **Agregar nuevo elemento...** En el cuadro de diálogo que aparezca, seleccione **Web Forms** y especifique el nombre de archivo `Libros.aspx`. Seleccione la casilla de verificación **Seleccionar la página principal** para indicar que este formulario Web Forms debe crearse como una página de contenido que hace referencia a una página principal, y después haga clic en **Agregar**. En el cuadro de diálogo **Seleccionar una página principal**, seleccione `Bug2Bug.master` y haga clic en **Aceptar**. El IDE creará el archivo y lo abrirá en modo **Código**. Cambie la propiedad `Title` de la directiva `Page` a **Información sobre libros**.

Paso 10: personalizar la página segura

Para personalizar la página `Libros.aspx` para un usuario específico, agregaremos un mensaje de bienvenida que contenga un control **LoginName**, el cual muestra el nombre del usuario autenticado actual. Abra `Libros.aspx` en modo **Diseño**. En el control `Content`, escriba la palabra **¡Bienvenido**, seguida de una coma y un espacio. Después arrastre un control `LoginName` del **Cuadro de herramientas** hacia la página. Cuando esta página se ejecute en el servidor, el texto (Nombre de usuario) que aparece en este control en modo **Diseño** se sustituirá por el nombre del usuario actual. En modo **Código**, escriba un signo de admiración (!) justo después del control `LoginName` (sin espacios entre ellos). [Nota: Si agrega el signo de admiración en modo **Diseño**, el IDE tal vez inserte espacios adicionales o un retorno de línea entre este carácter y el control que lo antecede. Al escribir el ! en modo **Código** nos aseguramos que aparezca adyacente al nombre del usuario.]

A continuación agregaremos un control `LoginStatus`, el cual permitirá al usuario cerrar la sesión en el sitio Web cuando termine de ver la lista de libros en la base de datos. Un control **LoginStatus** se representa en una página Web de dos maneras: de manera predeterminada, si el usuario no está autenticado, el control muestra un hipervínculo con el texto **Iniciar sesión**; si el usuario está autenticado, el control muestra un hipervínculo con el texto **Cerrar sesión**. Cada vínculo realiza la acción indicada. Agregue un control `LoginStatus` a la página, arrastrándolo del **Cuadro de herramientas** hacia la página. En este ejemplo, cualquier usuario que llegue a esta página ya debe estar autenticado, por lo que el control siempre se representará como un vínculo **Cerrar sesión**. El menú de etiquetas inteligentes **Tareas de LoginStatus** le permite alternar entre las **Vistas** del control. Seleccione la vista **Sesión iniciada** para ver el vínculo **Cerrar sesión**. Para cambiar el texto de este vínculo, modifique la propiedad `LogoutText` a **Haga clic aquí para cerrar sesión**. Después, establezca la propiedad `LogoutAction` a `RedirectToLoginPage`.

Paso 11: conectar los controles CreateUserWizard y Login a la página segura

Ahora que hemos creado la página `Libros.aspx`, podemos especificar que ésta es la página a la que los controles `CreateUserWizard` y `Login` redirigirán a los usuarios, una vez que se autentiquen. Abra la página `CrearNuevoUsuario.aspx` en modo **Diseño** y establezca la propiedad `ContinueDestinationPageUrl` del control `CreateUserWizard` a `Libros.aspx`. Ahora abra `Login.aspx` y seleccione `Libros.aspx` como el valor para la propiedad `DestinationPageUrl` del control `Login`.

En este punto, puede ejecutar la aplicación Web seleccionando **Depurar > Iniciar sin depuración**. Primero, cree una cuenta de usuario en `CrearNuevoUsuario.aspx`, y después observe cómo aparecen los controles `LoginName` y `LoginStatus` en `Libros.aspx`. Después, cierre la sesión del sitio y vuelva a iniciarla usando `Login.aspx`.

Paso 12: generar un DataSet basado en la base de datos Libros.mdf

Ahora agregaremos el contenido (es decir, la información sobre los libros) a la página segura `Libros.aspx`. Esta página debe tener un control `DropDownList` con los nombres de los autores, y un control `GridView` que muestre información acerca de los libros escritos por el autor seleccionado en el control `DropDownList`. Un usuario seleccionará un autor del control `DropDownList` para hacer que el control `GridView` muestre sólo información acerca de los libros escritos por el autor seleccionado. Como verá a continuación, creamos esta función únicamente en modo **Diseño**, sin escribir código.

Para trabajar con la base de datos **Libros**, usaremos un enfoque algo distinto al del caso de estudio anterior, en el que accedimos a la base de datos **LibroVisitantes** mediante el uso de un control **SqlDataSource**. Aquí utilizaremos un control **ObjectDataSource**, el cual encapsula a un objeto que proporciona acceso a un origen de datos. Como recordará, en el capítulo 20 accedimos a la base de datos **Libros** en una aplicación para Windows, usando objetos **TableAdapter** configurados para comunicarse con el archivo de la base de datos. Estos objetos **TableAdapter** colocaban una copia en caché de los datos de la base de datos en un objeto **DataSet**, que la aplicación usaba para acceder a esos datos. En este ejemplo utilizamos un enfoque similar. Un objeto **ObjectDataSource** puede encapsular a un objeto **TableAdapter**, y utilizar sus métodos para acceder a los datos en la base de datos. Esto ayuda a separar la lógica de acceso a los datos de la lógica de presentación. Como veremos en breve, las instrucciones de SQL que se utilizan para obtener datos no aparecen en la página ASPX cuando se utiliza un **ObjectDataSource**.

El primer paso para acceder a los datos mediante un **ObjectDataSource** es crear un **DataSet** que contenga los datos de la base de datos **Libros** que requiera la aplicación. En Visual C# 2005 Express, esto ocurre de manera automática cuando agregamos un origen de datos a un proyecto. Sin embargo, en Visual Web Developer hay que generar el objeto **DataSet** de manera explícita. Haga clic con el botón derecho del ratón en la ubicación del proyecto dentro del **Explorador de soluciones** y seleccione la opción **Agregar nuevo elemento....** En el cuadro de diálogo que aparezca, seleccione **DataSet** y especifique **LibrosDataSet.xsd** como el nombre de archivo; después haga clic en **Agregar**. A continuación aparecerá un cuadro de diálogo que le preguntará si el objeto **DataSet** debe colocarse en la carpeta **App_Code**: una carpeta cuyo contenido se compila y está disponible para todas las partes del proyecto. Haga clic en **Sí** para que el IDE cree esta carpeta y almacene **LibrosDataSet.xsd** ahí.

Paso 13: crear y configurar un objeto AuthorsTableAdapter

Una vez que se agregue el objeto **DataSet**, aparecerá el **Diseñador de DataSet** y se abrirá el **Asistente para la configuración de TableAdapter**. En el capítulo 20 vimos que este asistente le permite configurar un objeto **TableAdapter** para llenar un objeto **DataTable** en un conjunto **DataSet**, con los datos provenientes de una base de datos. La página **Libros.aspx** requiere dos conjuntos de datos: una lista de autores que se mostrarán en el control **DropDownList** de la página (que crearemos en breve) y una lista de libros escritos por un autor específico. Aquí nos enfocaremos en el primer conjunto de datos: los autores. Por ende, utilizaremos primero el **Asistente para la configuración de TableAdapter** para configurar un objeto **AutoresTableAdapter**. En el siguiente paso configuraremos un objeto **TitlesTableAdapter**.

En el **Asistente para la configuración de TableAdapter**, seleccione **Libros.mdf** de la lista desplegable. Después haga clic en **Siguiente >** dos veces para guardar la cadena de conexión en el archivo **Web.config** de la aplicación y avanzar a la pantalla **Elija un tipo de comando**.

En la pantalla **Elija un tipo de comando** del asistente, seleccione **Usar instrucciones SQL** y haga clic en **Siguiente >**. La siguiente pantalla le permite escribir una instrucción **SELECT** para recuperar datos de la base de datos, que después se colocarán en un objeto **DataTable** llamado **Autores**, dentro de **LibrosDataSet**. Escriba la siguiente instrucción SQL:

```
SELECT IDAutor, PrimerNombre + ' ' + ApellidoPaterno AS Nombre FROM Autores
```

en el cuadro de texto de la pantalla **Escriba una instrucción SQL**. Esta consulta selecciona el **IDAutor** de cada fila. El resultado de esta consulta también debe contener una columna llamada **Nombre**, que se crea mediante la concatenación del **PrimerNombre** y el **ApellidoPaterno** de cada fila, separados por un espacio. La palabra clave **AS** de SQL nos permite generar una columna en el resultado de una consulta (lo que se conoce como *alias*) que contiene el resultado de una expresión SQL (por ejemplo, **PrimerNombre + ' ' + ApellidoPaterno**). Pronto veremos cómo utilizar el resultado de esta consulta para llenar el control **DropDownList** con elementos que contengan los nombres completos de los autores.

Después de escribir la instrucción SQL, haga clic en el botón **Opciones avanzadas...** y desactive la opción **Generar instrucciones Insert, Update y Delete**, ya que esta aplicación no necesita modificar el contenido de la base de datos. Haga clic en **Aceptar** para cerrar el cuadro de diálogo **Opciones avanzadas**. Haga clic en **Siguiente >** para avanzar a la pantalla **Elija los métodos que se van a generar**. Deje los nombres predeterminados y haga clic en **Finalizar**. Observe que ahora el **Diseñador de DataSet** (figura 21.61) muestra un objeto **DataTable** llamado **Autores** con los miembros **IDAutor** y **Nombre**, y los métodos **Fill** y **GetData**.

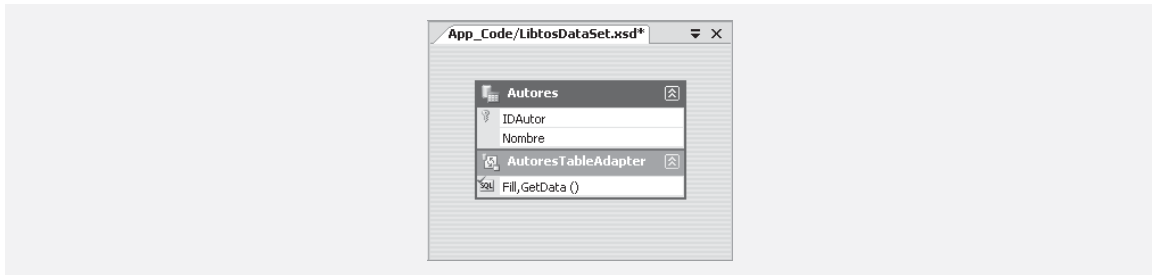


Figura 21.61 | El objeto DataTable Autores en el Diseñador de DataSet.

Paso 14: crear y configurar un objeto TitulosTableAdapter

Libros.aspx necesita acceder a una lista de libros escritos por un autor específico, y a una lista de autores. Por ende, debemos crear un objeto TitulosTableAdapter que recupere la información deseada de la tabla Titulos de la base de datos. Haga clic con el botón derecho del ratón en el Diseñador de DataSet y en el menú que aparezca, seleccione **Agregar > TableAdapter...** para iniciar el Asistente para la configuración de TableAdapter. Asegúrese que la cadena LibrosConnectionString esté seleccionada como la conexión en la primera pantalla del asistente, y después haga clic en **Siguiente >**. Elija la opción **Usar instrucciones SQL** y haga clic en **Siguiente >**.

En la pantalla **Escriba una instrucción SQL**, abra el cuadro de diálogo **Opciones avanzadas** y desactive la opción **Generar instrucciones Insert, Update y Delete**; después haga clic en **Aceptar**. Nuestra aplicación permite a los usuarios filtrar los libros mostrados por el nombre del autor, por lo que necesitamos crear una consulta que reciba un IDAutor como parámetro y devuelva las filas en la tabla Titulos para los libros escritos por ese autor. Para crear esta consulta compleja, haga clic en el botón **Generador de consultas...**

En el cuadro de diálogo **Agregar tabla** que aparezca, seleccione **ISBNAutor** y haga clic en **Agregar**. Después agregue también la tabla **Titulos**. Nuestra consulta requerirá acceso a los datos en ambas tablas. Haga clic en **Cerrar** para salir del cuadro de diálogo **Agregar tabla**. En el panel superior de la ventana **Generador de consultas** (figura 21.62), seleccione la casilla marcada como ***(Todas las columnas)** en la tabla **Titulos**. A continuación, en el panel de en medio agregue una fila con la propiedad **Columna** establecida en **ISBNAutor.IDAutor**. Desactive la casilla **Resultados**, ya que no queremos que IDAutor aparezca en el resultado de nuestra consulta. Agregue un parámetro @IDAutor en la columna **Filtro** de la fila recién agregada. La instrucción SQL generada por estas acciones obtendrá

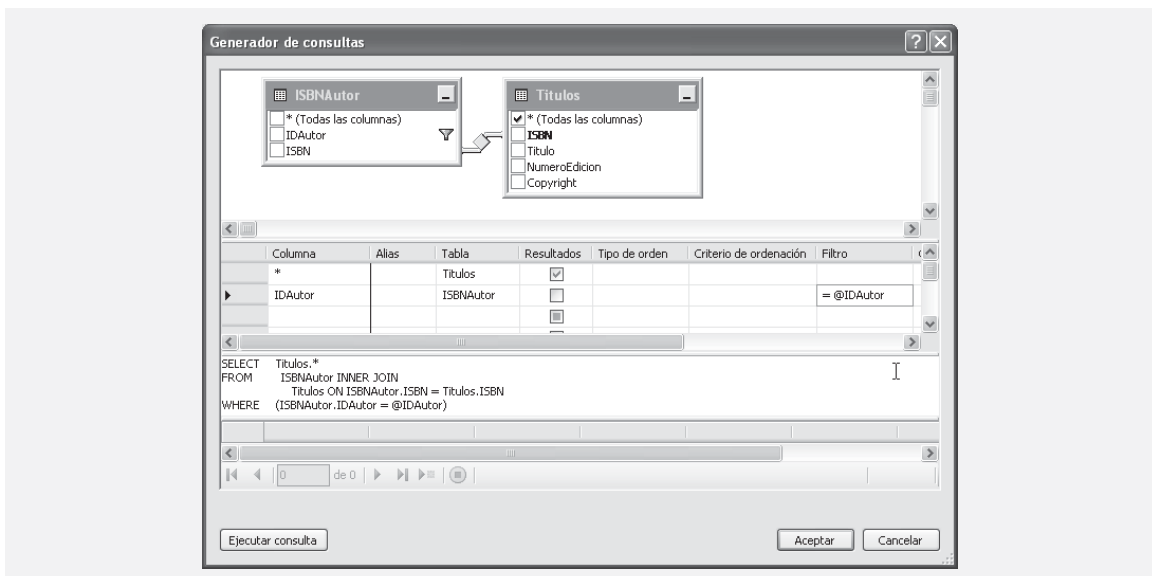


Figura 21.62 | El Generador de consultas para diseñar una consulta que seleccione los libros escritos por un autor específico.

información acerca de todos los libros escritos por el autor especificado mediante el parámetro @IDautor. La instrucción primero mezcla los datos de las tablas ISBNAutor y Titulos. La cláusula INNER JOIN especifica que se deben comparar las columnas ISBN de cada tabla, para determinar cuáles filas se van a mezclar. La operación INNER JOIN produce una tabla temporal que contiene las columnas de ambas tablas. La porción exterior de la instrucción SQL selecciona la información de los libros de esta tabla temporal, para un autor específico (es decir, todas las filas en las que la columna IDAutor es igual a @IDautor).

Haga clic en **Aceptar** para salir del **Generador de consultas** y después haga clic en el botón **Siguiente >** del **Asistente para la configuración de TableAdapter**. En la pantalla **Elija los métodos que se van a generar**, escriba **LlenarPorIDAutor** y **ObtenerDatosPorIDAutor** como los nombres de los dos métodos que se van a generar para **TitulosTableAdapter**. Haga clic en **Finalizar** para salir del asistente. Ahora deberá ver un objeto **DataTables** llamado **Titulo** en el **Diseñador de DataSet** (figura 21.63).

Paso 15: agregar un control DropDownList que contenga el primer nombre y el apellido de los autores

Ahora que hemos creado un objeto **LibrosDataSet** y que configuramos los objetos **TableAdapter** necesarios, agregaremos controles a **Libros.aspx** para mostrar los datos en la página Web. Primero agregaremos el control **DropDownListBox**, del cual los usuarios pueden seleccionar un autor. Abra **Libros.aspx** en modo **Diseño** y después agregue el texto **Autor:** y un control **DropDownList** llamado **autoresDropDownList** en el control **Content** de la página, debajo del contenido existente. Al principio, el control **DropDownList** muestra el texto **[Sin enlazar]**. Ahora enlazaremos la lista a un origen de datos, por lo que ésta mostrará la información del autor, que el objeto **AutoresTableAdapter** colocó en el objeto **LibrosDataSet**. En el menú de etiquetas inteligentes **Tareas de DropDownList**, haga clic en **Elegir origen de datos...** para iniciar el **Asistente para la configuración de orígenes de datos**. Seleccione **<Nuevo origen de datos...>** de la lista desplegable **Seleccionar un origen de datos** en la primera pantalla del asistente. Al hacer esto, se abre la pantalla **Elija un tipo de origen de datos**. Seleccione **Objeto** y establezca la ID a **autoresObjectDataSource**; después haga clic en **Aceptar**.

Un objeto **ObjectDataSource** accede a los datos a través de otro objeto, que a menudo se le conoce como **objeto comercial**. En la sección 21.3 vimos que el nivel intermedio de una aplicación de tres niveles contiene la lógica comercial que controla la forma en que la interfaz de usuario del nivel superior de una aplicación (en este caso, **Libros.aspx**) accede a los datos del nivel inferior (en este caso, el archivo de base de datos **Libros.mdf**). Por lo tanto, un objeto comercial representa el nivel medio de una aplicación, y media las interacciones entre los otros dos niveles. En una aplicación Web ASP.NET, es común que un objeto **TableAdapter** sirva como el objeto comercial que obtiene los datos de la base de datos de nivel inferior, y los pone a disposición de la interfaz de usuario de nivel superior, a través de un objeto **DataSet**. En la pantalla **Elegir un objeto comercial** del asistente **Configurar origen de datos** (figura 21.64), seleccione **LibrosDataSetTableAdapters.AutoresTableAdapter**. [Nota: Tal vez necesite guardar el proyecto para ver el objeto **AutoresTableAdapter**.] **LibrosDataSetTableAdapters** es un espacio de nombres que declara el IDE cuando usted crea a **LibrosDataSet**. Haga clic en **Siguiente >** para continuar.

La pantalla **Definir métodos de datos** (figura 21.65) nos permite especificar cuál método del objeto comercial (en este caso, **AutoresTableAdapter**) debe utilizarse para obtener los datos a los que se accede a través del objeto **ObjectDataSource**. Sólo se pueden elegir métodos que devuelvan datos, por lo que la única opción que se proporciona es el método **GetData**, que devuelve un objeto **AutoresDataTable**. Haga clic en **Finalizar** para

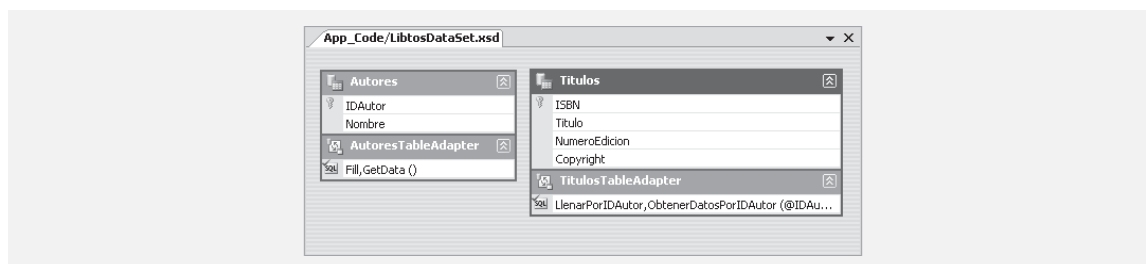


Figura 21.63 | El **Diseñador de DataSet**, después de agregar el objeto **TitulosTableAdapter**.

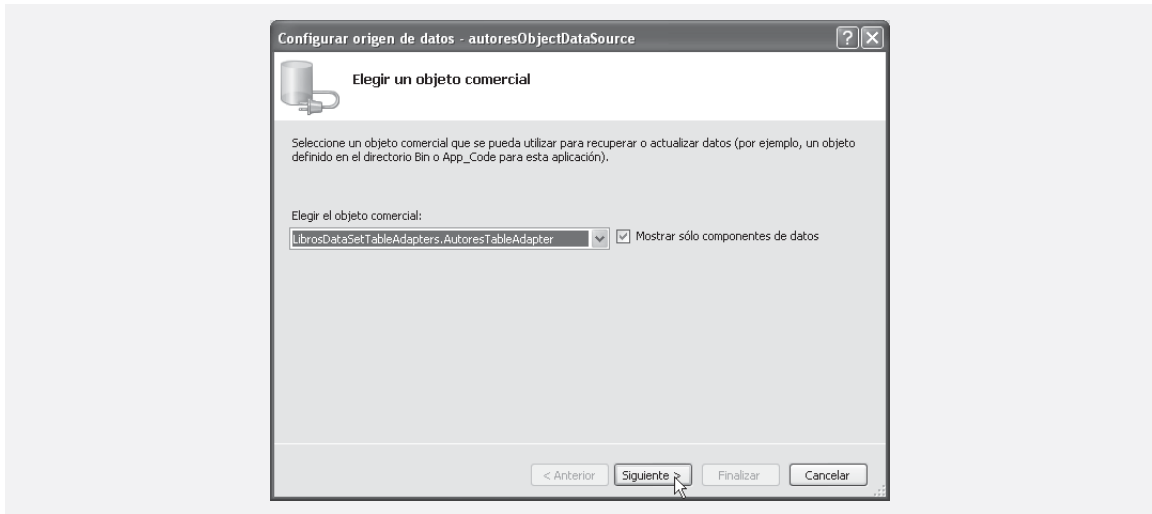


Figura 21.64 | Elegir un objeto comercial para un ObjectDataSource.

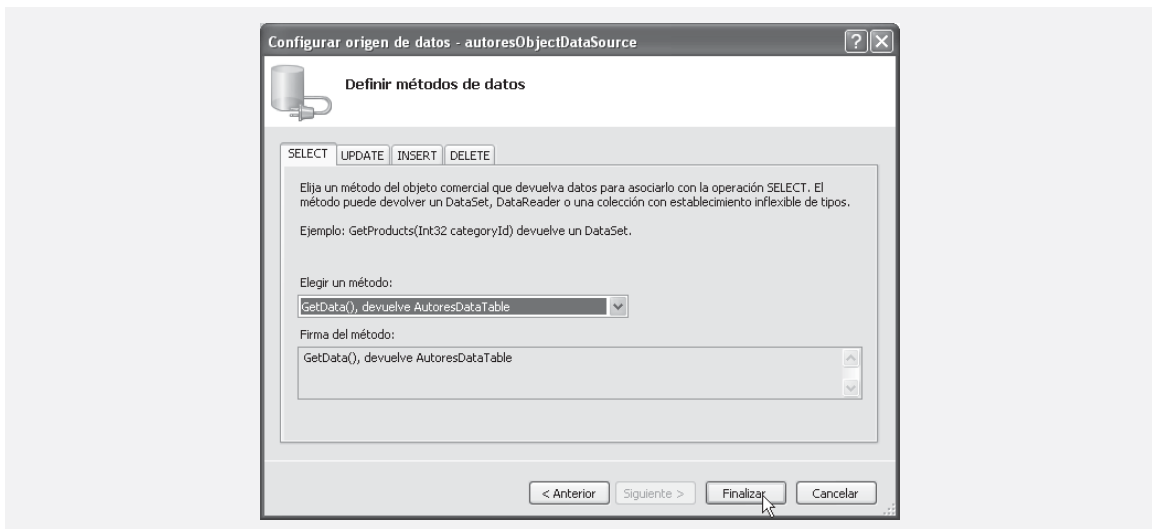


Figura 21.65 | Elegir un método de datos de un objeto comercial, para usarlo con un ObjectDataSource.

cerrar el asistente **Configurar origen de datos** y regresar al **Asistente para la configuración de orígenes de datos** para el control `DropDownList` (figura 21.66). El origen de datos recién creado (es decir, `autoresObjectDataSource`) deberá estar seleccionado en la lista desplegable superior. Las otras dos listas desplegables en esta pantalla le permiten configurar la forma en que el control `DropDownList` utiliza los datos del origen de datos. Establezca `Nombre` como el campo de datos para mostrar, e `IDAutor` como el campo de datos para usarlo como valor. Así, cuando se represente `autoresDropDownList` en un explorador Web, los elementos de la lista mostrarán los nombres de los autores, pero los valores subyacentes asociados con cada elemento serán los campos `IDAutor` de los autores. Por último, haga clic en **Aceptar** para enlazar el control `DropDownList` con los datos especificados.

El último paso para configurar el control `DropDownList` en `Libros.aspx` es establecer la propiedad **AutoPostBack** del control a `True`. Esta propiedad indica que se produce un postback cada vez que el usuario selecciona un elemento en el control `DropDownList`. Como veremos en breve, esto hace que el control `GridView` (que crearemos en el siguiente paso) de la página muestre nuevos datos.

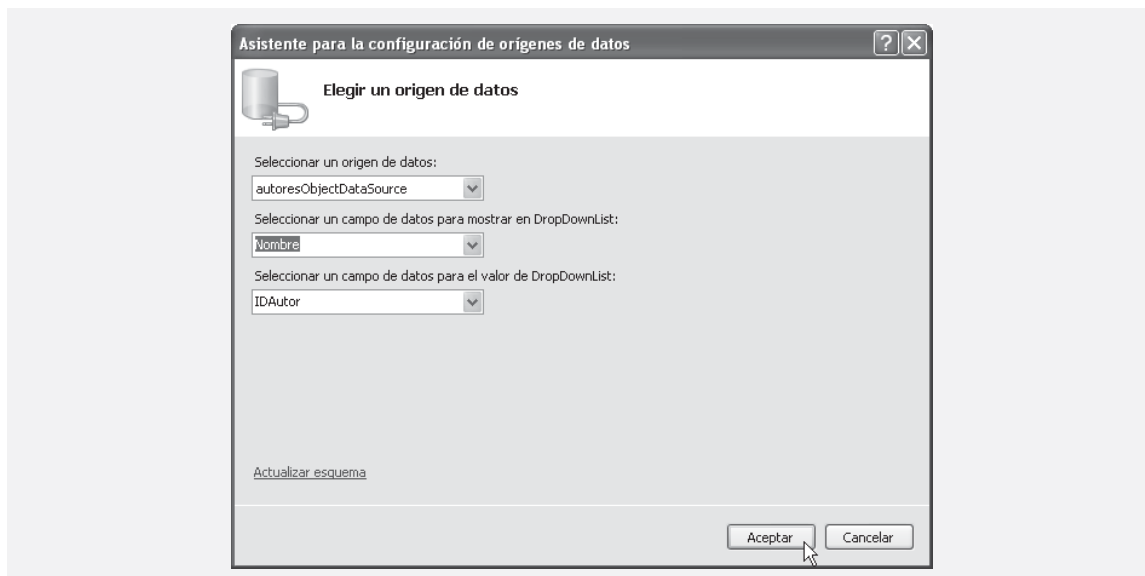


Figura 21.66 | Elegir un origen de datos para un control DropDownList.

Paso 16: crear un control GridView para mostrar los libros de los autores seleccionados

Ahora agregaremos un control GridView a Libros.aspx, para mostrar la información de los libros según el autor seleccionado en el control autoresDropDownList. Agregue un control GridView llamado titulosGridView debajo de los demás controles que se encuentran en el control Content de la página.

Para enlazar el control GridView a los datos de la base de datos Libros, seleccione <Nuevo origen de datos...> de la lista desplegable **Elegir origen de datos** en el menú de etiquetas inteligentes **Tareas de GridView**. Cuando se abra el **Asistente para la configuración de orígenes de datos**, seleccione **Objeto** y establezca el ID del origen de datos a titulosObjectDataSource; después haga clic en **Aceptar**. En la pantalla **Elija un objeto comercial**, seleccione LibrosDataSetTableAdapters.TitulosTableAdapter de la lista desplegable, para indicar el objeto que se utilizará para acceder a los datos. Haga clic en **Siguiente >**. En la pantalla **Definir métodos de datos**, deje la selección predeterminada de ObtenerDatosPorIDAutor como el método que se invocará para obtener los datos y mostrarlos en el control GridView. Haga clic en **Siguiente >**.

Recuerde que el método ObtenerDatosPorIDAutor de TitulosTableAdapter requiere un parámetro para indicar el IDAutor para el que deben obtenerse los datos. La pantalla **Definir parámetros** (figura 21.67) le permite especificar en dónde obtener el valor del parámetro @IDAutor en la instrucción SQL ejecutada por ObtenerDatosPorAutorID. Seleccione **Control** de la lista desplegable **Origen del parámetro**. Seleccione autoresDropDownList como el **IDControl** (es decir, el ID del control de origen del parámetro). A continuación, escriba 1 en **DefaultValue** para que se muestren los libros escritos por Harvey Deitel (que tiene el IDAutor 1 en la base de datos) la primera vez que se cargue la página (es decir, antes de que el usuario haya realizado alguna selección usando el control autoresDropDownList). Por último, haga clic en **Finalizar** para salir del asistente. Ahora el control GridView está configurado para mostrar los datos obtenidos por TitulosTableAdapter. ObtenerDatosPorIDAutorID, usando el valor de la selección actual en autoresDropDownList como parámetro. Así, cuando el usuario seleccione un nuevo autor y ocurra un postback, el control GridView mostrará un nuevo conjunto de datos.

Ahora que el control GridView está enlazado a un origen de datos, modificaremos varias de sus propiedades para ajustar su apariencia y comportamiento. Establezca la propiedad CellPadding del control GridView a 5, establezca la propiedad BackColor de AlternatingRowStyle a LightYellow y establezca la propiedad BackColor de HeaderStyle a LightGreen. Cambie la anchura (Width) del control a 600px, para poder alojar valores de datos extensos.

A continuación, en el menú de etiquetas inteligentes **Tareas de GridView**, seleccione la opción **Habilitar ordenación**. Esto hace que los encabezados de las columnas en el control GridView se conviertan en hipervínculos,

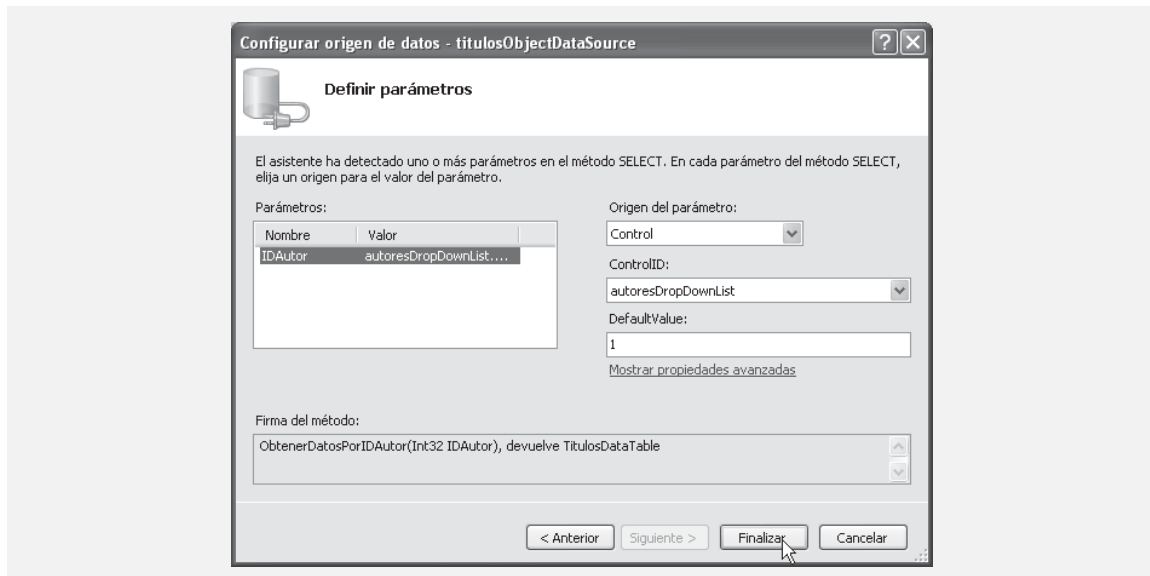


Figura 21.67 | Elegir el origen de datos para un parámetro en el método de datos del objeto comercial.

para permitir a los usuarios ordenar los datos. Por ejemplo, al hacer clic en el encabezado `Titles` en el explorador Web, los datos mostrados aparecerán ordenados alfabéticamente. Si hacemos clic en este encabezado por segunda vez, los datos se mostrarán en orden alfabético inverso. ASP.NET oculta los detalles requeridos para lograr esta funcionalidad.

Por último, en el menú de etiquetas inteligentes **Tareas de GridView**, seleccione la opción **Habilitar paginación**. Esto hace que el control GridView se divida en varias páginas. El usuario puede hacer clic en los vínculos numerados en la parte inferior del control GridView para mostrar una página distinta de datos. La propiedad **PageSize** de GridView determina el número de entradas por página. Establezca la propiedad **PageSize** a 4, usando la ventana **Propiedades**, de manera que el control GridView sólo muestre cuatro libros por página. Esta técnica para mostrar los datos aumenta la legibilidad del sitio y permite que las páginas se carguen más rápido (debido a que se muestran menos datos a la vez). Observe que, al igual que con el proceso de ordenar los datos en un control GridView, no necesitamos agregar código para lograr la funcionalidad de la paginación. La figura 21.68 muestra el archivo `Libros.aspx` completo en modo **Diseño**.

Paso 17: examinar el marcado en `Libros.aspx`

La figura 21.69 presenta el marcado en `Libros.aspx` (cambiamos el formato por cuestión de legibilidad). Aparte del signo de admiración en la línea 9, que agregamos de forma manual en modo **Código**, el resto del marcado lo generó el IDE, en respuesta a las acciones que realizamos en modo **Diseño**. El control `Content` (líneas 6-55) define el contenido específico para la página que sustituirá al control `ContentPlaceHolder` llamado `cuerpoContent`. Recuerde que este control se encuentra en la página maestra que especificamos en la línea 3. La línea 9 crea el control `LoginName`, que muestra el nombre del usuario autenticado cuando solicitamos y vemos la página desde un explorador. Las líneas 10-12 crean el control `LoginStatus`. Recuerde que este control está configurado para redirigir al usuario a la página de inicio de sesión después de cerrar la sesión (es decir, hacer clic en el hipervínculo con la propiedad `LogoutText`).

Las líneas 15-18 definen el control `DropDownList` que muestra los nombres de los autores en la base de datos `Libros`. La línea 16 contiene la propiedad `AutoPostBack` del control, la cual indica que si se cambia el elemento seleccionado en la lista, se produce un postback. La propiedad `DataSourceID` en la línea 16 especifica que los elementos del control `DropDownList` se crean con base en los datos que se obtuvieron a través del objeto `autoresObjectDataSource` (definido en las líneas 20-23). La línea 21 especifica que este objeto `ObjectDataSource` accede a la base de datos `Libros` mediante una llamada al método `GetData` del objeto `AutoresTableAdapter` de `LibrosDataSet` (línea 22).

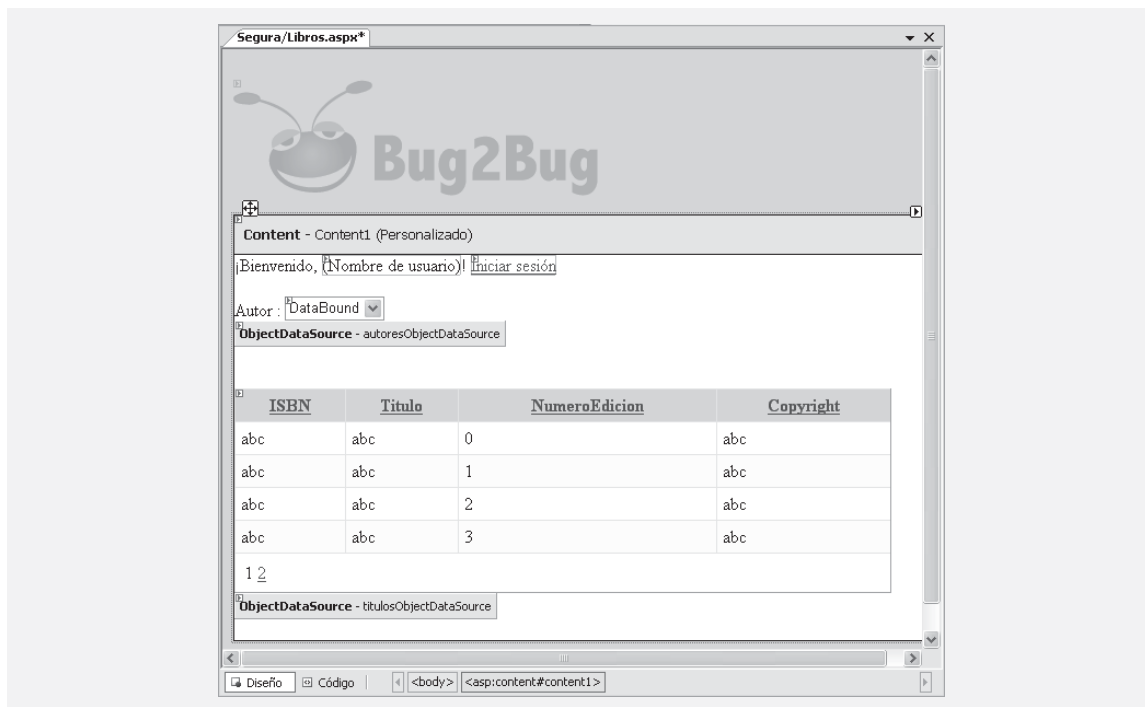


Figura 21.68 | La página Libros.aspx completa, en modo Diseño.

Las líneas 25-43 crean el control GridView que muestra información acerca de los libros escritos por el autor seleccionado. La etiqueta inicial (líneas 25-28) indica que el control GridView tiene habilitadas la paginación (con un tamaño de página de 4) y la ordenación. La propiedad `AutoGenerateColumns` indica si las columnas en el control GridView se generan en tiempo de ejecución, con base en los campos en el origen de

```

1  <!-- Fig. 21.69: Libros.aspx -->
2  <!-- Muestra información de la base de datos Libros. -->
3  <%@ Page Language="C#" MasterPageFile="~/Bug2Bug.master"
4     Title="Información sobre libros" %>
5
6  <asp:Content ID="Content1" ContentPlaceHolderID="cuerpoContent"
7     Runat="Server">
8
9     ¡Bienvenido, <asp:LoginName ID="LoginName1" runat="server" />!
10    <asp:LoginStatus ID="LoginStatus1" runat="server"
11        LogoutAction="RedirectToLoginPage"
12        LogoutText="Haga clic aquí para cerrar sesión" /><br /><br />
13
14    Autor:
15    <asp:DropDownList ID="autoresDropDownList" runat="server"
16        AutoPostBack="True" DataSourceID="autoresObjectDataSource"
17        DataTextField="Nombre" DataValueField="IDAutor">
18    </asp:DropDownList>
19
20    <asp:ObjectDataSource ID="autoresObjectDataSource"
21        runat="server" SelectMethod="GetData"
22        TypeName="LibrosDataSetTableAdapters.AutoresTableAdapter">

```

Figura 21.69 | Marcado para el archivo Libros.aspx completo. (Parte 1 de 3).

```

23 </asp:ObjectDataSource><br /><br />
24
25 <asp:GridView ID="titulosGridView" runat="server" AllowPaging="True"
26   AllowSorting="True" AutoGenerateColumns="False" CellPadding="5"
27   DataKeyNames="ISBN" DataSourceID="titulosObjectDataSource"
28   PageSize="4" Width="600px">
29
30   <Columns>
31     <asp:BoundField DataField="ISBN"
32       HeaderText="ISBN" ReadOnly="True" SortExpression="ISBN" />
33     <asp:BoundField DataField="Titulo"
34       HeaderText="Titulo" SortExpression="Titulo" />
35     <asp:BoundField DataField="NumeroEdicion"
36       HeaderText="NumeroEdicion" SortExpression="NumeroEdicion" />
37     <asp:BoundField DataField="Copyright"
38       HeaderText="Copyright" SortExpression="Copyright" />
39   </Columns>
40
41   <HeaderStyle BackColor="LightGreen" />
42   <AlternatingRowStyle BackColor="LightYellow" />
43 </asp:GridView>
44
45 <asp:ObjectDataSource ID="titulosObjectDataSource" runat="server"
46   SelectMethod="ObtenerDatosPorIDAutor"
47   TypeName="LibrosDataSetTableAdapters.TitulosTableAdapter">
48
49   <SelectParameters>
50     <asp:ControlParameter ControlID="autoresDropDownList"
51       DefaultValue="1" Name="IDAutor"
52       PropertyName="SelectedValue" Type="Int32" />
53   </SelectParameters>
54 </asp:ObjectDataSource>
55 </asp:Content>

```

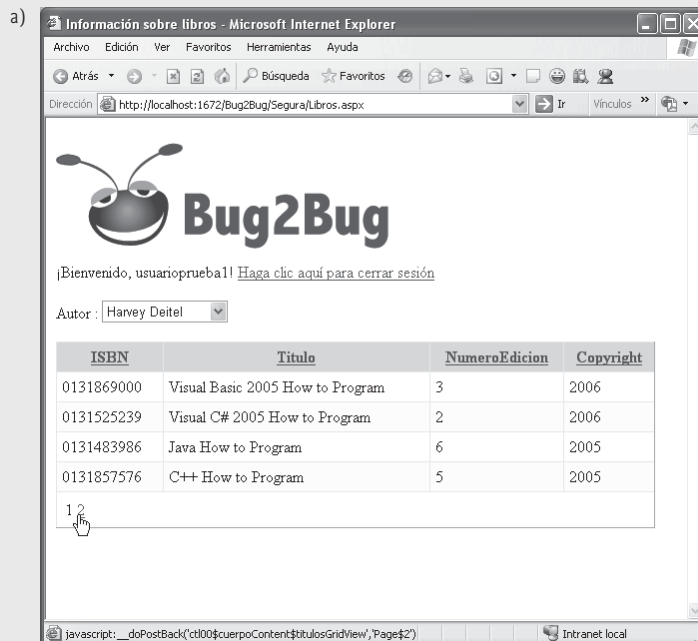


Figura 21.69 | Marcado para el archivo Libros.aspx completo. (Parte 2 de 3).

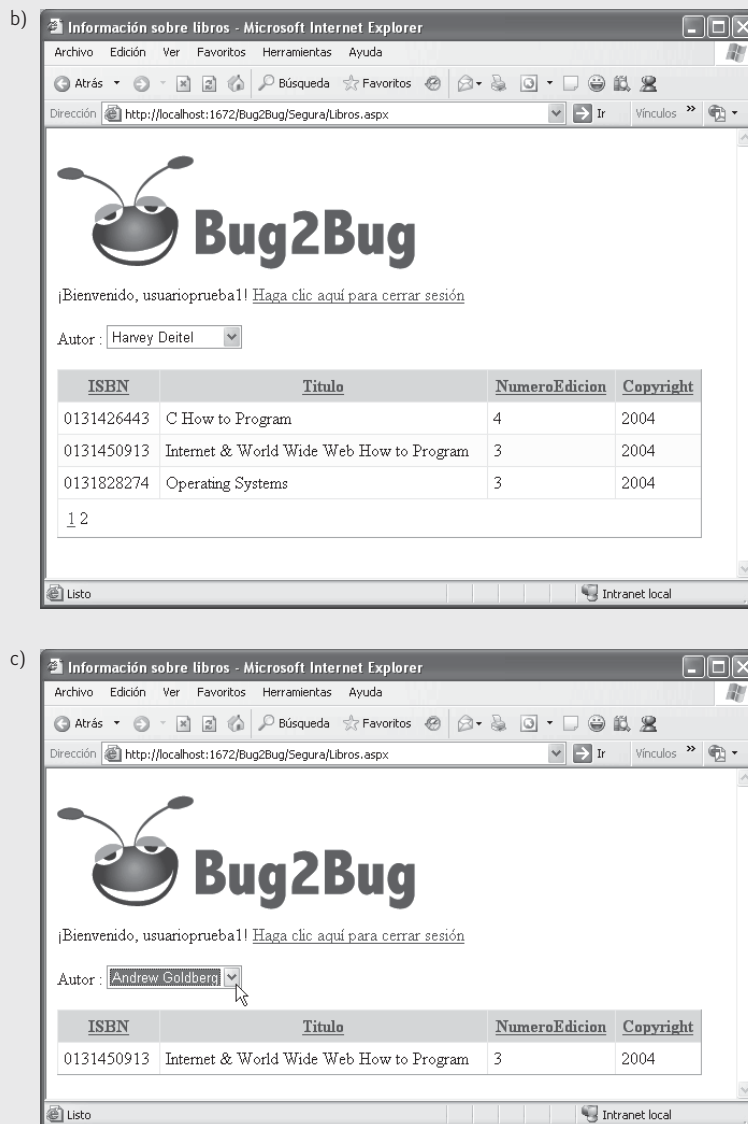


Figura 21.69 | Marcado para el archivo Libros.aspx completo. (Parte 3 de 3).

datos. Esta propiedad se estableció en False, debido a que el elemento Columns generado por el IDE (líneas 30-39) ya especifica las columnas para el control GridView, mediante el uso de elementos BoundField. Las líneas 45-54 definen el objeto ObjectDataSource que se utilizará para llenar el control GridView de datos. Recuerde que configuramos a titulosObjectDataSource para que utilizara el método ObtenerDatosPorIDAutor del objeto TitlesTableAdapter de LibrosDataSet para este fin. El elemento ControlParameter en las líneas 50-52 especifica que el valor del parámetro del método ObtenerDatosPorIDAutor proviene de la propiedad SelectedValue de autoresDropDownList.

La figura 21.69(a) ilustra la apariencia predeterminada de Libros.aspx en un explorador Web. Como la propiedad DefaultValue (línea 51) del elemento ControlParameter para el objeto titulosObjectDataSource es 1, cuando se cargue la página por primera vez, se mostrarán los libros escritos por el autor con el IDAutor de 1 (es decir, Harvey Deitel). Observe que el control GridView muestra los vínculos de paginación

debajo de los datos, ya que el número de filas de datos devueltas por `ObtenerDatosPorIDAutor` es mayor que el tamaño de la página. La figura 21.69(b) muestra el control `GridView` después de hacer clic en el vínculo 2 para ver la segunda página de datos. La figura 21.69(c) presenta a `Libros.aspx` después de que el usuario selecciona un autor distinto de `autoresDropDownList`. Los datos se ajustan a una página, por lo que el control `GridView` no muestra los vínculos de paginación.

21.9 Conclusión

En este capítulo presentamos el desarrollo de aplicaciones Web mediante ASP.NET y Visual Web Developer 2005 Express. Primero hablamos sobre las transacciones HTTP simples que se llevan a cabo al solicitar y recibir una página Web a través de un explorador Web. Después aprendió acerca de los tres niveles (es decir, el cliente o nivel superior, la lógica comercial o nivel medio y la información o nivel inferior) que componen la mayoría de las aplicaciones Web.

Después explicamos la función de los archivos ASPX (es decir, archivos Web Forms) y los archivos de código subyacente, y la relación entre ellos. Hablamos sobre cómo ASP.NET compila y ejecuta aplicaciones Web, de manera que se puedan mostrar como XHTML en un explorador Web. También aprendió a crear una aplicación Web ASP.NET, usando el IDE de Visual Web Developer.

En este capítulo se demostraron varios controles Web comunes de ASP.NET, que se utilizan para mostrar texto e imágenes en un formulario Web Forms. Aprendió a utilizar un control `AdRotator` para mostrar imágenes seleccionadas al azar. También hablamos sobre los controles de validación, que le permiten asegurar que la entrada del usuario en una página Web cumpla con ciertos requerimientos.

Hablamos sobre los beneficios de mantener la información de estado acerca de un usuario, a través de varias páginas de un sitio Web. Después le mostramos cómo puede incluir dicha funcionalidad en una aplicación Web, usando cookies o rastreo de sesiones con objetos `HttpSessionState`.

Por último, el capítulo presentó dos casos de estudio acerca de la creación de aplicaciones ASP.NET que interactúan con bases de datos. Primero le mostramos cómo crear una aplicación de libro de visitantes, la cual permite a los usuarios enviar comentarios acerca de un sitio Web. Aprendió a guardar la entrada del usuario en una base de datos SQL y cómo mostrar la información de usuarios anteriores en la página Web.

El segundo caso de estudio demostró cómo crear una aplicación que requiere que los usuarios inicien sesión antes de acceder a la información de la base de datos `Libros`, que vimos en el capítulo 20. Utilizó la **Herramienta Administración de sitios Web** para configurar la aplicación y utilizar autenticación de formularios, para evitar que los usuarios anónimos accedan a la información sobre los libros. Este caso de estudio le explicó cómo utilizar los controles `Login`, `CreateUserWizard`, `LoginName` y `LoginStatus` para simplificar la autenticación de usuarios. También aprendió a crear una apariencia visual uniforme para un sitio Web, usando una página principal y varias páginas de contenido.

En el siguiente capítulo continuaremos nuestra cobertura sobre la tecnología ASP.NET, con una introducción a los servicios Web, que permiten que los métodos en un equipo llamen a los métodos en otro equipo a través de formatos de datos y protocolos comunes, tales como XML y HTTP. Aprenderá cómo los servicios Web promueven la reutilización de software y la interoperabilidad a través de varias computadoras en una red como Internet.

21.10 Recursos Web

beta.asp.net

Este sitio oficial de Microsoft presenta las generalidades acerca de ASP.NET y proporciona un vínculo para descargar Visual Web Developer. Este sitio incluye artículos sobre ASP.NET, vínculos a recursos útiles de ASP.NET y listas de libros acerca del desarrollo Web mediante ASP.NET.

beta.asp.net/QuickStartv20/aspnet/

El Tutorial de inicio rápido sobre ASP.NET de Microsoft proporciona ejemplos de código y discusiones sobre los temas fundamentales de ASP.NET.

beta.asp.net/guidedtour2/

Este paseo con guía sobre Visual Web Developer 2005 Express introduce las características clave en el IDE que se utiliza para desarrollar aplicaciones Web ASP.NET.

www.15seconds.com

Este sitio ofrece noticias, artículos, ejemplos de código, FAQs y vínculos a valiosos recursos comunitarios sobre ASP.NET, como un foro de mensajes ASP.NET y una lista de correo.

aspalliance.com

Este sitio comunitario contiene artículos, tutoriales y ejemplos acerca de ASP.NET.

aspadvice.com

Este sitio proporciona acceso a muchas listas de correo electrónico, en las que cualquiera puede hacer y responder preguntas acerca de ASP.NET y las tecnologías relacionadas.

www.asp101.com/aspdotnet/

Este sitio presenta las generalidades acerca de ASP.NET, e incluye artículos, ejemplos de código, un foro de discusión y vínculos a recursos sobre ASP.NET. Los ejemplos de código se basan en muchas de las técnicas que presentamos en este capítulo, como el rastreo de sesiones y la conexión a una base de datos.

www.411asp.net

Este sitio de recursos ofrece a los programadores tutoriales y ejemplos de código sobre ASP.NET. Las páginas comunitarias permiten a los programadores hacer preguntas, responder preguntas y publicar mensajes.

www.123aspx.com

Este sitio ofrece un directorio de vínculos a recursos sobre ASP.NET. El sitio incluye también boletines de noticias diarios y semanales.

22

Servicios Web

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Qué es un servicio Web.
- Cómo crear servicios Web.
- La parte importante que juegan XML y el Protocolo simple de acceso a objetos basado en XML para habilitar los servicios Web.
- Los elementos que conforman los servicios Web, como las descripciones del servicio y los archivos de descubrimiento.
- Cómo crear un cliente que utilice un servicio Web.
- Cómo utilizar servicios Web con aplicaciones para Windows y aplicaciones Web.
- Cómo utilizar el rastreo de sesiones en los servicios Web, para mantener la información de estado para el cliente.
- Cómo pasar tipos definidos por el usuario a un servicio Web.

*Un cliente para mí
es una simple unidad,
un factor en un problema.*

—Sir Arthur Conan Doyle

*...si las cosas más simples
de la naturaleza tienen
un mensaje que comprendas,
regocíjate, por que tu alma
está viva.*

—Eleonora Duse

El protocolo es todo.

—Francoise Giuliani

*Ellos también sirven
a quien sólo está esperando.*

—John Milton

- 22.1 Introducción
- 22.2 Fundamentos de los servicios Web .NET
 - 22.2.1 Creación de un servicio Web en Visual Web Developer
 - 22.2.2 Descubrimiento de servicios Web
 - 22.2.3 Determinación de la funcionalidad de un servicio Web
 - 22.2.4 Prueba de los métodos de un servicio Web
 - 22.2.5 Creación de un cliente para utilizar un servicio Web
- 22.3 Protocolo simple de acceso a objetos (SOAP)
- 22.4 Publicación y consumo de los servicios Web
 - 22.4.1 Definición del servicio Web EnteroEnorme
 - 22.4.2 Creación de un servicio Web en Visual Web Developer
 - 22.4.3 Despliegue del servicio Web EnteroEnorme
 - 22.4.4 Creación de un cliente para consumir el servicio Web EnteroEnorme
 - 22.4.5 Consumo del servicio Web EnteroEnorme
- 22.5 Rastreo de sesiones en los servicios Web
 - 22.5.1 Creación de un servicio Web de Blackjack
 - 22.5.2 Consumo del servicio Web de Blackjack
- 22.6 Uso de formularios Web Forms y servicios Web
 - 22.6.1 Agregar componentes de datos a un servicio Web
 - 22.6.2 Creación de un formulario Web Forms para interactuar con el servicio Web de reservaciones de una aerolínea
- 22.7 Tipos definidos por el usuario en los servicios Web
- 22.8 Conclusión
- 22.9 Recursos Web

22.1 Introducción

En este capítulo se introducen los servicios Web, que promueven la reutilización de software en sistemas distribuidos, en los que las aplicaciones se ejecutan a través de varias computadoras en una red. Un *servicio Web* es una clase que permite que sus métodos sean llamados por otros métodos en otros equipos, a través de formatos de datos y protocolos comunes, como XML (vea el capítulo 19) y HTTP. En .NET, las llamadas a los métodos a través de la red se implementan comúnmente a través del *Protocolo simple de acceso a objetos (SOAP)*, un protocolo basado en XML que describe cómo marcar solicitudes y respuestas, de manera que puedan transferirse a través de protocolos como HTTP. Mediante el uso de SOAP, las aplicaciones representan y transmiten datos en un formato estandarizado, basado en XML.

Microsoft está alentando a los distribuidores de software y comercios electrónicos a que desplieguen servicios Web. Como cada vez hay más organizaciones a nivel mundial que se conectan a Internet, el concepto de aplicaciones que llaman a métodos a través de una red se ha vuelto más práctico. Los servicios Web representan el siguiente paso en la programación orientada a objetos; en vez de desarrollar software a partir de un pequeño número de bibliotecas de clases que se proporcionan en una ubicación, los programadores pueden acceder a las bibliotecas de clases de un servicio Web que estén distribuidas en todo el mundo.



Tip de rendimiento 22.1

Los servicios Web no son la mejor solución para ciertas aplicaciones de rendimiento intensivo, ya que las aplicaciones que invocan a los servicios Web experimentan retrasos en la red. Además, las transferencias de datos son por lo general más extensas, ya que los datos se transmiten en formatos XML basados en texto.

Los servicios Web facilitan la colaboración y permiten que los comercios crezcan. Al comprar servicios Web y utilizar algunos gratuitos en la red que son relevantes para sus comercios, las compañías pueden invertir menos tiempo desarrollando nuevas aplicaciones. Los comercios electrónicos pueden usar servicio Web para proporcionar a sus clientes una mejor experiencia de compra. Por ejemplo, considere una tienda de música en línea. El sitio Web de la tienda proporciona vínculos a información acerca de varios CDs, para permitir al usuario comprar los CDs o ver información acerca de los artistas. Otra compañía que vende boletos para los conciertos proporciona un servicio Web que muestra las próximas fechas de conciertos de varios artistas, para permitir a los usuarios comprar los boletos. Al consumir el servicio Web de boletos de conciertos en su sitio, la tienda de música en línea puede ofrecer un servicio adicional a sus clientes e incrementar el tráfico de su sitio. La compañía que vende boletos de conciertos también se beneficia de la relación comercial, al vender más boletos y posiblemente recibir ingresos de la tienda de música en línea, por el uso de su servicio Web. Muchos servicios Web se proporcionan sin costo. Por ejemplo, Amazon y Google ofrecen servicios Web gratuitos, que usted puede utilizar en sus propias aplicaciones para acceder a la información que proporcionan.

Visual Web Developer y el .NET Framework proporcionan una manera simple y amigable para el usuario, para crear servicios Web. En este capítulo le mostraremos cómo emplear estas herramientas para crear, desplegar y utilizar servicios Web. Para cada ejemplo, proporcionamos el código para el servicio Web y después presentamos una aplicación que utiliza este servicio Web. Nuestros primeros ejemplos analizan los servicios Web y cómo funcionan en Visual Web Developer. Después mostramos servicios Web que utilizan características más sofisticadas, como el rastreo de sesiones (que vimos en el capítulo 21) y la manipulación de objetos de tipos definidos por el usuario.

Al igual que en el capítulo 21, en este capítulo haremos distinciones entre Visual C# 2005 Express y Visual Web Developer 2005. Crearemos los servicios Web en Visual Web Developer y crearemos aplicaciones cliente que utilicen estos servicios Web mediante el uso de Visual C# 2005 y de Visual Web Developer 2005. La versión completa de Visual Studio 2005 incluye la funcionalidad de ambas ediciones Express.

22.2 Fundamentos de los servicios Web .NET

Un servicio Web es un componente de software que se almacena en un equipo, al que una aplicación (o cualquier otro componente de software) puede acceder desde otro equipo, a través de una red. El equipo en el que reside el servicio Web se conoce como *equipo remoto*. La aplicación (es decir, el cliente) que accede al servicio Web envía la llamada a un método a través de una red hasta el equipo remoto, el cual procesa la llamada y devuelve una respuesta a la aplicación, a través de la red. Este tipo de computación distribuida beneficia a varios sistemas. Por ejemplo, una aplicación sin acceso directo a ciertos datos en otro sistema podría obtener estos datos a través de un servicio Web. De manera similar, una aplicación que carezca del poder de procesamiento necesario para realizar cálculos específicos podría utilizar un servicio Web para aprovechar los recursos superiores de otro sistema.

Por lo general, un servicio Web se implementa como una clase. En capítulos anteriores incluimos una clase en un proyecto, ya sea definiendo la clase en el proyecto, o agregando una referencia a una DLL compilada. Todas las piezas de una aplicación residen en un equipo. Cuando un cliente utiliza un servicio Web, la clase (y su DLL compilada) se almacena en un equipo remoto; una versión compilada de la clase del servicio Web no se coloca en el directorio de la aplicación actual. En breve hablaremos sobre lo que ocurre.

Las solicitudes y respuestas relacionadas con los servicios Web que se crean mediante Visual Web Developer se transmiten, por lo general, a través de SOAP. Así, cualquier cliente capaz de generar y procesar mensajes SOAP puede interactuar con un servicio Web, sin importar el lenguaje en el que esté escrito. En la sección 22.3 hablaremos más acerca de SOAP.

Es posible para los servicios Web limitar el acceso a los clientes autorizados. Al final del capítulo encontrará los Recursos Web, en donde podrá ver vínculos a información acerca de los mecanismos y protocolos estándar que tratan con las cuestiones de seguridad de los servicios Web.

Los servicios Web tienen implicaciones importantes para las *transacciones de negocio a negocio (B2B)*. Estos servicios permiten a los comercios realizar transacciones a través de servicios Web estandarizados y con una disponibilidad amplia, en vez de depender de aplicaciones propietarias. Los servicios Web y SOAP son independientes de la plataforma y el lenguaje, por lo que las compañías pueden colaborar mediante los servicios Web, sin preocuparse acerca de la compatibilidad de su hardware, software y tecnologías de comunicaciones. Las compañías como Amazon, Google, eBay y muchas otras están utilizando servicios Web para su provecho. Para leer casos de estudio acerca del uso de los servicios Web en las empresas, visite msdn.microsoft.com/webservices/understanding/casestudies/default.aspx.

22.2.1 Creación de un servicio Web en Visual Web Developer

Para crear un servicio Web en Visual Web Developer, primero debe crear un proyecto tipo **Servicio Web ASP.NET**. Después, Visual Web Developer genera lo siguiente:

- archivos para contener el código del servicio Web (que implementa el servicio Web).
- un *archivo ASMX* (que proporciona acceso al servicio Web).
- un *archivo DISCO* (que utilizan los clientes potenciales para descubrir el servicio Web).

La figura 22.1 muestra los archivos que conforman un servicio Web. Al crear una aplicación **Servicio Web ASP.NET** en Visual Web Developer, el IDE por lo común genera varios archivos adicionales. Sólo mostramos los archivos específicos para las aplicaciones de servicios Web. En la sección 22.2.2 hablaremos sobre estos archivos.

Visual Web Developer genera archivos de código para la clase del servicio Web y cualquier otro código que forma parte de la implementación del servicio Web. En la clase del servicio Web se definen los métodos que estarán disponibles para las aplicaciones cliente. Al igual que las aplicaciones Web ASP.NET, los servicios Web ASP.NET pueden probarse con el servidor integrado de Visual Web Developer. No obstante, para que un servicio Web ASP.NET pueda accederse públicamente desde clientes que estén fuera de Visual Web Developer, debe desplegar el servicio Web en un servidor Web como el servidor Web Internet Information Services (IIS).

En un servicio Web, los métodos se invocan a través de una *Llamada a procedimiento remoto (RPC)*. Estos métodos, que se marcan con el atributo **WebMethod**, a menudo se conocen como *métodos del servicio Web*, o simplemente como *métodos Web*; de aquí en adelante nos referiremos a ellos como métodos Web. Al declarar un método con el atributo **WebMethod**, otras clases pueden acceder a él mediante RPCs, lo cual se conoce como *exponer* un método Web. En la sección 22.4 hablaremos sobre los detalles acerca de cómo exponer los métodos Web.

22.2.2 Descubrimiento de servicios Web

Una vez que se implementa un servicio Web, se compila y despliega en un servidor Web (lo cual veremos en la sección 22.4), una aplicación cliente puede *consumir* (es decir, usar) el servicio Web. No obstante, los clientes deben poder encontrar el servicio Web y aprender acerca de sus herramientas. *Descubrimiento de servicios Web (DISCO)* es una tecnología específica de Microsoft, que se utiliza para localizar los servicios Web en un servidor. Cuatro tipos de archivos DISCO facilitan el proceso de descubrimiento: archivos *.disco*, *.vsdisco*, *.discomap* y *.map*.

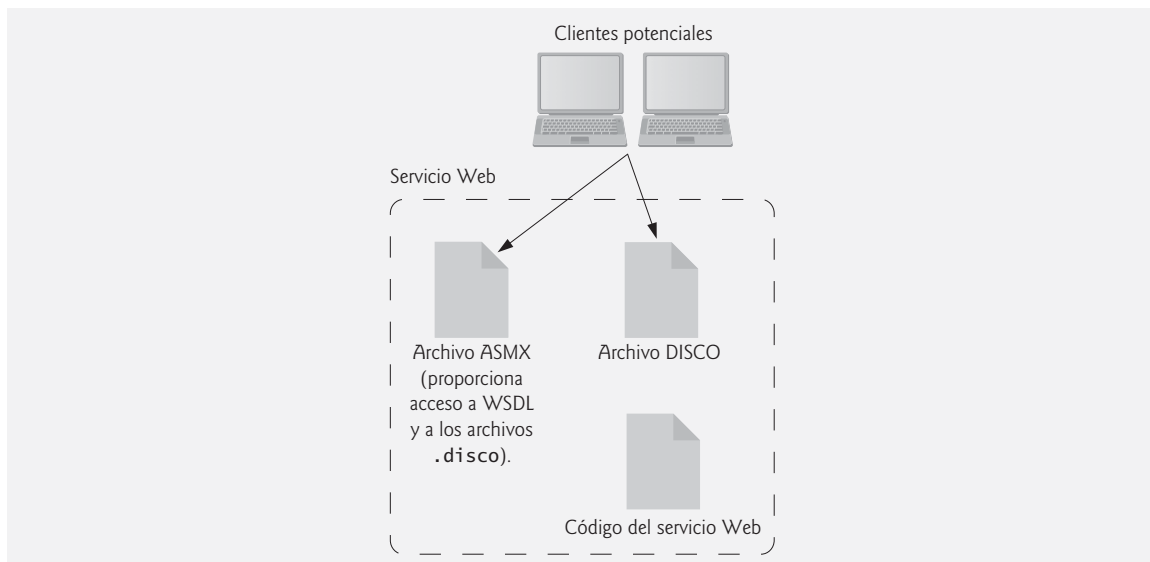


Figura 22.1 | Componentes de un servicio Web.

Los archivos DISCO consisten en marcado XML que describe para los clientes la ubicación de los servicios Web. Para acceder a un archivo `.disco` se utiliza una página ASMX del servicio Web, la cual contiene marcado que especifica referencias a los documentos que definen varios servicios Web. Los datos resultantes que se devuelven del acceso a un archivo `.disco` se colocan en el archivo `.discomap`.

Un archivo `.vsdisco` se coloca en el directorio de la aplicación de un servicio Web, y se comporta de una manera un poco distinta. Cuando un cliente potencial solicita un archivo `.vsdisco`, se genera en forma dinámica el marcado XML que describe las ubicaciones de los servicios Web, y después se devuelve al cliente. Primero, el .NET Framework busca los servicios Web en el directorio en el que se encuentra el archivo `.vsdisco`, así como en los subdirectorios de ese directorio. Después, el .NET Framework genera XML (usando la misma sintaxis que la de un archivo `.disco`) que contiene referencias a todos los servicios Web que encontró en esta búsqueda.

Debemos tener en cuenta que un archivo `.vsdisco` no almacena el marcado que se genera en respuesta a una solicitud, sino que el archivo `.vsdisco` en el disco contiene opciones de configuración que especifican su comportamiento. Por ejemplo, los desarrolladores pueden especificar en el archivo `.vsdisco` ciertos directorios en los que *no* debe buscarse cuando un cliente solicita un archivo `.vsdisco`. Aunque un desarrollador puede abrir un archivo `.vsdisco` en un editor de texto y examinar su contenido, esto muy raras veces es necesario; el propósito de un archivo `.vsdisco` es que lo soliciten (es decir, lo vean en un explorador) los clientes a través de la Web. Cada vez que esto ocurre, se genera y se visualiza nuevo marcado.

El uso de archivos `.vsdisco` beneficia a los desarrolladores de diversas formas. Estos archivos sólo contienen una pequeña cantidad de datos y proporcionan información actualizada acerca de los servicios Web disponibles en un servidor. No obstante, los archivos `.vsdisco` generan más sobrecarga (es decir, requieren más procesamiento) que los archivos `.disco`, ya que debe realizarse una búsqueda cada vez que se accede a un archivo `.vsdisco`. Por lo tanto, algunos desarrolladores consideran más benéfico actualizar los archivos `.disco` en forma manual. Muchos sistemas utilizan ambos tipos de archivos. Como veremos en breve, los servicios Web que se crean mediante ASP.NET contienen la funcionalidad para generar un archivo `.disco`, cuando éste se solicita. Este archivo `.disco` contiene referencias sólo a archivos en el servicio Web actual. Por ende, es común que un desarrollador coloque un archivo `.vsdisco` en el directorio raíz de un servidor; al acceder a este archivo, localiza los archivos `.disco` para los servicios Web en cualquier parte del sistema, y utiliza el marcado que encuentra en estos archivos `.disco` para devolver información acerca de todo el sistema completo.

22.2.3 Determinación de la funcionalidad de un servicio Web

Después de localizar un servicio Web, el cliente debe determinar la funcionalidad de éste y cómo utilizarlo. Por lo general, y para este fin, los servicios Web contienen una *descripción del servicio*. Éste es un documento XML que se conforma al *Lenguaje de descripción de servicios Web* (WS-DL): un vocabulario de XML que define los métodos que un servicio Web hace disponibles, y la manera en que los clientes interactúan con estos métodos. El documento WSDL también especifica información de bajo nivel que podrían necesitar los clientes, como los formatos requeridos para las solicitudes y respuestas.

El propósito de los documentos WSDL no es que los desarrolladores puedan leerlos, sino las aplicaciones, para que éstas sepan cómo interactuar con los servicios Web que se describen en los documentos. Visual Web Developer genera un archivo ASMX cuando se construye un servicio Web. Los archivos con la extensión de archivo `.asmx` son archivos de servicio Web ASP.NET, y se ejecutan en un servidor Web (por ejemplo, IIS). Al mostrarse en un explorador Web, un archivo ASMX presenta las descripciones de los métodos Web y vínculos a páginas de prueba que permiten a los usuarios ejecutar llamadas de ejemplo a estos métodos. En esta sección explicaremos estas páginas de prueba con mayor detalle. El archivo ASMX también especifica la clase de implementación del servicio Web, y opcionalmente el archivo de código de subyacente en el que se definen el servicio Web y los ensamblados a los que éste hace referencia. Cuando el servidor Web recibe una solicitud en relación con el servicio Web, accede al archivo ASMX que, a su vez, invoca a la implementación del servicio Web. Para ver información más técnica acerca del servicio Web, los desarrolladores pueden acceder al archivo WSDL (que se genera mediante ASP.NET). En breve le mostraremos cómo hacer esto.

La página ASMX en la figura 22.2 muestra información acerca del servicio Web EnteroEnorme que crearemos en la sección 22.4. Este servicio Web está diseñado para realizar cálculos con enteros que contengan un máximo de 100 dígitos. La mayoría de los lenguajes de programación no pueden realizar cálculos fácilmente al usar enteros tan grandes. El servicio Web proporciona a las aplicaciones cliente métodos que reciben dos “enteros enormes” y determinan su suma, su diferencia, cuál es mayor o menor y si los dos números son iguales. Observe que la parte

superior de la página contiene un vínculo a la **Descripción del servicio** Web. ASP.NET genera la descripción del servicio WSDL del código que usted escribe para definir el servicio Web. Los programas cliente utilizan la descripción de un servicio Web para validar las llamadas a los métodos Web cuando se compilan los programas cliente.

ASP.NET genera la información WSDL en forma dinámica, en vez de crear un archivo WSDL. Si un cliente solicita la descripción WSDL del servicio Web (ya sea adjuntando *?WSDL* al URL del archivo ASMX, o haciendo clic en el vínculo **Descripción del servicio**), ASP.NET genera la descripción WSDL y después la devuelve al cliente para mostrarla en el explorador Web. Al generar la descripción WSDL en forma dinámica, nos aseguramos que los clientes reciban la información más actual acerca del servicio Web. Es común que un documento XML (por ejemplo, una descripción WSDL) se cree en forma dinámica y no se guarde en disco.

Cuando un usuario hace clic en el vínculo **Descripción del servicio** en la parte superior de la página ASMX en la figura 22.2, el explorador muestra el documento WSDL generado que contiene la descripción para nuestro servicio Web EnteroEnorme (figura 22.3).

22.2.4 Prueba de los métodos de un servicio Web

Debajo del vínculo **Descripción del servicio**, la página ASMX que se muestra en la figura 22.2 lista los métodos que ofrece el servicio Web. Al hacer clic en el nombre de cualquier método se solicita una página de prueba que describe a ese método (figura 22.4). La página de prueba permite a los usuarios probar el método, escribiendo valores para los parámetros y haciendo clic en el botón **Invocar**. (En breve hablaremos sobre el proceso de probar un método Web.) Debajo del botón **Invocar**, la página muestra mensajes de solicitud y respuesta de ejemplo, en los que se utilizan SOAP y HTTP POST. Estos protocolos son dos opciones para enviar y recibir mensajes en los servicios Web. El protocolo que transmite mensajes de solicitud y respuesta también se conoce como el *formato de alambre o protocolo de alambre* del servicio Web, ya que define la manera en que se va a enviar la información “a través del alambre”. SOAP es el formato de alambre que se utiliza con más frecuencia, ya que los mensajes SOAP pueden enviarse mediante el uso de varios protocolos de transporte, mientras que HTTP POST debe utilizar HTTP. Cuando se prueba un servicio Web a través de una página ASMX (como en la figura 22.4), la página ASMX utiliza HTTP POST para probar los métodos del servicio Web. Más adelante en este capítulo, cuando utilicemos servicios Web en nuestros programas en C#, emplearemos SOAP: el protocolo predeterminado para los servicios Web .NET.

La figura 22.4 muestra la página de prueba para el método Web MasGrande de EnteroEnorme. De esta página, los usuarios pueden probar el método escribiendo valores en los campos **primero:** y **segundo:**, y después haciendo clic en **Invocar**. El método se ejecuta y aparece una nueva ventana del explorador Web, en la que se muestra un documento que contiene el resultado (figura 22.5).

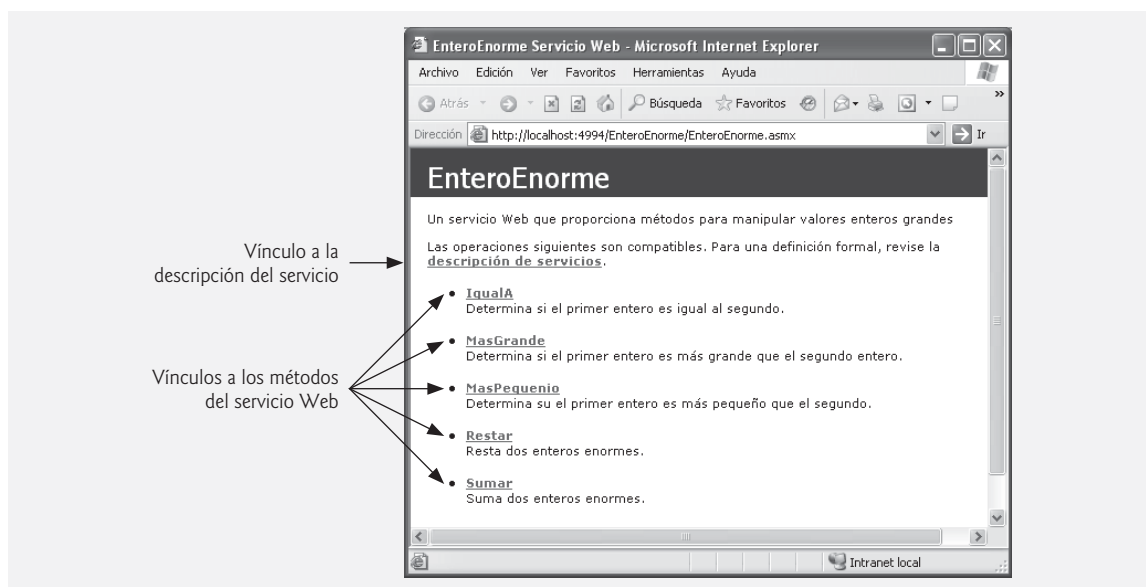


Figura 22.2 | Archivo ASMX representado en un explorador Web.

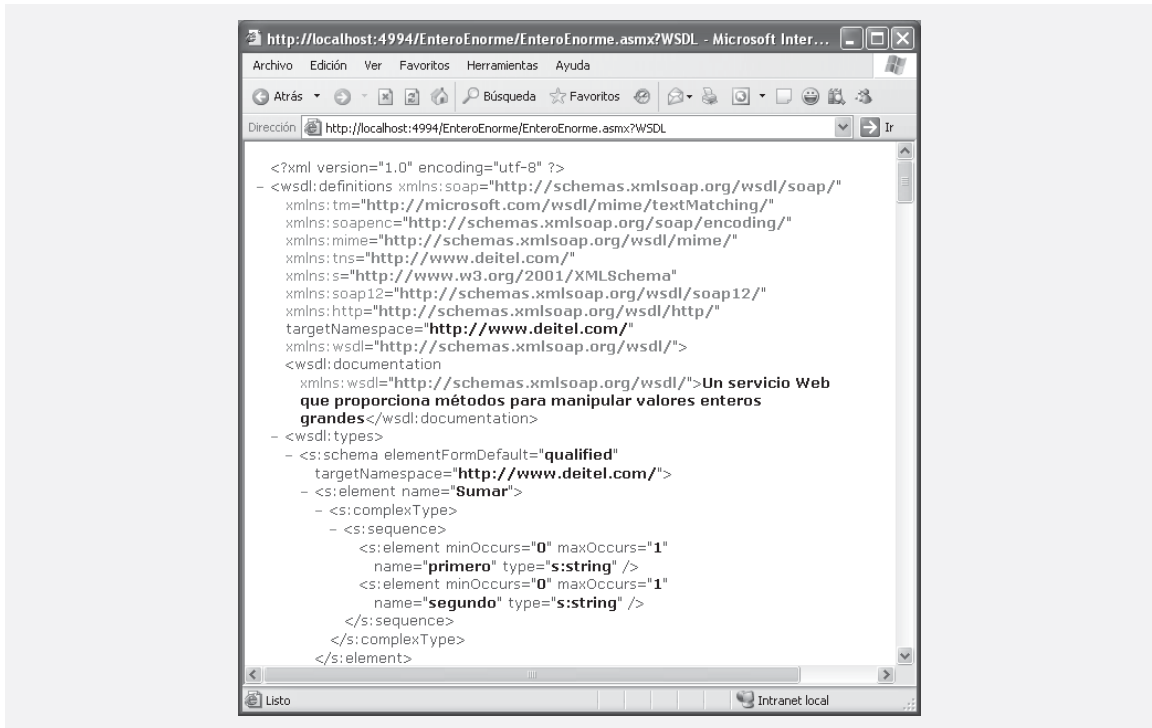


Figura 22.3 | Descripción de nuestro servicio Web EnteroEnorme.

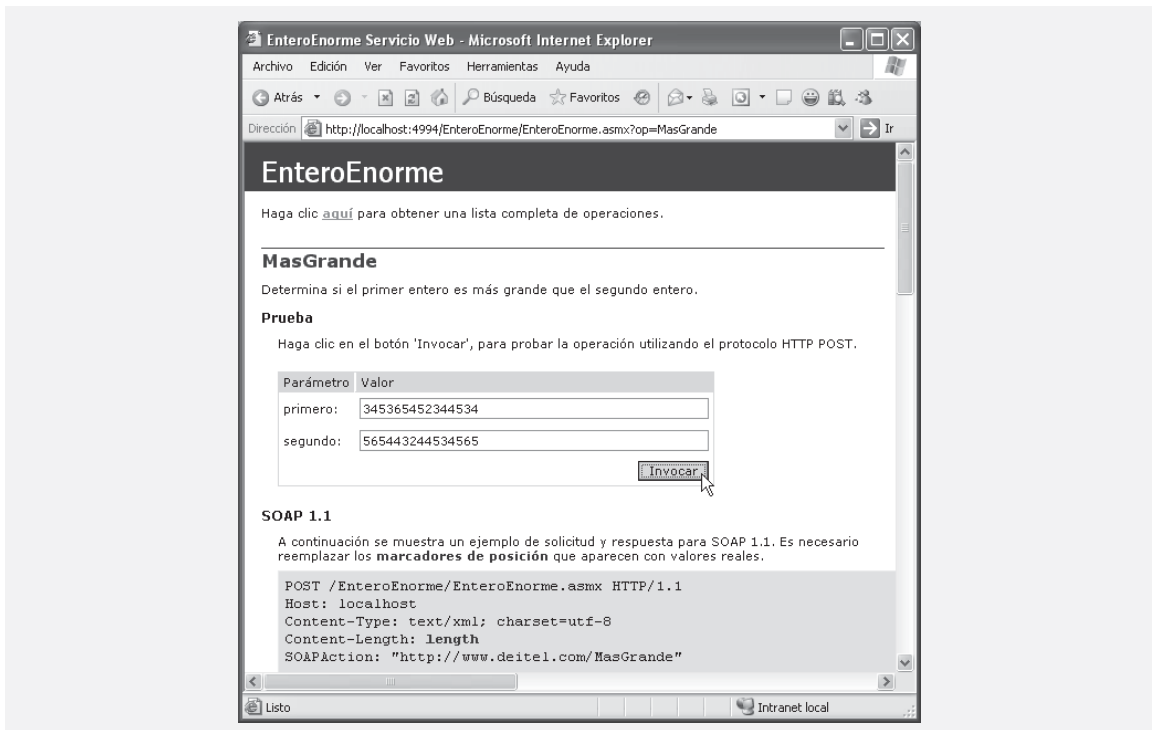


Figura 22.4 | Invocación de un método desde un explorador Web.

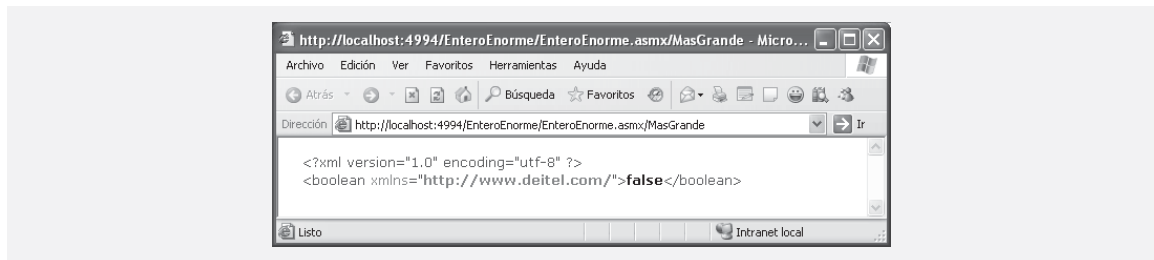


Figura 22.5 | Resultados de la invocación de un método Web desde un explorador Web.



Tip de prevención de errores 22.1

Usar la página ASMX de un servicio Web para probar y depurar los métodos puede ser útil para hacer que el servicio Web sea más confiable y robusto.

22.2.5 Creación de un cliente para utilizar un servicio Web

Ahora que hemos hablado acerca de los distintos archivos que conforman un servicio Web .NET, examinaremos las partes de un cliente del servicio Web .NET (figura 22.6). Un cliente .NET puede ser cualquier tipo de aplicación .NET, como una aplicación Windows, una aplicación de consola o una aplicación Web. Para habilitar una aplicación cliente, de manera que consuma un servicio Web, es necesario *agregar una referencia Web* al cliente. Este proceso agrega, a la aplicación cliente, archivos que permiten que el cliente acceda al servicio Web. Esta sección habla sobre Visual C# 2005, pero la discusión también se aplica a Visual Web Developer.

Para agregar una referencia Web en Visual C# 2005, haga clic con el botón derecho en el nombre del proyecto dentro del **Explorador de soluciones**, y seleccione la opción **Agregar referencia Web...** En el cuadro de diálogo que aparezca, especifique el servicio Web que se va a consumir. A continuación, Visual C# 2005 agrega una referencia Web apropiada a la aplicación cliente. En la sección 22.4 demostraremos en detalle cómo agregar referencias Web.

Cuando especificamos el servicio Web que deseamos consumir, Visual C# 2005 accede a la información WSDL del servicio Web y la copia en un archivo WSDL que se almacena en la carpeta Web References del proyecto del cliente. Este archivo está visible cuando la opción **Mostrar todos los archivos** está seleccionada en Visual C# 2005. [Nota: una copia del archivo WSDL proporciona a la aplicación cliente acceso local a la descripción del servicio Web. Para asegurar que el archivo WSDL esté actualizado, Visual C# 2005 proporciona una opción **Actualizar referencia Web** (disponible haciendo clic con el botón derecho en la referencia Web, dentro del

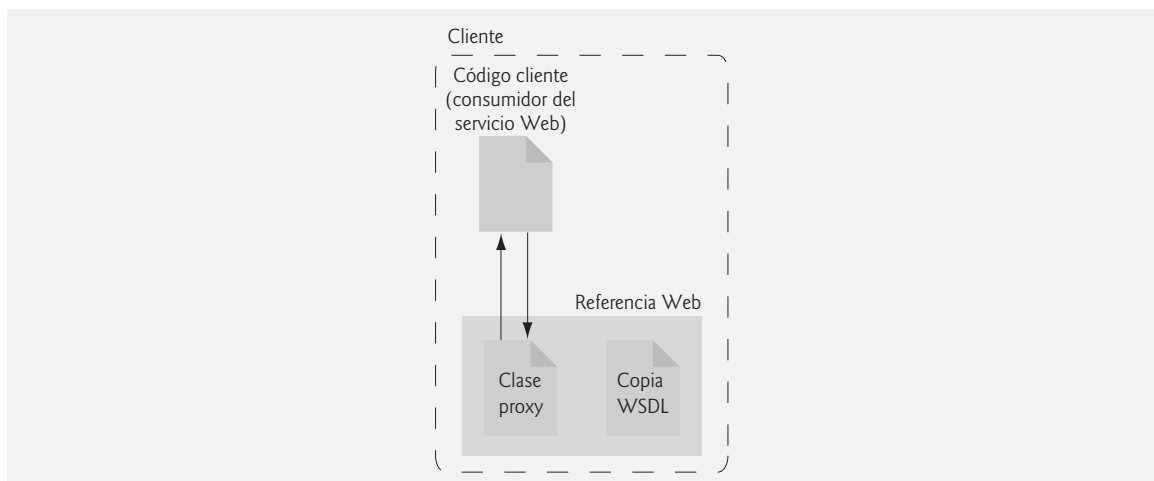


Figura 22.6 | Cliente de un servicio Web .NET, después de agregar una referencia Web.

Explorador de Soluciones), la cual actualiza los archivos en la carpeta Web References.] La información WSDL se utiliza para crear una *clase proxy*, la cual se encarga de todo el trabajo técnico requerido para las llamadas a los métodos Web (es decir, los detalles sobre el trabajo en red y la formación de los mensajes SOAP). Cada vez que la aplicación cliente llama a un método Web, en realidad llama a su correspondiente método en la clase proxy. Este método tiene el mismo nombre y parámetros que el método Web al que se está llamando, pero da formato a la llamada para que se envíe como una solicitud en un mensaje SOAP. El servicio Web recibe esta solicitud como un mensaje SOAP, ejecuta la llamada al método y devuelve el resultado como otro mensaje SOAP. Cuando la aplicación cliente recibe el mensaje SOAP que contiene la respuesta, la clase proxy lo deserializa y devuelve los resultados como el valor de retorno del método Web que se llamó. La figura 22.7 muestra las interacciones entre el código cliente, la clase proxy y el servicio Web.

El entorno .NET oculta la mayoría de estos detalles a usted como programador. Muchos aspectos de la creación y consumo de servicios Web (como la generación de archivos WSDL, clases proxy y archivos DISCO) los manejan Visual Web Developer, Visual C# 2005 y ASP.NET. Aunque los desarrolladores quedan liberados del tedioso proceso de crear estos archivos, de todas formas pueden modificar los archivos, en caso de ser necesario. Esto se requiere sólo cuando se desarrollan servicios Web avanzados; ninguno de nuestros ejemplos requiere modificaciones a estos archivos.

22.3 Protocolo simple de acceso a objetos (SOAP)

El Protocolo simple de acceso a objetos (SOAP) es independiente de la plataforma, utiliza XML para hacer llamadas a procedimientos remotos, por lo general a través de HTTP. Cada solicitud y respuesta se encapsula en un *mensaje SOAP*: un mensaje XML que contiene la información que requiere un servicio Web para procesar el mensaje. Los mensajes SOAP se escriben en XML, para ser legibles por los humanos e independientes de la plataforma. La mayoría de los *firewalls* (barreras de seguridad que restringen la comunicación entre redes) no restringen el tráfico HTTP. Por lo tanto, XML y HTTP permiten que computadoras en distintas plataformas envíen y reciban mensajes SOAP con pocas limitaciones.

Los servicios Web también utilizan SOAP para el extenso conjunto de tipos que soporta. El formato de alambre que se utiliza para transmitir solicitudes y respuestas debe soportar todos los tipos que se pasan entre las aplicaciones. Los tipos SOAP incluyen los tipos primitivos (por ejemplo, Integer), así como DateTime, XmlNode y otros. SOAP también puede transmitir arreglos de todos estos tipos. Además, los objetos DataSet pueden serializarse en SOAP. En la sección 22.7 podrá ver que puede transmitir tipos definidos por el usuario en mensajes SOAP.

Cuando un programa invoca a un método Web, la solicitud y toda la información relevante se encapsulan en un mensaje SOAP y se envían al servidor en el que reside el servicio Web. Cuando el servicio Web recibe este mensaje SOAP, empieza a procesar el contenido (que está dentro de una *envoltura SOAP*) que especifica el método que el cliente desea ejecutar, junto con cualquier argumento que el cliente pase a ese método. A este proceso de interpretar el contenido de un mensaje SOAP se le conoce como *analizar un mensaje SOAP*. Una vez que el servicio Web recibe y analiza una solicitud, se hace una llamada al método apropiado con los argumentos especificados (si los hay), y la respuesta se envía de vuelta al cliente en otro mensaje SOAP. El cliente analiza la respuesta para obtener el resultado de la llamada al método.

La solicitud SOAP en la figura 22.8 se obtuvo de la página de prueba para el método MasGrande del servicio Web EnteroEnorme (figura 22.4). Visual C# 2005 crea dicho mensaje cuando un cliente desea ejecutar el método MasGrande del servicio Web EnteroEnorme. Si el cliente es una aplicación Web, Visual Web Developer crea el mensaje SOAP. El mensaje en la figura 22.8 contiene receptáculos (length en la línea 4 y string en las líneas 16-17)

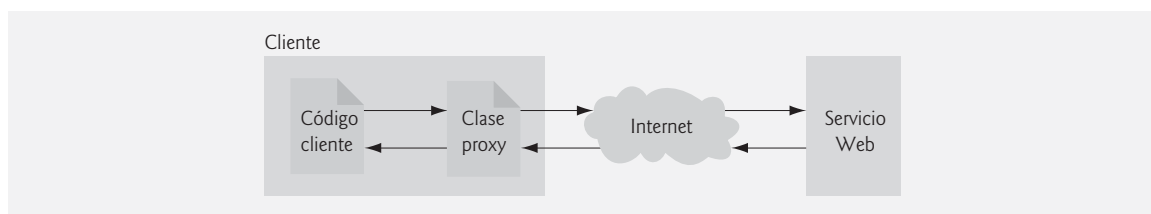


Figura 22.7 | Interacción entre el cliente de un servicio Web y el servicio Web.

```

1  POST /EnteroEnorme/EnteroEnorme.asmx HTTP/1.1
2  Host: localhost
3  Content-Type: text/xml; charset=utf-8
4  Content-Length: length
5  SOAPAction: "http://www.deitel.com/MasGrande"
6
7  <?xml version="1.0" encoding="utf-8"?>
8
9  <soap:Envelope
10     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
11     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
12     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
13
14     <soap:Body>
15         <MasGrande xmlns="http://www.deitel.com/">
16             <primero>string</primero>
17             <segundo>string</segundo>
18         </MasGrande>
19     </soap:Body>
20 </soap:Envelope>

```

Figura 22.8 | Mensaje de la solicitud SOAP para el servicio Web EnteroEnorme.

que representan los valores específicos para una llamada específica a MasGrande. Si ésta fuera una verdadera solicitud SOAP, los elementos *primero* y *segundo* (líneas 16-17) contendrían un valor real que el cliente pasa al servicio Web, en vez del receptáculo *string*. Por ejemplo, si esta envoltura transmitiera la solicitud de la figura 22.4, el elemento *primero* y el elemento *segundo* contendrían los números que se muestran en la figura, y el receptáculo *length* (línea 4) contendría la longitud del mensaje SOAP. La mayoría de los programadores no manipulan los mensajes SOAP directamente, sino que permiten al .NET Framework encargarse de los detalles de la transmisión.

22.4 Publicación y consumo de los servicios Web

En esta sección presentamos varios ejemplos de la creación (que también se conoce como *publicación*) y el uso (que también se conoce como *consumo*) de servicios Web. Recuerde que una aplicación que consume un servicio Web, en realidad consiste de dos partes: una clase proxy que representa al servicio Web y una aplicación cliente que accede al servicio Web a través de una instancia de la clase proxy. Esa instancia de la clase proxy pasa los argumentos de un método Web de la aplicación cliente al servicio Web. Cuando el método Web termina su tarea, la instancia de la clase proxy recibe el resultado y la analiza para la aplicación cliente. Visual C# 2005 y Visual Web Developer crean estas clases proxy por usted. En unos momentos demostraremos esto.

22.4.1 Definición del servicio Web EnteroEnorme

La figura 22.9 presenta el archivo de código subyacente para el servicio Web que crearemos en la sección 22.4.2. Cuando se crean servicios Web en Visual Web Developer, es necesario trabajar casi exclusivamente en el archivo

```

1  // Fig. 22.9: EnteroEnorme.cs
2  // El servicio Web EnteroEnorme realiza operaciones con enteros grandes.
3  using System;
4  using System.Web;
5  using System.Web.Services;
6  using System.Web.Services.Protocols;
7
8  [ WebService( Namespace = "http://www.deitel.com/",
9    Description = "Un servicio Web que proporciona métodos para" +
10      " manipular valores enteros grandes" ) ]

```

Figura 22.9 | El servicio Web EnteroEnorme. (Parte I de 4).

```

11 [ WebServiceBinding( ConformsTo = WsiProfiles.BasicProfile1_1 ) ]
12 public class EnteroEnorme : System.Web.Services.WebService
13 {
14     private const int MAXIMO = 100; // número máximo de dígitos
15     public int[] numero; // arreglo que representa el entero enorme
16
17     // constructor predeterminado
18     public EnteroEnorme()
19     {
20         numero = new int[ MAXIMO ];
21     } // fin del constructor predeterminado
22
23     // indexador que acepta un parámetro entero
24     public int this[ int indice ]
25     {
26         get
27         {
28             return numero[ indice ];
29         } // fin de get
30
31         set
32         {
33             numero[ indice ] = value;
34         } // fin de set
35     } // fin del indexador
36
37     // devuelve representación de cadena de EnteroEnorme
38     public override string ToString()
39     {
40         string returnString = "";
41
42         foreach ( int i in numero )
43             returnString = i + returnString;
44
45         return returnString;
46     } // fin del método ToString
47
48     // crea EnteroEnorme con base en el argumento
49     public static EnteroEnorme DeString( string valor )
50     {
51         // crea EnteroEnorme temporal, que el método va a devolver
52         EnteroEnorme enteroAnalizado = new EnteroEnorme();
53
54         for ( int i = 0; i < valor.Length; i++ )
55             enteroAnalizado[ i ] = Int32.Parse(
56                 valor[ valor.Length - i - 1 ].ToString() );
57
58         return enteroAnalizado;
59     } // fin del método DeString
60
61     // Método Web que suma los enteros representados por los argumentos String
62     [ WebMethod( Description = "Suma dos enteros enormes." ) ]
63     public string Sumar( string primero, string segundo )
64     {
65         int acarreo = 0;
66         EnteroEnorme operando1 = EnteroEnorme.DeString( primero );
67         EnteroEnorme operando2 = EnteroEnorme.DeString( segundo );
68         EnteroEnorme resultado = new EnteroEnorme(); // almacena el resultado de la suma
69     }

```

Figura 22.9 | El servicio Web EnteroEnorme. (Parte 2 de 4).

```

70      // realiza el algoritmo de suma para cada dígito
71      for ( int i = 0; i < MAXIMO; i++ )
72      {
73          // suma dos dígitos en la misma columna,
74          // el resultado es su suma más el acarreo de
75          // la operación anterior, módulo 10
76          resultado[ i ] =
77              ( operando1[ i ] + operando2[ i ] + acarreo ) % 10;
78
79          // asigna al acarreo el residuo de la división de las sumas de dos dígitos
            entre 10
80          acarreo = ( operando1[ i ] + operando2[ i ] + acarreo ) / 10;
81      } // fin de for
82
83      return resultado.ToString();
84  } // fin del método Sumar
85
86  // Método Web que resta enteros
87  // representados por los argumentos string
88  [ WebMethod( Description = "Resta dos enteros enormes." ) ]
89  public string Restar( string primero, string segundo )
90  {
91      EnteroEnorme operando1 = EnteroEnorme.DeString( primero );
92      EnteroEnorme operando2 = EnteroEnorme.DeString( segundo );
93      EnteroEnorme resultado = new EnteroEnorme();
94
95      // resta el dígito inferior del dígito superior
96      for ( int i = 0; i < MAXIMO; i++ )
97      {
98          // si el dígito superior es menor que el dígito inferior, necesitamos pedir
            prestado
99          if ( operando1[ i ] < operando2[ i ] )
100              PedirPrestado( operando1, i );
101
102          // resta inferior de superior
103          resultado[ i ] = operando1[ i ] - operando2[ i ];
104      } // fin de for
105
106      return resultado.ToString();
107  } // fin del método Restar
108
109  // pide prestado 1 al siguiente dígito
110  private void PedirPrestado( EnteroEnorme enteroEnorme, int place)
111  {
112      // si no hay lugar de dónde pedir prestado, indica el problema
113      if ( lugar >= MAXIMO - 1 )
114          throw new ArgumentException();
115
116      // en caso contrario, si el siguiente dígito es cero, pide prestado de la columna
            a la izquierda
117      else if ( enteroEnorme[ lugar + 1 ] == 0 )
118          PedirPrestado( enteroEnorme, lugar + 1 );
119
120      // suma diez al lugar actual, ya que pedimos prestado y restamos
121      // uno del dígito anterior; éste es el dígito al que pedimos prestado
122      enteroEnorme[ lugar ] += 10;
123      enteroEnorme[ lugar + 1 ]--;
124  } // fin del método PedirPrestado
125

```

Figura 22.9 | El servicio Web EnteroEnorme. (Parte 3 de 4).

```

126 // Método Web que devuelve verdadero si el primer entero es mayor que el segundo
127 [ WebMethod( Description = "Determina si el primer entero es " +
128   "más grande que el segundo entero." ) ]
129 public bool MasGrande( string primero, string segundo )
130 {
131     char[] ceros = { '0' };
132
133     try
134     {
135         // si la eliminación de todos los ceros del resultado de la resta
136         // es una cadena vacía, los números son iguales, por lo que se
137         // devuelve falso, en caso contrario se devuelve verdadero
138         if ( Restar( primero, segundo ).Trim( ceros ) == "" )
139             return false;
140         else
141             return true;
142     } // fin de try
143     // si ocurre una excepción ArgumentException, el primer
144     // número era más pequeño, por lo que se devuelve falso
145     catch ( ArgumentException exception )
146     {
147         return false;
148     } // fin de catch
149 } // fin del método MasGrande
150
151 // Método Web que devuelve verdadero si el primer entero es más pequeño que el segundo
152 [ WebMethod( Description = "Determina si el primer entero " +
153   "es más pequeño que el segundo." ) ]
154 public bool MasPequeno( string primero, string segundo )
155 {
156     // si segundo es más grande que primero, entonces primero es más pequeño que
157     // segundo
158     return MasGrande( segundo, primero );
159 } // fin del método MasPequeno
160
161 // Método Web que devuelve verdadero si dos enteros son iguales
162 [ WebMethod( Description = "Determina si el primer entero " +
163   "es igual al segundo." ) ]
164 public bool IgualA( string primero, string segundo )
165 {
166     // si el primero es más grande que el segundo, o el primero es
167     // más pequeño que el segundo, no son iguales
168     if ( MasGrande( primero, segundo ) || MasPequeno( primero, segundo ) )
169         return false;
170     else
171         return true;
172 } // fin del método IgualA
173 } // fin de la clase EnteroEnorme

```

Figura 22.9 | El servicio Web EnteroEnorme. (Parte 4 de 4).

de código subyacente. Como dijimos antes, este servicio Web está diseñado para realizar cálculos con enteros que tengan un máximo de 100 dígitos. Las variables `long` no pueden manejar enteros de este tamaño (es decir, se produce un desbordamiento). El servicio Web proporciona métodos que reciben dos “enteros enormes” (representados como objetos `string`) y determinan su suma, su diferencia, cuál es más grande o más pequeño, y si los dos números son iguales. Puede considerar estos métodos como *servicios* disponibles para los programadores de otras aplicaciones a través de la *Web* (de aquí que se utilice el término servicios Web). Cualquier programador puede acceder a este servicio Web, utilizar los métodos y por consecuencia evitar escribir 172 líneas de código.

Las líneas 8-10 contienen un atributo **WebService**. Si adjuntamos este atributo a la declaración de la clase de un servicio Web, podemos especificar el espacio de nombres y la descripción del servicio Web. Al igual que un espacio de nombres XML (vea la sección 19.4), las aplicaciones cliente utilizan el espacio de nombres de un servicio Web para diferenciar a ese servicio Web de los otros servicios disponibles en la Web. La línea 8 asigna `http://www.deitel.com` como el espacio de nombres del servicio Web, usando la propiedad **Namespace** del atributo **WebService**. Las líneas 9-10 utilizan la propiedad **Description** del atributo **WebService** para describir el propósito del servicio Web; éste aparece en la página ASMX (figura 22.2).

Visual Web Developer coloca la línea 11 en todos los servicios Web recién creados. Esta línea indica que el servicio Web se conforma al **Perfil básico 1.1 (BP 1.1)** desarrollado por la **Organización de interoperabilidad de servicios Web (WS-I)**, un grupo dedicado a promover la interoperabilidad entre los servicios Web desarrollados en distintas plataformas, con distintos lenguajes de programación. BP 1.1 es un documento que define las mejores prácticas para diversos aspectos de la creación y consumo de servicios Web (www.ws-i.org). Como vimos en la sección 22.2, el entorno .NET oculta muchos de estos detalles a los programadores. Al establecer la propiedad **ConformsTo** del atributo **WebServiceBinding** a **wsProfiles.BasicProfile1_1**, indicamos a Visual Web Developer que debe realizar su trabajo “detrás de las cámaras”, como la generación de archivos WSDL y ASMX, en conformidad con los lineamientos establecidos en BP 1.1. Para obtener más información acerca de la interoperabilidad de los servicios Web y el Perfil básico 1.1, visite el sitio Web de WS-I en www.ws-i.org.

De manera predeterminada, cada nuevo servicio Web que se crea en Visual Web Developer hereda de la clase **System.Web.Services.WebService** (línea 12). Aunque un servicio Web no necesita derivarse de la clase **WebService**, esta clase proporciona miembros que son útiles para determinar información acerca del cliente y del servicio Web en sí. Varios métodos en la clase **EnteroEnorme** están etiquetados con el atributo **WebMethod** (líneas 62, 88, 127, 152 y 161), el cual expone a un método de tal forma que pueda llamarse por vía remota. Cuando este atributo está ausente, el método no está accesible para los clientes que consumen el servicio Web. Al igual que el atributo **WebService**, este atributo contiene una propiedad **Description**, la cual permite que la página ASMX muestre información acerca del método (en la figura 22.2 se muestran estas descripciones).



Error común de programación 22.1

*Si no se expone un método como método Web, declarándolo con el atributo **WebMethod**, los clientes del servicio Web no podrán acceder a ese método.*



Tip de portabilidad 22.1

Especifique un espacio de nombres para cada servicio Web, de manera que los clientes puedan identificarlo en forma única. En general, debe utilizar el nombre de dominio de su compañía como el espacio de nombres del servicio Web, ya que se garantiza que los nombres de dominio de las compañías sean únicos.



Tip de portabilidad 22.2

Especifique las descripciones para un servicio Web y sus métodos Web, de tal forma que los clientes del servicio Web puedan ver información acerca del servicio, en la página ASMX del mismo.



Error común de programación 22.2

*Ningún método con el atributo **WebMethod** puede declararse **static**; para que un cliente pueda acceder a un método Web, debe existir una instancia de ese servicio Web.*

Las líneas 24-35 definen un indexador, el cual nos permite acceder a cualquier dígito en un **EnteroEnorme**. Las líneas 62-84 y 88-107 definen los métodos **Web Sumar** y **Web Restar**, que realizan las operaciones de suma y resta, respectivamente. El método **PedirPrestado** (líneas 110-124) maneja el caso en el cual el dígito que estamos examinando en el operando izquierdo es menor que el dígito correspondiente en el operando derecho. Por ejemplo, cuando restamos 19 de 32, por lo general examinamos los números en los operandos dígito por dígito, empezando desde la derecha. El número 2 es más pequeño que 9, por lo que sumamos 10 al 2 (cuyo resultado es 12). Después de pedir prestado, podemos restar 9 de 12, lo que produce un 3 para el dígito de más a la derecha en la solución. Después restamos 1 del 3 en 32; el siguiente dígito a la izquierda (es decir, el dígito al que le pedimos prestado). Esto deja un 2 en el lugar de las decenas. El dígito correspondiente en el otro operando es ahora el

1 del 19. Al restar 1 de 2 se produce 1, con lo cual el dígito correspondiente en el resultado es 1. El resultado final, cuando se juntan todos los dígitos, es 13. El método `PedirPrestado` es el método que suma 10 a los dígitos apropiados y resta 1 de los dígitos a la izquierda. Éste es un método utilitario que no está diseñado para que se llame en forma remota, por lo que no se califica con el atributo `WebMethod`.

En la figura 22.2 presentamos una captura de pantalla de la página ASMX `EnteroEnorme.asmx`, para la que el archivo `EnteroEnorme.cs` (figura 22.9) de código subyacente define los métodos Web. Una aplicación cliente puede invocar sólo a los cinco métodos que se listan en la captura de pantalla de la figura 22.2 (es decir, los métodos calificados con el atributo `WebMethod` en la figura 22.9).

22.4.2 Creación de un servicio Web en Visual Web Developer

Ahora le mostraremos cómo crear el servicio Web `EnteroEnorme`. En los siguientes pasos creará un proyecto **Servicio Web ASP.NET** que se ejecuta en el servidor Web IIS local de su computadora. Para crear el servicio Web `EnteroEnorme` en Visual Web Developer, realice los siguientes pasos:

Paso 1: crear el proyecto

Para empezar, debemos crear un proyecto de tipo **Servicio Web ASP.NET**. Seleccione **Archivo > Nuevo sitio Web...** para mostrar el cuadro de diálogo **Nuevo sitio Web** (figura 22.10). Seleccione **Servicio Web ASP.NET** en el panel **Plantillas**. Seleccione **HTTP** en la lista desplegable **Ubicación** para indicar que los archivos deben colocarse en un servidor Web. De manera predeterminada, Visual Web Developer indica que colocará los archivos en el servidor Web IIS del equipo local, en un directorio virtual llamado `WebSite` (`http://localhost/WebSite`). Sustituya el nombre `WebSite` con `EnteroEnorme` para este ejemplo. A continuación, seleccione **Visual C#** de la lista desplegable **Lenguaje** para indicar que utilizará Visual C# para generar este servicio Web. Visual Web Developer coloca el archivo de solución (`.sln`) del proyecto del servicio Web en la subcarpeta `Projects` dentro de la carpeta `Mis documentos\Visual Studio 2005` del usuario de Windows actual. Si no tiene acceso a un servidor Web IIS para generar y probar los ejemplos en este capítulo, puede elegir **Sistema de archivos** de la lista desplegable **Ubicación**. En este caso, Visual Web Developer colocará los archivos de su servicio Web en su disco duro local. Después podrá probar el servicio Web con el servidor Web integrado de Visual Web Developer.

Paso 2: examinar el proyecto recién creado

Cuando se crea el proyecto, se muestra de manera predeterminada el archivo de código subyacente `Service.cs`, que contiene el código para un servicio Web simple (figura 22.11). Si el archivo de código subyacente no está abierto, puede abrirlo haciendo doble clic sobre el archivo en el directorio `App_Code` que se lista en el **Explorador de soluciones**. Visual Web Developer incluye cuatro declaraciones `using` que son útiles para desarrollar servicios

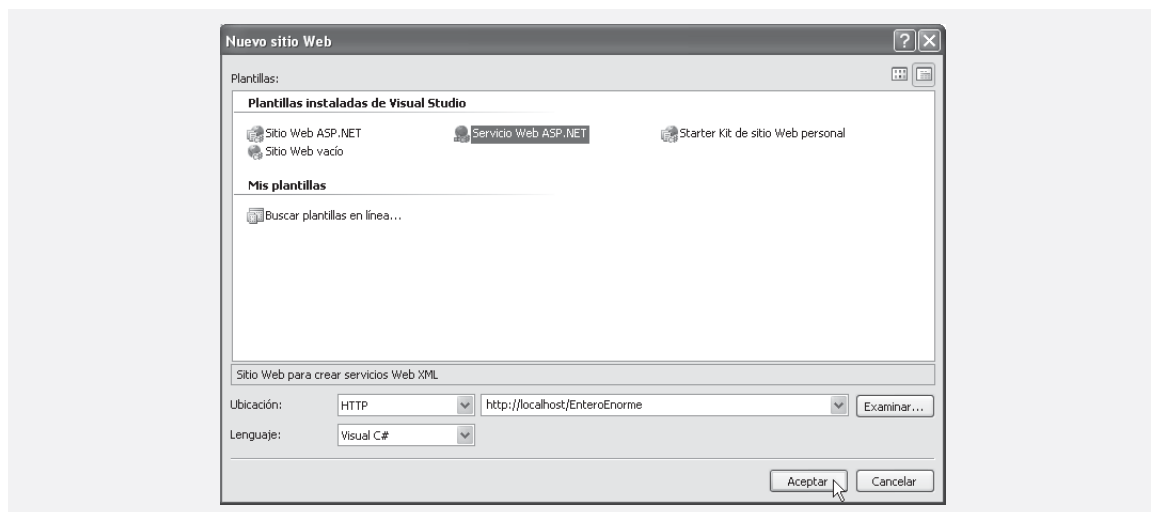


Figura 22.10 | Creación de un **Servicio Web ASP.NET** en Visual Web Developer.

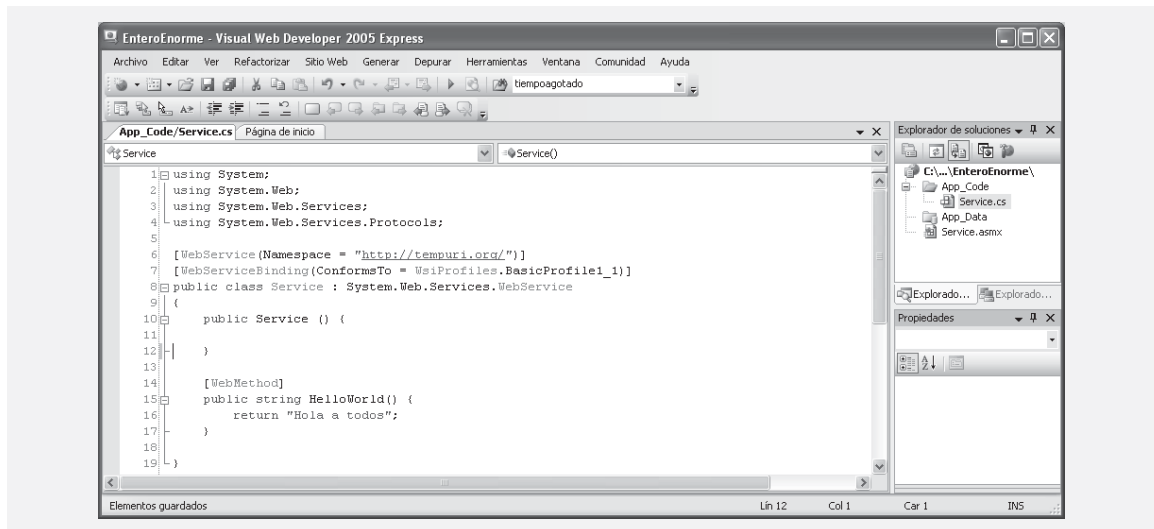


Figura 22.11 | Vista del código de un servicio Web.

Web (líneas 1-4). De manera predeterminada, un nuevo archivo de código subyacente define a una clase llamada *Service*, la cual está marcada con los atributos *WebService* y *WebServiceBinding* (líneas 6-7). La clase contiene un método Web de ejemplo llamado *HelloWorld* (líneas 14-17). Este método es un receptáculo que usted sustituirá con su(s) propio(s) método(s).

Paso 3: modificar y cambiar el nombre del archivo de código subyacente

Para crear el servicio Web *EnteroEnorme* que desarrollamos en esta sección, modifique el archivo *Service.cs*, sustituyendo todo el código de ejemplo que proporciona Visual Web Developer con todo el código del archivo de código subyacente *EnteroEnorme* (figura 22.9). Después, cambie el nombre al archivo por el de *EnteroEnorme.cs* (haga clic con el botón derecho del ratón en el **Explorador de soluciones**, y seleccione la opción **Cambiar nombre**).

Paso 4: examinar el archivo ASMX

El **Explorador de soluciones** lista un archivo (*Service.asmx*) además del archivo de código subyacente. En la figura 22.2 vimos cuando se accede a la página ASMX de un servicio Web a través de un explorador Web, esta página muestra información acerca de los métodos del servicio Web y proporciona acceso a la información WSDL del servicio Web. No obstante, si abre el archivo ASMX en el disco, verá que en realidad sólo contiene lo siguiente:

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/Service.cs"
    Class="Service" %>
```

para indicar el lenguaje de programación en el que está escrito el archivo de código subyacente del servicio Web, la ubicación del archivo de código subyacente y la clase que define al servicio Web. Cuando se solicita la página ASMX a través de IIS, ASP.NET utiliza esta información para generar el contenido que se muestra en el explorador Web (es decir, la lista de métodos Web y sus descripciones).

Paso 5: modificar el archivo ASMX

Cada vez que cambiamos el nombre del archivo de código subyacente o el nombre de la clase que define al servicio Web, debemos modificar el archivo ASMX de manera acorde. Por lo tanto, después de definir la clase *EnteroEnorme* en el archivo *EnteroEnorme.cs* de código subyacente, modifique el archivo ASMX para que contenga las siguientes líneas:

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/EnteroEnorme.cs"
    Class="EnteroEnorme" %>
```



Tip de prevención de errores 22.2

Actualice el archivo ASMX del servicio Web de manera apropiada, cada vez que cambie el nombre del archivo de código subyacente, o el nombre de la clase de un servicio Web. Visual Web Developer crea el archivo ASMX, pero no lo actualiza de manera automática cuando realizamos cambios en otros archivos del proyecto.

Paso 6: cambiar el nombre del archivo ASMX

El paso final para crear el servicio Web EnteroEnorme es cambiar el nombre del archivo ASMX a EnteroEnorme.asmx.

22.4.3 Despliegue del servicio Web EnteroEnorme

El servicio Web EnteroEnorme ya está desplegado, ya que lo creamos en el servidor IIS local de nuestra computadora. Puede elegir la opción **Generar sitio Web** del menú **Generar** para asegurarse que el servicio Web se compile sin errores. También puede probar el servicio Web directamente desde Visual Web Developer; seleccione **Iniciar sin depuración** del menú **Depurar**. Esta opción abre una ventana del explorador, la cual contiene la página ASMX que se muestra en la figura 22.2. Al hacer clic en el vínculo para un método específico del servicio Web EnteroEnorme, se muestra una página Web como la de la figura 22.4, que le permite probar el método. También puede acceder a la página ASMX del servicio Web desde su computadora, si escribe el siguiente URL en un explorador Web:

`http://localhost/EnteroEnorme/EnteroEnorme.asmx`

Acceso a la página ASMX del servicio Web EnteroEnorme desde otra computadora

En algún momento dado, requerirá que otros clientes puedan acceder a su servicio Web para utilizarlo. Si despliega el servicio Web en un servidor Web IIS, un cliente puede conectarse a ese servidor para acceder al servicio Web, con un URL de la forma

`http://host/EnteroEnorme/EnteroEnorme.asmx`

en donde *host* es el nombre de host, o dirección IP, del servidor Web. Para acceder al servicio Web desde otra computadora en la red de área local de su compañía o escuela, puede sustituir *host* con el nombre real de la computadora en la que se ejecuta IIS.

Si tiene el sistema operativo Windows XP Service Pack 2 en la computadora que ejecuta IIS, tal vez esa computadora no acepte solicitudes de otras computadoras de manera predeterminada. Si desea permitir que otras computadoras se conecten al servidor Web de su computadora, realice los siguientes pasos:

1. Seleccione **Inicio > Panel de control** para abrir la ventana **Panel de control** de su sistema; después haga doble clic en **Firewall de Windows** para que aparezca el cuadro de diálogo de opciones de **Firewall de Windows**.
2. En ese cuadro de diálogo, haga clic en la ficha **Opciones avanzadas**, seleccione **Conexión de área local** (o el nombre de la conexión de su red, si es distinto) en la lista **Configuración de conexión de red** y haga clic en el botón **Configuración** para mostrar el cuadro de diálogo **Opciones avanzadas**.
3. En este cuadro de diálogo, asegúrese que esté seleccionada la casilla de verificación **Servidor Web (HTTP)**, para permitir que los clientes en otras computadoras envíen solicitudes al servidor Web de su computadora.
4. Haga clic en el botón **Aceptar** del cuadro de diálogo **Opciones avanzadas**; después en el botón **Aceptar** del cuadro de diálogo de opciones de **Firewall de Windows**.

Acceder a la página ASMX del servicio Web EnteroEnorme cuando éste se ejecuta en el servidor Web integrado de Visual Web Developer

En el *paso 1* de la sección 22.4.2 vimos que, si no tiene acceso a un servidor IIS para desplegar y probar su servicio Web, puede crearlo en el disco duro de su computadora y utilizar el servidor Web integrado de Visual Web Developer para probarlo. En este caso, si selecciona **Iniciar sin depuración** en el menú **Depurar**, Visual Web Developer ejecuta su servidor Web integrado y después abre un explorador Web que contiene la página ASMX del servicio Web, para que usted pueda probarlo.

Por lo general, los servidores Web reciben las solicitudes en el puerto 80. Para asegurar que el servidor Web integrado de Visual Web Developer no entre en conflicto con otro servidor Web que se ejecute en su equipo local, el servidor Web de Visual Web Developer recibe las solicitudes en un número de puerto elegido al azar. Cuando un servidor Web recibe solicitudes en un número de puerto que no sea el puerto 80, debe especificarse ese número de puerto como parte de la solicitud. En este caso, el URL para acceder a la página ASMX del servicio Web EnteroEnorme sería de siguiente forma

```
http://host:númeroPuerto/EnteroEnorme/EnteroEnorme.asmx
```

en donde *host* es el nombre de host, o dirección IP, de la computadora en la que se ejecuta el servidor Web integrado de Visual Web Developer, y *númeroPuerto* es el puerto específico en el que el servidor Web recibe las solicitudes. Cuando pruebe el servicio Web desde Visual Web Developer podrá ver este número de puerto en el campo **Dirección** de su explorador Web. Por desgracia, los servicios Web que se ejecutan mediante el uso del servidor Web integrado de Visual Web Developer no pueden utilizarse a través de una red.

22.4.4 Creación de un cliente para consumir el servicio Web EnteroEnorme

Ahora que hemos definido y desplegado nuestro servicio Web, demostraremos cómo consumirlo desde una aplicación cliente. En esta sección creará una aplicación Windows que actuará como cliente, mediante el uso de Visual C# 2005. Después de crear la aplicación cliente, agregará una clase proxy al proyecto para permitir que el cliente acceda al servicio Web. Recuerde que la clase proxy (o proxy) se genera a partir del archivo WSDL del servicio Web, y permite al cliente llamar a los métodos Web a través de Internet. La clase proxy se encarga de todos los detalles relacionados con el proceso de comunicarse con el servicio Web. Esta clase proxy está oculta a los programadores de manera predeterminada; puede verla en el **Explorador de soluciones**, haciendo clic en el botón **Mostrar todos los archivos**. El propósito de la clase proxy es hacer que los clientes creen que están llamando a los métodos Web directamente.

Este ejemplo demuestra cómo crear un cliente de un servicio Web y generar una clase proxy, que permita al cliente acceder al servicio Web EnteroEnorme. Para empezar, creará un proyecto y le agregará una referencia Web. Cuando agregue la referencia Web, Visual C# 2005 generará la clase proxy apropiada. Después creará una instancia de la clase proxy y la utilizará para llamar a los métodos del servicio Web. Primero, cree una aplicación Windows en Visual C# 2005, y después realice los siguientes pasos:

Paso 1: abrir el cuadro de diálogo Agregar referencia Web

Haga clic con el botón derecho del ratón en el nombre del proyecto dentro del **Explorador de soluciones**, y seleccione la opción **Agregar referencia Web...** (figura 22.12).

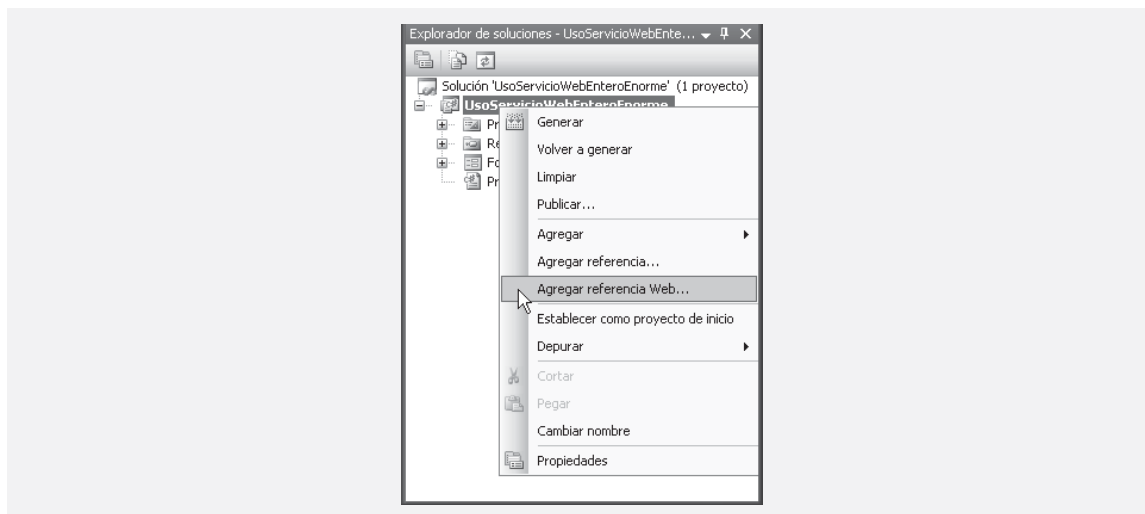


Figura 22.12 | Agregar la referencia de un servicio Web a un proyecto.

Paso 2: localizar los servicios Web en su computadora

En el cuadro de diálogo **Agregar referencia Web** que aparezca (figura 22.13), haga clic en la opción **Servicios Web del equipo local** para localizar las referencias Web almacenadas en el servidor Web IIS en su computadora local (<http://localhost>). Los archivos de este servidor se encuentran en `C:\Inetpub\wwwroot` de manera predefinida. Observe que el cuadro de diálogo **Agregar referencia Web** le permite buscar servicios Web en varias ubicaciones distintas. Muchas compañías que proporcionan servicios Web sólo distribuyen los URLs exactos con los que se puede acceder a sus servicios Web. Por esta razón, el cuadro de diálogo **Agregar referencia Web** también le permite escribir el URL específico de un servicio Web en el campo **Dirección URL**.

Paso 3: elegir el servicio Web a referenciar

Seleccione el servicio Web **EnteroEnorme** de la lista de servicios Web disponibles (figura 22.14).

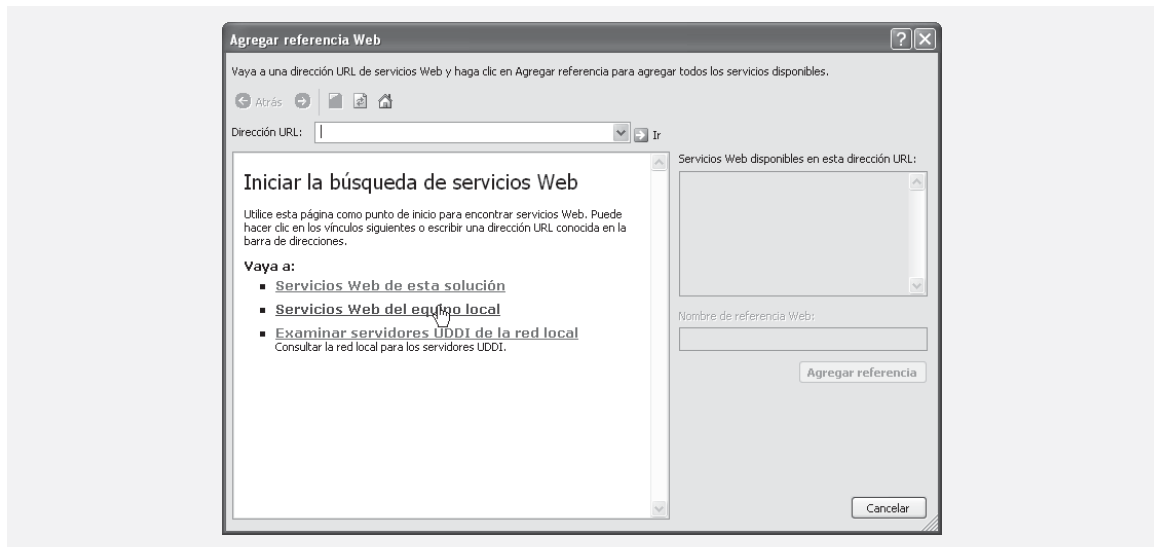


Figura 22.13 | El cuadro de diálogo **Agregar referencia Web**.

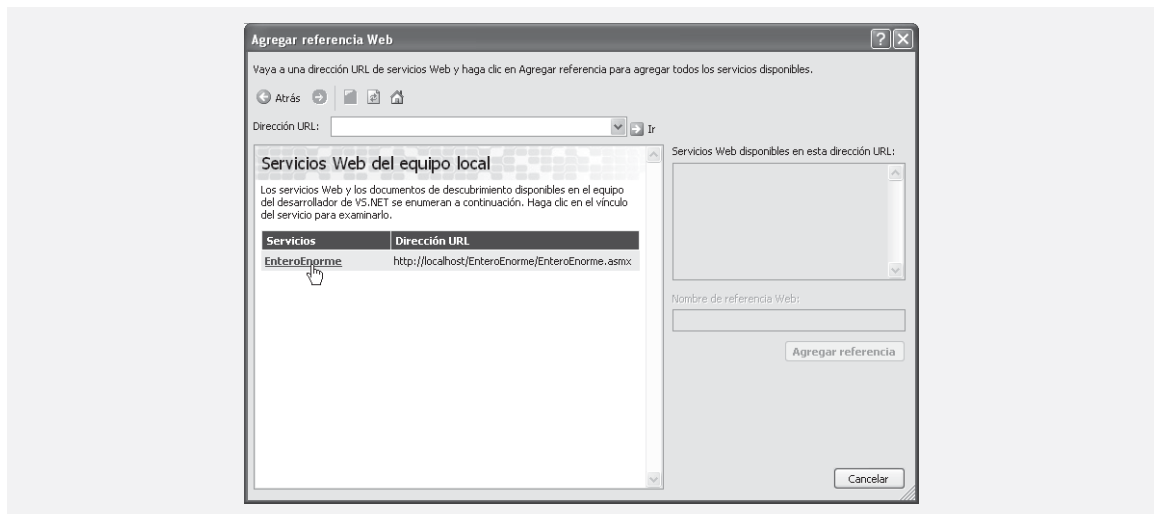


Figura 22.14 | Los servicios Web ubicados en `localhost`.

Paso 4: agregar la referencia Web

Agregue la referencia Web, haciendo clic en el botón **Agregar referencia** (figura 22.15).

Paso 5: ver la referencia Web en el Explorador de soluciones

Ahora el **Explorador de soluciones** (figura 22.16) debe contener una carpeta **Web References** con un nodo cuyo nombre debe ser igual al nombre del dominio en el que se encuentra el servicio Web. En este caso, el nombre es `localhost`, ya que estamos usando el servidor Web local. Cuando hagamos referencia a la clase `EnteroEnorme` en la aplicación cliente, lo haremos a través del espacio de nombres `localhost`.

Observaciones acerca de la creación de un cliente para consumir un servicio Web

Los pasos que acabamos de presentar también se aplican al proceso de agregar referencias Web a las aplicaciones Web creadas en Visual Web Developer. En la sección 22.6 presentaremos una aplicación Web que consume un servicio Web.

Al crear un cliente para consumir un servicio Web, agregue primero la referencia Web, de manera que Visual C# 2005 (o Visual Web Developer) pueda reconocer a un objeto de la clase proxy del servicio Web. Una vez que agregue la referencia Web al cliente, éste podrá acceder al servicio Web a través de un objeto de la clase proxy. Esta clase (llamada `EnteroEnorme`) está ubicada en el espacio de nombres `localhost`, por lo que debe usar `localhost.EnteroEnorme` para referenciar esta clase. Aunque debe crear un objeto de la clase proxy para acceder al

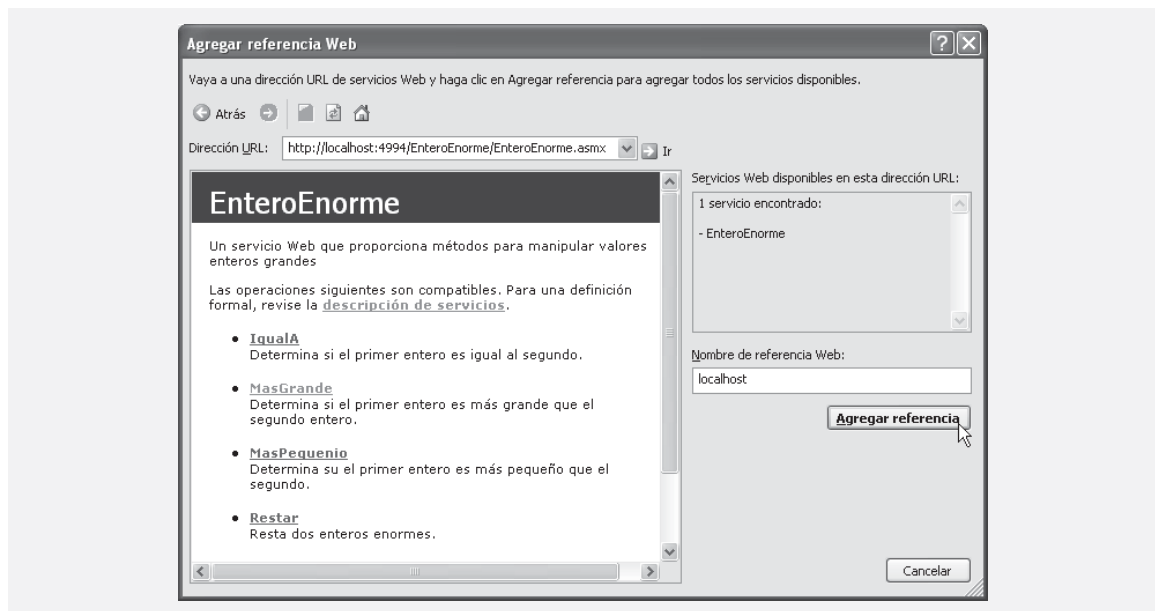


Figura 22.15 | Selección y descripción de una referencia Web.

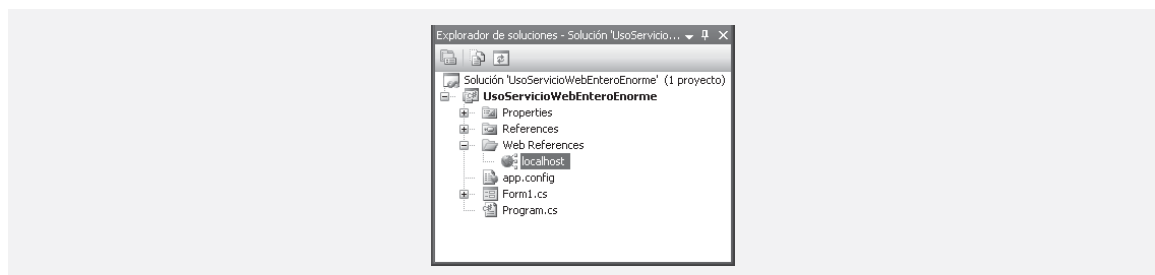


Figura 22.16 | Explorador de soluciones, después de agregar una referencia Web a un proyecto.

servicio Web, no necesita acceder al código de la clase proxy. Como veremos en la sección 22.4.5, puede invocar los métodos del objeto proxy como si fuera un objeto de la clase del servicio Web.

Los pasos que describimos en esta sección funcionan bien si conoce la referencia apropiada al servicio Web. No obstante, si está tratando de localizar un nuevo servicio Web, dos tecnologías comunes facilitan este proceso: *Descripción universal, descubrimiento e integración (UDDI)* y Descubrimiento de servicios Web (DISCO). En la sección 22.2 hablamos sobre DISCO. UDDI es un proyecto continuo para desarrollar un conjunto de especificaciones que definan la manera en que deben publicarse los servicios Web, para que los programadores en busca de servicios Web puedan encontrarlos. Microsoft y sus socios están trabajando en este proyecto para ayudar a los programadores a localizar servicios Web que se conformen a ciertas especificaciones, con lo cual los desarrolladores podrán buscar servicios Web a través de motores de búsqueda similares a Yahoo!® y Google™. En los sitios www.uddi.org y uddi.microsoft.com podrá aprender más acerca de UDDI y ver una demostración. Estos sitios contienen herramientas de búsqueda que hacen que sea conveniente buscar servicios Web.

22.4.5 Consumo del servicio Web EnteroEnorme

El formulario Windows Forms en la figura 22.17 utiliza el servicio Web EnteroEnorme para realizar cálculos con enteros positivos de hasta 100 dígitos de longitud. La línea 22 declara la variable `enteroRemoto` de tipo `localhost.EnteroEnorme`. Esta variable se utiliza en cada uno de los manejadores de eventos de la aplicación, para llamar a los métodos del servicio Web EnteroEnorme. El objeto proxy se crea y se asigna a esta variable en la línea 31 del manejador de eventos `Load` del formulario. Las líneas 52-53, 66-67, 95-96, 116-117 y 135-136 en los diversos manejadores de eventos de los botones invocan a los métodos del servicio Web. Observe que cada llamada se realiza en el objeto proxy local, que a su vez se comunica con el servicio Web a beneficio del cliente. Si descargó el ejemplo de www.deitel.com/books/csharpforprogrammers2, tal vez necesite regenerar el proxy; para ello, elimine la referencia Web y después agréguela de nuevo. Posteriormente haga clic con el botón derecho del ratón en `localhost` dentro de la carpeta **Web References** en el **Explorador de soluciones**, y seleccione la opción **Eliminar**. Después siga las instrucciones en la sección anterior para agregar la referencia Web al proyecto.

El usuario introduce dos enteros, cada uno de hasta 100 dígitos de largo. Al hacer clic en un botón, la aplicación invoca a un método Web para realizar la tarea apropiada y devolver el resultado. Observe que la aplicación cliente `UsoServicioEnteroEnorme` no puede realizar operaciones usando números de 100 dígitos de manera directa. En vez de eso, la aplicación crea representaciones `string` de estos números y los pasa como argumentos para los métodos Web que manejan dichas tareas para el cliente. Después utiliza el valor de retorno de cada operación para mostrar un mensaje apropiado.

```

1  // Fig. 22.17: UsoServicioWebEnteroEnorme.cs
2  // Uso del servicio Web EnteroEnorme.
3  using System;
4  using System.Collections.Generic;
5  using System.ComponentModel;
6  using System.Data;
7  using System.Drawing;
8  using System.Text;
9  using System.Windows.Forms;
10 using System.Web.Services.Protocols;
11
12 namespace UsoServicioWebEnteroEnorme
13 {
14     public partial class UsoServicioWebEnteroEnormeForm : Form
15     {
16         public UsoServicioWebEnteroEnormeForm()
17         {
18             InitializeComponent();
19         } // fin del constructor
20
21         // declara una referencia al servicio Web

```

Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte I de 5).

```

22     private localhost.EnteroEnorme enteroRemoto;
23
24     private char[] ceros = { '0' }; // carácter a eliminar de las cadenas
25
26     // instancia un objeto para interactuar con el servicio Web
27     private void UsoServicioEnteroEnormeForm_Load( object sender,
28         EventArgs e )
29     {
30         // crea instancia de enteroRemoto
31         enteroRemoto = new localhost.EnteroEnorme();
32     } // fin del método UsoServicioEnteroEnormeForm_Load
33
34     // suma dos enteros introducidos por el usuario
35     private void sumarButton_Click( object sender, EventArgs e )
36     {
37         // se asegura que los números no excedan de 100 dígitos y que ambos
38         // no sean de 100 dígitos de largo, lo cual produciría un desbordamiento
39         if ( primeroTextBox.Text.Length > 100 ||
40             segundoTextBox.Text.Length > 100 ||
41             ( primeroTextBox.Text.Length == 100 &&
42               segundoTextBox.Text.Length == 100 ) )
43         {
44             MessageBox.Show( "Los Enteros Enormes no deben tener " +
45                             "más de 100 dígitos\r\nAmbos enteros no pueden tener " +
46                             "longitud de 100: esto produce un desbordamiento", "Error",
47                             MessageBoxButtons.OK, MessageBoxIcon.Information );
48             return;
49         } // fin de if
50
51         // realiza la suma
52         resultadoLabel.Text = enteroRemoto.Sumar(
53             primeroTextBox.Text, segundoTextBox.Text ).TrimStart( ceros );
54     } // fin del método sumarButton_Click
55
56     // resta dos números introducidos por el usuario
57     private void restarButton_Click( object sender, EventArgs e )
58     {
59         // se asegura que los Enteros Enormes no excedan de 100 dígitos
60         if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
61             return;
62
63         // realiza la resta
64         try
65         {
66             string resultado = enteroRemoto.Restar(
67                 primeroTextBox.Text, segundoTextBox.Text ).TrimStart( ceros );
68
69             if ( resultado == "" )
70                 resultadoLabel.Text = "0";
71             else
72                 resultadoLabel.Text = resultado;
73         } // fin de try
74
75         // si el método Web lanza una excepción,
76         // entonces el primer argumento era menor que el segundo
77         catch ( SoapException exception )
78         {
79             //
80             MessageBox.Show(

```

Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte 2 de 5).


```

81         "El primer argumento era menor que el segundo" );
82     } // fin de catch
83 } // fin del método restarButton_Click
84
85 // determina si el primer número introducido
86 // por el usuario es mayor que el segundo
87 private void masGrandeButton_Click( object sender, EventArgs e )
88 {
89     // se asegura que los Enteros Enormes no excedan de 100 dígitos
90     if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
91         return;
92
93     // llama al método del servicio Web para determinar si
94     // el primer entero es más grande que el segundo
95     if ( enteroRemoto.MasGrande( primeroTextBox.Text,
96         segundoTextBox.Text ) )
97         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
98             " es mayor que " +
99             segundoTextBox.Text.TrimStart( ceros );
100     else
101         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
102             " no es mayor que " +
103             segundoTextBox.Text.TrimStart( ceros );
104 } // fin del método masGrandeButton_Click
105
106 // determina si el primer número introducido
107 // por el usuario es más pequeño que el segundo
108 private void masPequenioButton_Click( object sender, EventArgs e )
109 {
110     // se asegura que los Enteros Enormes no excedan de 100 dígitos
111     if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
112         return;
113
114     // llama al método del servicio Web para determinar si
115     // el primer entero es más pequeño que el segundo
116     if ( enteroRemoto.MasPequenio( primeroTextBox.Text,
117         segundoTextBox.Text ) )
118         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
119             " es menor que " +
120             segundoTextBox.Text.TrimStart( ceros );
121     else
122         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
123             " no es menor que " +
124             segundoTextBox.Text.TrimStart( ceros );
125 } // fin del método masPequenioButton_Click
126
127 // determina si dos números introducidos por el usuario son iguales
128 private void igualButton_Click( object sender, EventArgs e )
129 {
130     // se asegura que los Enteros Enormes no excedan de 100 dígitos
131     if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
132         return;
133
134     // llama al método del servicio Web para determinar si los enteros son iguales
135     if ( enteroRemoto.IgualA( primeroTextBox.Text,
136         segundoTextBox.Text ) )
137         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
138             " es igual a " + segundoTextBox.Text.TrimStart( ceros );
139     else

```

Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte 3 de 5).


```

140         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
141             " no es igual a " +
142             segundoTextBox.Text.TrimStart( ceros );
143     } // fin del método igualButton_Click
144
145     // determina si los números introducidos por el usuario son demasiado grandes
146     private bool ComprobarTamano( TextBox primero, TextBox segundo )
147     {
148         // muestra un mensaje de error si cualquier número tiene demasiados dígitos
149         if ( ( primero.Text.Length > 100 ) ||
150             ( segundo.Text.Length > 100 ) )
151         {
152             MessageBox.Show( "Los Enteros Enormes deben ser menores de 100 dígitos",
153                 "Error", MessageBoxButtons.OK, MessageBoxIcon.Information );
154             return true;
155         } // fin de if
156
157         return false;
158     } // fin del método ComprobarTamano
159 } // fin de la clase UsingHugeIntegerServiceForm
160 } // fin del espacio de nombres UsoServicioWebEnteroEnorme

```

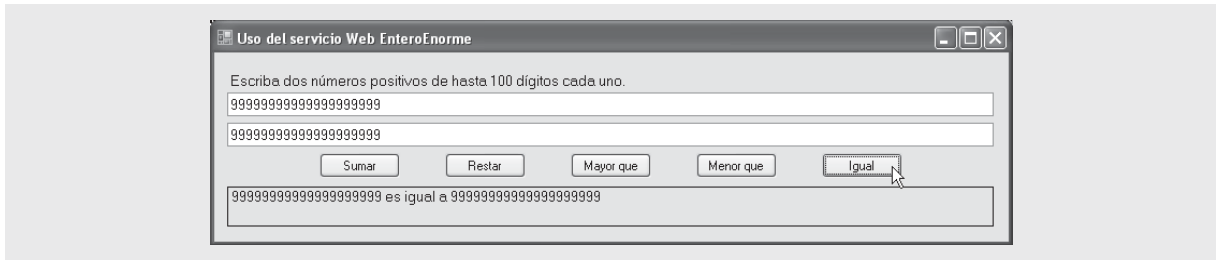


Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte 5 de 5).

Observe que la aplicación elimina los ceros a la izquierda en los números antes de mostrarlos, mediante una llamada al método `string TrimStart`. Al igual que el método `string Trim` (que vimos en el capítulo 16), `TrimStart` elimina todas las ocurrencias de los caracteres especificados por un arreglo `char` (línea 24) desde el principio de un objeto `string`.

22.5 Rastreo de sesiones en los servicios Web

En el capítulo 21, describimos las ventajas de mantener la información sobre los usuarios para personalizar sus experiencias. En especial, hablamos sobre el rastreo de sesiones mediante el uso de cookies y objetos `HttpSessionState`. Ahora incorporaremos el rastreo de sesiones a un servicio Web. Suponga que una aplicación cliente necesita llamar a varios métodos del mismo servicio Web, posiblemente varias veces a cada uno. En tal caso, sería benéfico para el servicio Web mantener la información de estado para el cliente. El rastreo de sesiones elimina la necesidad de pasar la información del cliente varias veces entre éste y el servicio Web. Por ejemplo, un servicio Web que proporciona acceso a las reseñas de restaurantes locales se beneficiaría de almacenar la dirección del usuario cliente. Una vez que se almacena la dirección del cliente en una variable de sesión, los métodos Web pueden devolver resultados personalizados y localizados, sin requerir que se pase la dirección en cada llamada a los métodos. Esto no sólo aumenta el rendimiento, sino que también requiere un menor esfuerzo por parte del programador; se pasa menos información en cada llamada a los métodos.

22.5.1 Creación de un servicio Web de Blackjack

Almacenar la información de las sesiones puede hacer que el servicio Web sea más intuitivo para los programadores de la aplicación cliente. Nuestro siguiente ejemplo es un servicio Web que asiste a los programadores en el desarrollo de un juego de cartas de blackjack (figura 22.18). El servicio Web proporciona métodos Web para repartir una carta y evaluar una mano de cartas. Después de presentar este servicio Web, lo utilizamos para que sirva como el repartidor para un juego de blackjack (figura 22.19). El servicio Web de blackjack utiliza una variable de sesión para mantener un mazo único de cartas para cada aplicación cliente. Varios clientes pueden utilizar el servicio al mismo tiempo, pero las llamadas a los métodos Web que realice un cliente específico sólo utilizarán el mazo almacenado en la sesión de ese cliente. Nuestro ejemplo utiliza un subconjunto simple de las reglas de blackjack de los casinos:

Se reparten dos cartas al repartidor y dos cartas al jugador. Las cartas del jugador se reparten con la cara hacia arriba. Sólo la primera de las cartas del repartidor se reparte con la cara hacia arriba. Cada carta tiene un valor. Una carta numerada del 2 al 10 vale lo que indique en su cara. Los jotos, reinas y reyes valen 10 cada uno. Los ases pueden valer 1 u 11, lo que sea más conveniente para el jugador (como pronto veremos). Si la suma de las dos cartas iniciales del jugador es 21 (por ejemplo, que se hayan repartido al jugador una carta con valor de 10 y un as, que cuenta como 11 en esta situación), el jugador tiene “blackjack” y gana el juego de inmediato. En caso contrario, el jugador puede empezar a tomar cartas adicionales, una a la vez. Estas cartas se reparten con la cara hacia arriba, y el jugador decide cuándo dejar de tomar cartas. Si el jugador “se pasa” (es decir, si la suma de las cartas del jugador excede a 21), el juego termina y el jugador pierde. Cuando el jugador está satisfecho con su conjunto actual de cartas, “se planta” (es decir, deja de tomar cartas) y se revela la carta oculta del repartidor. Si el total del repartidor es 16 o menos, debe tomar otra carta; en caso contrario, el repartidor debe plantarse. El repartidor debe seguir tomando cartas hasta que la suma de todas sus cartas sea mayor o igual a 17. Si el repartidor se pasa de 21, el jugador gana. En caso contrario, la mano con el total de puntos que sea mayor gana. Si el repartidor y el jugador tienen el mismo total de puntos, el juego es un “empate” y nadie gana.

El servicio Web (figura 22.18) proporciona métodos para repartir una carta y determinar el valor en puntos de una mano. Representamos cada carta como un objeto `string` que consiste en un dígito (por ejemplo, del 1 al 13), representando la cara de la carta (por ejemplo del as hasta el rey), seguido por un espacio y un dígito (por ejemplo 0-3) el cual representa el palo de la carta (por ejemplo, tréboles, diamantes, corazones o espadas). Por ejemplo, el joto de corazones se representa como "11 2", y el dos de tréboles se representa como "2 0". Después de desplegar el servicio Web, creamos una aplicación para Windows que utiliza los métodos del servicio Web `ServicioBlackjack` para implementar un juego de blackjack. Para crear y desplegar este servicio Web, siga los pasos que presentamos en las secciones 22.4.2-22.4.3 para el servicio `EnteroEnorme`.

Las líneas 15-16 definen el método `RepartirCarta` como un método Web. Al establecer la propiedad `EnableSession` a `True`, se indica que debe mantenerse la información de sesión y que debe estar accesible para

```

1  // Fig. 22.18: ServicioBlackjack.cs
2  // Servicio Web de Blackjack, para repartir y contar cartas.
3  using System;
4  using System.Web;
5  using System.Web.Services;
6  using System.Web.Services.Protocols;
7  using System.Collections;
8
9  [ WebService( Namespace = "http://www.deitel.com/", Description =
10     "Un servicio Web que reparte y cuenta cartas para el juego de Blackjack" ) ]
11  [ WebServiceBinding( ConformsTo = WsiProfiles.BasicProfile1_1 ) ]
12  public class ServicioBlackjack : System.Web.Services.WebService
13  {
14      // reparte una carta que no se ha repartido todavía
15      [ WebMethod( EnableSession = true,
16         Description = "Reparte una nueva carta del mazo." ) ]
17      public string RepartirCarta()
18      {
19          string carta = "2 2";
20
21          // obtiene el mazo del cliente
22          ArrayList mazo = ( ArrayList )( Session[ "mazo" ] );
23          carta = Convert.ToString( mazo[ 0 ] );
24          mazo.RemoveAt( 0 );
25          return carta;
26      } // fin del método RepartirCarta
27
28      // crea y baraja un mazo de cartas
29      [ WebMethod( EnableSession = true,
30         Description = "Crea y baraja un mazo de cartas." ) ]
31      public void Barajar()
32      {
33          object temporal; // almacena una carta temporalmente, durante el intercambio
34          Random objetoAleatorio = new Random(); // genera números aleatorios
35          int nuevoIndice; // índice de la carta seleccionada al azar
36          ArrayList mazo = new ArrayList(); // almacena el mazo de cartas (objetos string)
37
38          // genera todas las cartas posibles
39          for ( int i = 1; i <= 13; i++ ) // itera a través de los valores de las caras
40              for ( int j = 0; j <= 3; j++ ) // itera a través de los palos
41                  mazo.Add( i + " " + j ); // agrega carta (string) al mazo
42
43          // baraja el mazo, intercambiando cada carta con otra carta al azar
44          for ( int i = 0; i < mazo.Count; i++ )
45          {

```

Figura 22.18 | Servicio Web de Blackjack. (Parte I de 2).

```

46         // obtiene índice aleatorio
47         nuevoIndice = objetoAleatorio.Next( mazo.Count - 1 );
48         temporal = mazo[ i ]; // almacena la carta actual en la variable temporal
49         mazo[ i ] = mazo[ nuevoIndice ]; // copia la carta seleccionada al azar
50         mazo[nuevoIndice] = temporal; // copia la carta actual de vuelta al mazo
51     } // fin de for
52
53     // agrega este mazo al estado de la sesión del usuario
54     Session.Add( "mazo", mazo );
55 } // fin del método Barajar
56
57 // calcula el valor de la mano
58 [ WebMethod( Description =
59     "Calcula un valor numérico para la mano actual." ) ]
60 public int ObtenerValorMano( string repartidas )
61 {
62     // divide cadena que contiene todas las cartas
63     char[] tab = { '\t' };
64     string[] cartas = repartidas.Split( tab ); // obtiene arreglo de cartas
65     int total = 0; // valor total de las cartas en la mano
66     int cara; // cara de la carta actual
67     int cuentaAses = 0; // número de ases en la mano
68
69     // itera a través de las cartas en la mano
70     foreach ( string tomada in cartas )
71     {
72         // obtiene la cara de la carta
73         cara = Int32.Parse( tomada.Substring( 0, tomada.IndexOf( " " ) ) );
74
75         switch ( cara )
76         {
77             case 1: // si es as, incrementa cuentaAses
78                 cuentaAses++;
79                 break;
80             case 11: // si es joto, suma 10
81             case 12: // si es quina, suma 10
82             case 13: // si es rey, suma 10
83                 total += 10;
84                 break;
85             default: // en caso contrario, suma el valor de la cara
86                 total += cara;
87                 break;
88         } // fin de switch
89     } // fin de foreach
90
91     // si hay ases, calcula el total óptimo
92     if ( cuentaAses > 0 )
93     {
94         // si es posible contar un as como 11, y el resto como
95         // 1 cada uno, lo hace; en caso contrario, cuenta todos los ases como 1
96         if ( total + 11 + cuentaAses - 1 <= 21 )
97             total += 11 + cuentaAses - 1;
98         else
99             total += cuentaAses;
100     } // fin de if
101
102     return total;
103 } // fin del método ObtenerValorMano
104 } // fin de la clase ServicioBlackjack

```

Figura 22.18 | Servicio Web de Blackjack. (Parte 2 de 2).

este método. Esto se requiere sólo para los métodos que deben acceder a la información de la sesión. Al hacer esto, el servicio Web puede utilizar un objeto `HttpSessionState` (ASP.NET lo llama `Session`) para mantener el mazo de cartas para cada aplicación cliente que utilice este servicio Web (línea 22). Podemos usar `Session` para almacenar objetos para un cliente específico, entre las llamadas a los métodos. En el capítulo 21 hablamos con detalle sobre el estado de una sesión.

El método `RepartirCarta` extrae una carta del mazo y la envía al cliente. Sin usar una variable de sesión, el mazo de cartas tendría que pasarse de un lado a otro, con cada llamada al método. Al utilizar el estado de la sesión se facilita la llamada al método (no requiere argumentos), y se evita la sobrecarga de enviar el mazo varias veces a través de la red.

En este punto, nuestro servicio Web contiene métodos que utilizan variables de sesión. Sin embargo, el servicio Web aún no puede determinar cuáles variables de sesión pertenecen a cada usuario. Si dos clientes llaman con éxito al método `RepartirCarta`, se manipularía el mismo mazo. Para evitar este problema, el servicio Web crea de manera automática una cookie para identificar en forma única a cada cliente. Un cliente explorador Web que tiene habilitado el manejo de cookies las almacena en forma automática. Una aplicación cliente que no sea explorador Web y consuma este servicio, deberá crear un objeto **`CookieContainer`** para almacenar las cookies que envíe el servidor. En la sección 22.5.2 hablaremos sobre esto con más detalle, cuando examinemos el cliente del servicio Web de blackjack.

El método Web `RepartirCarta` (líneas 15-26) selecciona una carta del mazo y la envía al cliente. El método obtiene primero el mazo del usuario actual, en forma de un objeto `ArrayList`, del objeto `Session` del servicio Web (línea 22). Después de obtener el mazo del usuario, `RepartirCarta` extrae la carta superior del mazo (línea 24) y devuelve el valor de esa carta como un objeto `string` (línea 25).

El método `Barajar` (líneas 29-55) genera un objeto `ArrayList` que representa un mazo de cartas, lo baraja y almacena las cartas barajadas en el objeto `Session` del cliente. Las líneas 39-41 utilizan instrucciones `for` anidadas para generar objetos `string` de la forma "*cara palo*", para representar a todas las posibles cartas en un mazo. Las líneas 44-51 barajan el mazo, intercambiando cada carta con otra carta seleccionada al azar. La línea 54 agrega el objeto `ArrayList` al objeto `Session`, para mantener el mazo entre las llamadas al método desde un cliente específico.

El método `ObtenerValorMano` (líneas 58-103) determina el valor total de las cartas en una mano, tratando de obtener la puntuación más alta posible sin pasar de 21. Recuerde que un as puede contarse como 1 o como 11, y todas las cartas con cara cuentan como 10.

Como verá en la figura 22.19, la aplicación cliente mantiene una mano de cartas como un objeto `string`, en el que cada carta se separa mediante un carácter de tabulación. La línea 64 separa la mano de cartas (representada por `repartidas`) en cartas individuales mediante una llamada al método `string Split` y le pasa un arreglo que contiene los caracteres delimitadores (en este caso, sólo un tabulador). `Split` utiliza los caracteres delimitadores para separar los tokens en el objeto `string`. Las líneas 70-89 cuentan el valor de cada carta. La línea 73 recibe el primer entero (la cara) y utiliza ese valor en la instrucción `switch` (líneas 75-88). Si la carta es un as, el método incrementa a la variable `cuentaAses`. En breve veremos cómo se utiliza esta variable. Si la carta es un 11, 12 o 13 (joto, quina o rey), el método suma 10 al valor total de la mano (línea 83). Si la carta es cualquier otra, el método incrementa el total en base a su valor (línea 86).

Como un as puede tener uno de dos valores, se requiere lógica adicional para procesar los ases. Las líneas 92-100 del método `ObtenerValorMano` procesan los ases después de todas las demás cartas. Si una mano contiene varios ases, sólo uno puede contarse como 11 (si dos ases se cuentan cada uno como 11, la mano tendría un valor perdedor de 22). La condición en la línea 96 determina si contar un as como 11 y el resto como 1 producen un total que no excede a 21. Si esto es posible, la línea 97 ajusta el total de manera acorde. En caso contrario, la línea 99 ajusta el total, contando cada as como 1.

El método `ObtenerValorMano` maximiza el valor de las cartas actuales sin exceder a 21. Por ejemplo, imagine que el repartidor tiene un 7 y recibe un as. El nuevo total podría ser 8 o 18. No obstante, `ObtenerValorMano` siempre maximiza el valor de las cartas sin pasar de 21, por lo que el nuevo total es 18.

22.5.2 Consumo del servicio Web de Blackjack

Ahora utilizaremos el servicio Web de blackjack en una aplicación para Windows (figura 22.19). Esta aplicación utiliza una instancia de `ServicioBlackjack` (se declara en la línea 18 y se crea en la línea 47) para representar al repartidor. El servicio Web mantiene la cuenta de las cartas del jugador y del repartidor (es decir, todas las cartas que se han repartido).

```

1  // Fig. 22.19: Blackjack.cs
2  // Juego de Blackjack que utiliza el servicio Web Blackjack.
3  using System;
4  using System.Collections.Generic;
5  using System.ComponentModel;
6  using System.Data;
7  using System.Drawing;
8  using System.Text;
9  using System.Windows.Forms;
10 using System.Net;
11 using System.Collections;
12
13 namespace Blackjack
14 {
15     public partial class BlackjackForm : Form
16     {
17         // referencia al servicio Web
18         private localhost.ServicioBlackjack repartidor;
19
20         // cadena que representa las cartas del repartidor
21         private string cartasRepartidor;
22
23         // cadena que representa las cartas del jugador
24         private string cartasJugador;
25         private ArrayList cuadrosCartas; // lista de controles PictureBoxes para las
            imágenes de las cartas
26         private int cartaActualJugador; // número de carta actual del jugador
27         private int cartaActualRepartidor; // número de carta actual del repartidor
28
29         // enumeración que representa los posibles resultados del juego
30         public enum EstadoJuego
31         {
32             EMPATE, // el juego termina en un empate
33             PIERDE, // el jugador pierde
34             GANA, // el jugador gana
35             BLACKJACK // el jugador tiene blackjack
36         } // fin de enumeración EstadoJuego
37
38         public BlackjackForm()
39         {
40             InitializeComponent();
41         } // fin del constructor
42
43         // prepara el juego
44         private void BlackjackForm_Load( object sender, EventArgs e )
45         {
46             // crea instancia de objeto que permite la comunicación con el servicio Web
47             repartidor = new localhost.ServicioBlackjack();
48
49             // permite el estado de la sesión
50             repartidor.CookieContainer = new CookieContainer();
51             cuadrosCartas = new ArrayList();
52
53             // coloca controles PictureBox en cuadrosCartas
54             cuadrosCartas.Add( pictureBox1 );
55             cuadrosCartas.Add( pictureBox2 );
56             cuadrosCartas.Add( pictureBox3 );
57             cuadrosCartas.Add( pictureBox4 );
58             cuadrosCartas.Add( pictureBox5 );

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte I de 8).

```

59     cuadrosCartas.Add( pictureBox6 );
60     cuadrosCartas.Add( pictureBox7 );
61     cuadrosCartas.Add( pictureBox8 );
62     cuadrosCartas.Add( pictureBox9 );
63     cuadrosCartas.Add( pictureBox10 );
64     cuadrosCartas.Add( pictureBox11 );
65     cuadrosCartas.Add( pictureBox12 );
66     cuadrosCartas.Add( pictureBox13 );
67     cuadrosCartas.Add( pictureBox14 );
68     cuadrosCartas.Add( pictureBox15 );
69     cuadrosCartas.Add( pictureBox16 );
70     cuadrosCartas.Add( pictureBox17 );
71     cuadrosCartas.Add( pictureBox18 );
72     cuadrosCartas.Add( pictureBox19 );
73     cuadrosCartas.Add( pictureBox20 );
74     cuadrosCartas.Add( pictureBox21 );
75     cuadrosCartas.Add( pictureBox22 );
76 } // fin del método BlackjackForm_Load
77
78 // reparte cartas al repartidor, mientras su total sea menor que 17,
79 // después calcula el valor de cada mano y determina el ganador
80 private void RepartidorJuega()
81 {
82     // mientras el valor de la mano del repartidor sea menor a 17,
83     // el repartidor debe tomar cartas
84     while ( repartidor.ObtenerValorMano( cartasRepartidor ) < 17 )
85     {
86         cartasRepartidor += '\t' + repartidor.RepartirCarta(); // reparte
            nueva carta
87
88         // actualiza GUI para mostrar nueva carta
89         MostrarCarta( cartaActualRepartidor, "" );
90         cartaActualRepartidor++;
91         MessageBox.Show( "El repartidor toma una carta" );
92     } // fin de while
93
94     int totalRepartidor = repartidor.ObtenerValorMano( cartasRepartidor );
95     int totalJugador = repartidor.ObtenerValorMano( cartasJugador );
96
97     // si el repartidor se pasó, el jugador gana
98     if ( totalRepartidor > 21 )
99     {
100         JuegoTerminado( EstadoJuego.GANA );
101         return;
102     } // fin de if
103
104     // si el repartidor y el jugador no se han pasado de 21,
105     // el que tenga más puntos gana; si tienen la misma puntuación es empate.
106     if ( totalRepartidor > totalJugador )
107         JuegoTerminado( EstadoJuego.PIERDE );
108     else if ( totalJugador > totalRepartidor )
109         JuegoTerminado( EstadoJuego.GANA );
110     else
111         JuegoTerminado( EstadoJuego.EMPATE );
112 } // fin del método RepartidorJuega
113
114 // muestra la carta representada por valorCarta en el control PictureBox
    especificado
115 public void MostrarCarta( int carta, string valorCarta )

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 2 de 8).


```

116     {
117         // obtiene el control PictureBox apropiado del objeto ArrayList
118         PictureBox mostrarBox = ( PictureBox )( cuadrosCartas[ carta ] );
119
120         // si la cadena que representa a carta está vacía,
121         // establece mostrarBox para que muestre la parte posterior de carta
122         if ( valorCarta == "" )
123         {
124             mostrarBox.Image =
125                 Image.FromFile( "imagenes_blackjack/cartaposterior.png" );
126             return;
127         } // fin de if
128
129         // obtiene de valorCarta el valor de la cara de la carta
130         string cara = valorCarta.Substring( 0, valorCarta.IndexOf( " " ) );
131
132         // obtiene de valorCarta el palo de la carta
133         string palo =
134             valorCarta.Substring( valorCarta.IndexOf( " " ) + 1 );
135
136         char letraPalo; // letra del palo que se usa para formar el nombre del
            archivo de imagen
137
138         // determina la letra del palo de la carta
139         switch ( Convert.ToInt32( palo ) )
140         {
141             case 0: // tréboles
142                 letraPalo = 't';
143                 break;
144             case 1: // diamantes
145                 letraPalo = 'd';
146                 break;
147             case 2: // corazones
148                 letraPalo = 'c';
149                 break;
150             default: // espadas
151                 letraPalo = 'e';
152                 break;
153         } // fin de switch
154
155         // establece mostrarBox para mostrar la imagen apropiada
156         mostrarBox.Image = Image.FromFile(
157             "imagenes_blackjack/" + cara + letraPalo + ".png" );
158     } // fin del método MostrarCarta
159
160     // muestra todas las cartas del jugador y el
161     // mensaje apropiado del estado del juego
162     public void JuegoTerminado( EstadoJuego ganador )
163     {
164         char[] tab = { '\t' };
165         string[] cartas = cartasRepartidor.Split( tab );
166
167         // muestra todas las cartas del repartidor
168         for ( int i = 0; i < cartas.Length; i++ )
169             MostrarCarta( i, cartas[ i ] );
170
171         // muestra la imagen apropiada del estado
172         if ( ganador == EstadoJuego.EMPATE ) // empate
173             estadoPictureBox.Image =

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 3 de 8).


```

174         Image.FromFile( "imagenes_blackjack/empate.png" );
175     else if ( ganador == EstadoJuego.PIERDE ) // el jugador pierde
176         estadoPictureBox.Image =
177             Image.FromFile( "imagenes_blackjack/pierde.png" );
178     else if ( ganador == EstadoJuego.BLACKJACK )
179         // el jugador tiene blackjack
180         estadoPictureBox.Image =
181             Image.FromFile( "imagenes_blackjack/blackjack.png" );
182     else // el jugador gana
183         estadoPictureBox.Image =
184             Image.FromFile( "imagenes_blackjack/gana.png" );
185
186     // muestra los totales finales para repartidor y jugador
187     repartidorTotalLabel.Text =
188         "Repartidor: " + repartidor.ObtenerValorMano( cartasRepartidor );
189     jugadorTotalLabel.Text =
190         "Jugador: " + repartidor.ObtenerValorMano( cartasJugador );
191
192     // restablece los controles para un nuevo juego
193     plantarButton.Enabled = false;
194     pedirButton.Enabled = false;
195     repartirButton.Enabled = true;
196 } // fin del método JuegoTerminado
197
198 // reparte dos cartas al repartidor y dos al jugador
199 private void repartirButton_Click( object sender, EventArgs e )
200 {
201     string carta; // almacena una carta temporalmente, hasta que se agrega a una mano
202
203     // borra las imágenes de las cartas
204     foreach ( PictureBox imagenCarta in cuadrosCartas )
205         imagenCarta.Image = null;
206
207     estadoPictureBox.Image = null; // borra imagen de estado
208     repartidorTotalLabel.Text = ""; // borra el total final para el repartidor
209     jugadorTotalLabel.Text = ""; // borra el total final para el jugador
210
211     // crea un nuevo mazo barajado en el equipo remoto
212     repartidor.Barajar();
213
214     // reparte dos cartas al jugador
215     cartasJugador = repartidor.RepartirCarta(); // reparte una carta a la mano
    del jugador
216
217     // actualiza GUI para mostrar la nueva carta
218     MostrarCarta( 11, cartasJugador );
219     carta = repartidor.RepartirCarta(); // reparte una segunda carta
220     MostrarCarta( 12, carta ); // actualiza GUI para mostrar la nueva carta
221     cartasJugador += '\t' + carta; // agrega la segunda carta a la mano del jugador
222
223     // reparte dos cartas al repartidor, sólo muestra la cara de la primera carta
224     cartasRepartidor = repartidor.RepartirCarta(); // reparte una carta a la mano
    del repartidor
225     MostrarCarta( 0, cartasRepartidor ); // actualiza GUI para mostrar la
    nueva carta
226     carta = repartidor.RepartirCarta(); // reparte una segunda carta
227     MostrarCarta( 1, "" ); // actualiza GUI para mostrar la carta cara abajo
228     cartasRepartidor += '\t' + carta; // agrega la segunda carta a la mano del
    repartidor

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 4 de 8).

```

229
230     plantarButton.Enabled = true; // permite al jugador plantarse
231     pedirButton.Enabled = true; // permite al jugador pedir otra carta
232     repartirButton.Enabled = false; // deshabilita el botón Repartir
233
234     // determina el valor de las dos manos
235     int totalRepartidor = repartidor.ObtenerValorMano( cartasRepartidor );
236     int totalJugador = repartidor.ObtenerValorMano( cartasJugador );
237
238     // si las manos son iguales a 21, es un empate
239     if ( totalRepartidor == totalJugador && totalRepartidor == 21 )
240         JuegoTerminado( EstadoJuego.EMPATE );
241     else if ( totalRepartidor == 21 ) // si el repartidor tiene 21, gana
242         JuegoTerminado( EstadoJuego.PIERDE );
243     else if ( totalJugador == 21 ) // el jugador tiene blackjack
244         JuegoTerminado( EstadoJuego.BLACKJACK );
245
246     // la siguiente carta del repartidor tiene el índice 2 en cuadrosCartas
247     cartaActualRepartidor = 2;
248
249     // la siguiente carta del jugador tiene el índice 13 en cuadrosCartas
250     cartaActualJugador = 13;
251 } // fin del método repartirButton
252
253 // reparte otra carta al jugador
254 private void pedirButton_Click( object sender, EventArgs e )
255 {
256     // da otra carta al jugador
257     string carta = repartidor.RepartirCarta(); // reparte una nueva carta
258     cartasJugador += '\t' + carta; // agrega la nueva carta a la mano del jugador
259
260     // actualiza GUI para mostrar la nueva carta
261     MostrarCarta( cartaActualJugador, carta );
262     cartaActualJugador++;
263
264     // determina el valor de la mano del jugador
265     int total = repartidor.ObtenerValorMano( cartasJugador );
266
267     // si el jugador excede a 21, la casa gana
268     if ( total > 21 )
269         JuegoTerminado( EstadoJuego.PIERDE );
270
271     // si el jugador tiene 21,
272     // no pueden tomar más cartas, y el repartidor juega
273     if ( total == 21 )
274     {
275         pedirButton.Enabled = false;
276         RepartidorJuega();
277     } // fin de if
278 } // fin del método pedirButton_Click
279
280 // juega la mano del repartidor después de que el jugador elige plantarse
281 private void plantarButton_Click( object sender, EventArgs e )
282 {
283     plantarButton.Enabled = false; // deshabilita el botón Plantar
284     pedirButton.Enabled = false; // deshabilita el botón Pedir
285     repartirButton.Enabled = true; // rehabilita el botón Repartir
286     RepartidorJuega(); // el jugador elige plantarse, por lo que se juega la mano
    del repartidor

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 5 de 8).

```

287         } // fin del método plantarButton_Click
288     } // fin de la clase BlackjackForm
289 } // fin del espacio de nombres Blackjack

```

a) Se reparten las cartas iniciales al jugador y al repartidor cuando el usuario oprime el botón **Repartir**.

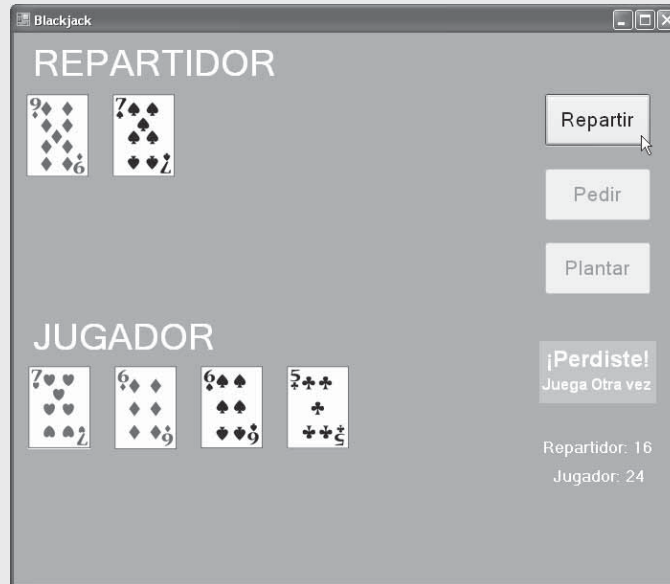


b) Las cartas, después de que el jugador oprime el botón **Pedir** dos veces, y después el botón **Plantar**. En este caso, el jugador ganó el juego debido a que el repartidor se pasó (se excedió de 21).



Figura 22.19 | Juego de blackjack que utiliza el servicio Web B1ackJack. (Parte 6 de 8).

c) Las cartas, después de que el jugador oprime el botón **Pedir** una vez, y luego el botón **Plantar**. En este caso, el jugador se pasó (se excedió de 21) y el repartidor ganó el juego.



d) Las cartas, después de que el jugador oprime el botón **Repartir**. En este caso, el jugador ganó con Blackjack, ya que las primeras dos cartas fueron un as y una carta con un valor de 10 (una reina en este caso).



Figura 22.19 | Juego de blackjack que utiliza el servicio Web B1ackJack. (Parte 7 de 8).

e) Las cartas, después de que el jugador oprime el botón **Plantar**. En este caso, el jugador y el repartidor empatan; tienen el mismo total de puntos.

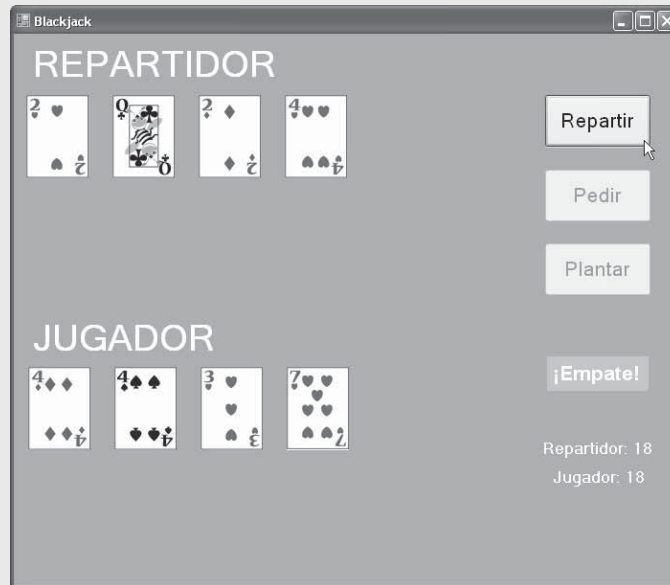


Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 8 de 8).

Cada jugador tiene 11 controles `PictureBox`; el número máximo de cartas que pueden repartirse sin excederse automáticamente de 21 (es decir, cuatro ases, cuatro números dos y tres números tres). Estos controles `PictureBox` se colocan en un objeto `ArrayList` (líneas 54-75), por lo que podemos indexar el objeto `ArrayList` durante el juego, para determinar el control `PictureBox` que mostrará la imagen de una carta específica.

En la sección 22.5.1, mencionamos que el cliente debe proporcionar una forma de aceptar las cookies creadas por el servicio Web, para identificar a los usuarios en forma única. La línea 50 en el manejador de eventos `Load` de `BlackjackForm` crea un nuevo objeto `CookieContainer` para la propiedad `CookieContainer` de `repartidor`. Un objeto `CookieContainer` (espacio de nombres `System.Net`) almacena la información de una cookie (creada por el servicio Web) en un objeto `Cookie`, en el objeto `CookieContainer`. El objeto `Cookie` contiene un identificador único, que el servicio Web puede usar para reconocer al cliente cuando éste realice solicitudes a futuro. Como parte de cada solicitud, la cookie se envía de vuelta al servidor en forma automática. Si el cliente no crea un objeto `CookieContainer`, el servicio Web crearía un nuevo objeto `Session` para cada solicitud, y la información de estado del usuario no persistiría de solicitud en solicitud.

El método `JuegoTerminado` (líneas 162-196) muestra todas las cartas del repartidor, muestra el mensaje apropiado en el control `PictureBox` de estado y muestra el total final de puntos del repartidor y del jugador. El método `JuegoTerminado` recibe como argumento a un miembro de la enumeración `EstadoJuego` (que se define en las líneas 30-36). La enumeración representa si el jugador empató, perdió o ganó el juego; sus cuatro miembros son `EMPATE`, `PIERDE`, `GANAR` y `BLACKJACK`.

Cuando el jugador hace clic en el botón **Repartir** (cuyo manejador de eventos aparece en las líneas 199-251), se borran todos los controles `PictureBox` y `Label` que muestran el total final de puntos. A continuación se baraja el mazo y tanto el jugador como el repartidor reciben dos cartas cada uno. Si ambos obtienen puntuaciones de 21, el programa llama al método `JuegoTerminado` y le pasa `EstadoJuego.PUSH`. Si sólo el jugador tiene 21 después de repartir las dos cartas, el programa pasa `EstadoJuego.BLACKJACK` al método `JuegoTerminado`. Si sólo el repartidor tiene 21, el programa pasa `EstadoJuego.PIERDE` al método `JuegoTerminado`.

Si `repartirButton_Click` no llama a `JuegoTerminado`, el jugador puede tomar más cartas, haciendo clic en el botón **Pedir**. El manejador de eventos para este botón se encuentra en las líneas 254-278. Cada vez que un jugador hace clic en **Pedir**, el programa reparte una carta más al jugador y la muestra en la GUI. Si el jugador se excede de 21, el juego se acaba y el jugador pierde. Si el jugador tiene exactamente 21, no puede tomar más cartas y

servicio Web, no necesita acceder al código de la clase proxy. Como veremos en la sección 22.4.5, puede invocar los métodos del objeto proxy como si fuera un objeto de la clase del servicio Web.

Los pasos que describimos en esta sección funcionan bien si conoce la referencia apropiada al servicio Web. No obstante, si está tratando de localizar un nuevo servicio Web, dos tecnologías comunes facilitan este proceso: *Descripción universal, descubrimiento e integración (UDDI)* y Descubrimiento de servicios Web (DISCO). En la sección 22.2 hablamos sobre DISCO. UDDI es un proyecto continuo para desarrollar un conjunto de especificaciones que definan la manera en que deben publicarse los servicios Web, para que los programadores en busca de servicios Web puedan encontrarlos. Microsoft y sus socios están trabajando en este proyecto para ayudar a los programadores a localizar servicios Web que se conformen a ciertas especificaciones, con lo cual los desarrolladores podrán buscar servicios Web a través de motores de búsqueda similares a Yahoo!® y Google™. En los sitios www.uddi.org y uddi.microsoft.com podrá aprender más acerca de UDDI y ver una demostración. Estos sitios contienen herramientas de búsqueda que hacen que sea conveniente buscar servicios Web.

22.4.5 Consumo del servicio Web EnteroEnorme

El formulario Windows Forms en la figura 22.17 utiliza el servicio Web EnteroEnorme para realizar cálculos con enteros positivos de hasta 100 dígitos de longitud. La línea 22 declara la variable `enteroRemoto` de tipo `localhost.EnteroEnorme`. Esta variable se utiliza en cada uno de los manejadores de eventos de la aplicación, para llamar a los métodos del servicio Web EnteroEnorme. El objeto proxy se crea y se asigna a esta variable en la línea 31 del manejador de eventos `Load` del formulario. Las líneas 52-53, 66-67, 95-96, 116-117 y 135-136 en los diversos manejadores de eventos de los botones invocan a los métodos del servicio Web. Observe que cada llamada se realiza en el objeto proxy local, que a su vez se comunica con el servicio Web a beneficio del cliente. Si descargó el ejemplo de www.deitel.com/books/csharpforprogrammers2, tal vez necesite regenerar el proxy; para ello, elimine la referencia Web y después agréguela de nuevo. Posteriormente haga clic con el botón derecho del ratón en `localhost` dentro de la carpeta **Web References** en el **Explorador de soluciones**, y seleccione la opción **Eliminar**. Después siga las instrucciones en la sección anterior para agregar la referencia Web al proyecto.

El usuario introduce dos enteros, cada uno de hasta 100 dígitos de largo. Al hacer clic en un botón, la aplicación invoca a un método Web para realizar la tarea apropiada y devolver el resultado. Observe que la aplicación cliente `UsoServicioEnteroEnorme` no puede realizar operaciones usando números de 100 dígitos de manera directa. En vez de eso, la aplicación crea representaciones `string` de estos números y los pasa como argumentos para los métodos Web que manejan dichas tareas para el cliente. Después utiliza el valor de retorno de cada operación para mostrar un mensaje apropiado.

```

1  // Fig. 22.17: UsoServicioWebEnteroEnorme.cs
2  // Uso del servicio Web EnteroEnorme.
3  using System;
4  using System.Collections.Generic;
5  using System.ComponentModel;
6  using System.Data;
7  using System.Drawing;
8  using System.Text;
9  using System.Windows.Forms;
10 using System.Web.Services.Protocols;
11
12 namespace UsoServicioWebEnteroEnorme
13 {
14     public partial class UsoServicioWebEnteroEnormeForm : Form
15     {
16         public UsoServicioWebEnteroEnormeForm()
17         {
18             InitializeComponent();
19         } // fin del constructor
20
21         // declara una referencia al servicio Web

```

Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte I de 5).

```

22     private localhost.EnteroEnorme enteroRemoto;
23
24     private char[] ceros = { '0' }; // carácter a eliminar de las cadenas
25
26     // instancia un objeto para interactuar con el servicio Web
27     private void UsoServicioEnteroEnormeForm_Load( object sender,
28         EventArgs e )
29     {
30         // crea instancia de enteroRemoto
31         enteroRemoto = new localhost.EnteroEnorme();
32     } // fin del método UsoServicioEnteroEnormeForm_Load
33
34     // suma dos enteros introducidos por el usuario
35     private void sumarButton_Click( object sender, EventArgs e )
36     {
37         // se asegura que los números no excedan de 100 dígitos y que ambos
38         // no sean de 100 dígitos de largo, lo cual produciría un desbordamiento
39         if ( primeroTextBox.Text.Length > 100 ||
40             segundoTextBox.Text.Length > 100 ||
41             ( primeroTextBox.Text.Length == 100 &&
42               segundoTextBox.Text.Length == 100 ) )
43         {
44             MessageBox.Show( "Los Enteros Enormes no deben tener " +
45                             "más de 100 dígitos\r\nAmbos enteros no pueden tener " +
46                             "longitud de 100: esto produce un desbordamiento", "Error",
47                             MessageBoxButtons.OK, MessageBoxIcon.Information );
48             return;
49         } // fin de if
50
51         // realiza la suma
52         resultadoLabel.Text = enteroRemoto.Sumar(
53             primeroTextBox.Text, segundoTextBox.Text ).TrimStart( ceros );
54     } // fin del método sumarButton_Click
55
56     // resta dos números introducidos por el usuario
57     private void restarButton_Click( object sender, EventArgs e )
58     {
59         // se asegura que los Enteros Enormes no excedan de 100 dígitos
60         if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
61             return;
62
63         // realiza la resta
64         try
65         {
66             string resultado = enteroRemoto.Restar(
67                 primeroTextBox.Text, segundoTextBox.Text ).TrimStart( ceros );
68
69             if ( resultado == "" )
70                 resultadoLabel.Text = "0";
71             else
72                 resultadoLabel.Text = resultado;
73         } // fin de try
74
75         // si el método Web lanza una excepción,
76         // entonces el primer argumento era menor que el segundo
77         catch ( SoapException exception )
78         {
79
80             MessageBox.Show(

```

Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte 2 de 5).

```

81         "El primer argumento era menor que el segundo" );
82     } // fin de catch
83 } // fin del método restarButton_Click
84
85 // determina si el primer número introducido
86 // por el usuario es mayor que el segundo
87 private void masGrandeButton_Click( object sender, EventArgs e )
88 {
89     // se asegura que los Enteros Enormes no excedan de 100 dígitos
90     if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
91         return;
92
93     // llama al método del servicio Web para determinar si
94     // el primer entero es más grande que el segundo
95     if ( enteroRemoto.MasGrande( primeroTextBox.Text,
96         segundoTextBox.Text ) )
97         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
98             " es mayor que " +
99             segundoTextBox.Text.TrimStart( ceros );
100     else
101         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
102             " no es mayor que " +
103             segundoTextBox.Text.TrimStart( ceros );
104 } // fin del método masGrandeButton_Click
105
106 // determina si el primer número introducido
107 // por el usuario es más pequeño que el segundo
108 private void masPequenioButton_Click( object sender, EventArgs e )
109 {
110     // se asegura que los Enteros Enormes no excedan de 100 dígitos
111     if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
112         return;
113
114     // llama al método del servicio Web para determinar si
115     // el primer entero es más pequeño que el segundo
116     if ( enteroRemoto.MasPequenio( primeroTextBox.Text,
117         segundoTextBox.Text ) )
118         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
119             " es menor que " +
120             segundoTextBox.Text.TrimStart( ceros );
121     else
122         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
123             " no es menor que " +
124             segundoTextBox.Text.TrimStart( ceros );
125 } // fin del método masPequenioButton_Click
126
127 // determina si dos números introducidos por el usuario son iguales
128 private void igualButton_Click( object sender, EventArgs e )
129 {
130     // se asegura que los Enteros Enormes no excedan de 100 dígitos
131     if ( ComprobarTamano( primeroTextBox, segundoTextBox ) )
132         return;
133
134     // llama al método del servicio Web para determinar si los enteros son iguales
135     if ( enteroRemoto.IgualA( primeroTextBox.Text,
136         segundoTextBox.Text ) )
137         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
138             " es igual a " + segundoTextBox.Text.TrimStart( ceros );
139     else

```

Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte 3 de 5).


```

140         resultadoLabel.Text = primeroTextBox.Text.TrimStart( ceros ) +
141         " no es igual a " +
142         segundoTextBox.Text.TrimStart( ceros );
143     } // fin del método igualButton_Click
144
145     // determina si los números introducidos por el usuario son demasiado grandes
146     private bool ComprobarTamano( TextBox primero, TextBox segundo )
147     {
148         // muestra un mensaje de error si cualquier número tiene demasiados dígitos
149         if ( ( primero.Text.Length > 100 ) ||
150             ( segundo.Text.Length > 100 ) )
151         {
152             MessageBox.Show( "Los Enteros Enormes deben ser menores de 100 dígitos" ,
153                             "Error", MessageBoxButtons.OK, MessageBoxIcon.Information );
154             return true;
155         } // fin de if
156
157         return false;
158     } // fin del método ComprobarTamano
159 } // fin de la clase UsingHugeIntegerServiceForm
160 } // fin del espacio de nombres UsoServicioWebEnteroEnorme

```

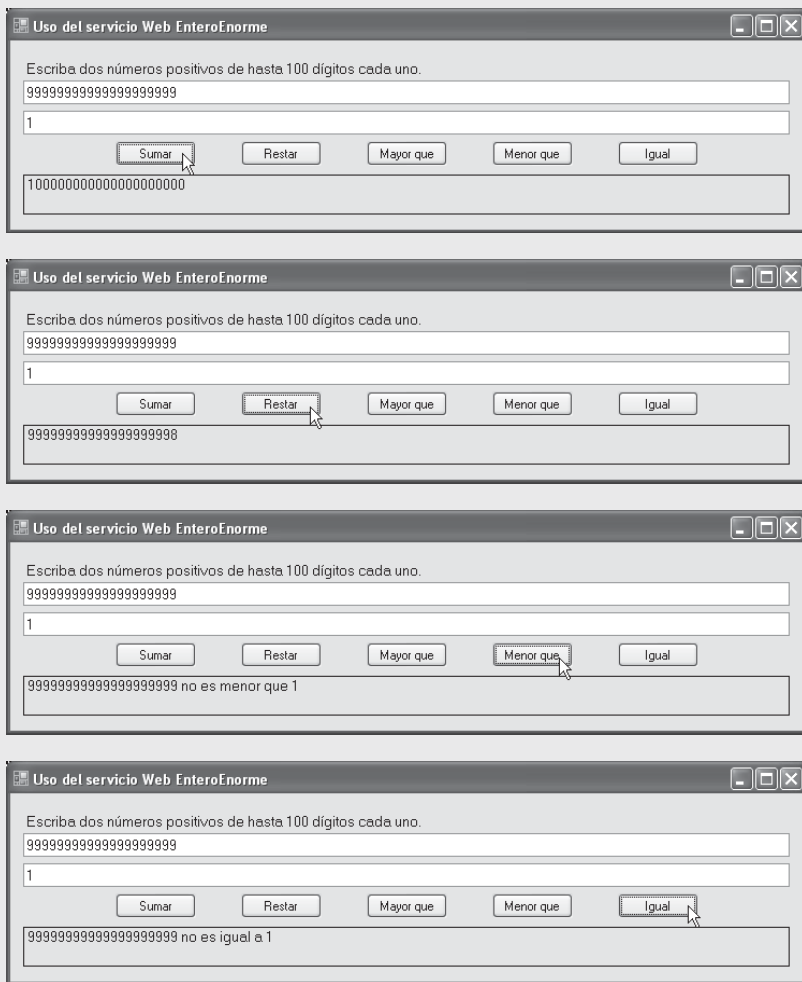


Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte 4 de 5).

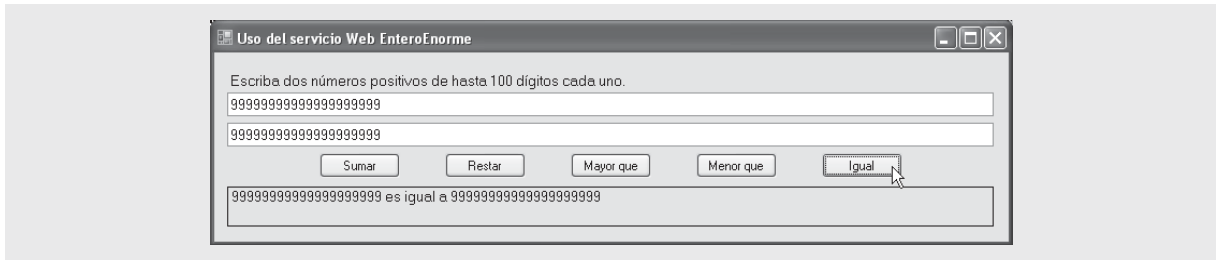


Figura 22.17 | Uso del servicio Web EnteroEnorme. (Parte 5 de 5).

Observe que la aplicación elimina los ceros a la izquierda en los números antes de mostrarlos, mediante una llamada al método `string TrimStart`. Al igual que el método `string Trim` (que vimos en el capítulo 16), `TrimStart` elimina todas las ocurrencias de los caracteres especificados por un arreglo `char` (línea 24) desde el principio de un objeto `string`.

22.5 Rastreo de sesiones en los servicios Web

En el capítulo 21, describimos las ventajas de mantener la información sobre los usuarios para personalizar sus experiencias. En especial, hablamos sobre el rastreo de sesiones mediante el uso de cookies y objetos `HttpSessionState`. Ahora incorporaremos el rastreo de sesiones a un servicio Web. Suponga que una aplicación cliente necesita llamar a varios métodos del mismo servicio Web, posiblemente varias veces a cada uno. En tal caso, sería benéfico para el servicio Web mantener la información de estado para el cliente. El rastreo de sesiones elimina la necesidad de pasar la información del cliente varias veces entre éste y el servicio Web. Por ejemplo, un servicio Web que proporciona acceso a las reseñas de restaurantes locales se beneficiaría de almacenar la dirección del usuario cliente. Una vez que se almacena la dirección del cliente en una variable de sesión, los métodos Web pueden devolver resultados personalizados y localizados, sin requerir que se pase la dirección en cada llamada a los métodos. Esto no sólo aumenta el rendimiento, sino que también requiere un menor esfuerzo por parte del programador; se pasa menos información en cada llamada a los métodos.

22.5.1 Creación de un servicio Web de Blackjack

Almacenar la información de las sesiones puede hacer que el servicio Web sea más intuitivo para los programadores de la aplicación cliente. Nuestro siguiente ejemplo es un servicio Web que asiste a los programadores en el desarrollo de un juego de cartas de blackjack (figura 22.18). El servicio Web proporciona métodos Web para repartir una carta y evaluar una mano de cartas. Después de presentar este servicio Web, lo utilizamos para que sirva como el repartidor para un juego de blackjack (figura 22.19). El servicio Web de blackjack utiliza una variable de sesión para mantener un mazo único de cartas para cada aplicación cliente. Varios clientes pueden utilizar el servicio al mismo tiempo, pero las llamadas a los métodos Web que realice un cliente específico sólo utilizarán el mazo almacenado en la sesión de ese cliente. Nuestro ejemplo utiliza un subconjunto simple de las reglas de blackjack de los casinos:

Se reparten dos cartas al repartidor y dos cartas al jugador. Las cartas del jugador se reparten con la cara hacia arriba. Sólo la primera de las cartas del repartidor se reparte con la cara hacia arriba. Cada carta tiene un valor. Una carta numerada del 2 al 10 vale lo que indique en su cara. Los jotos, reinas y reyes valen 10 cada uno. Los ases pueden valer 1 u 11, lo que sea más conveniente para el jugador (como pronto veremos). Si la suma de las dos cartas iniciales del jugador es 21 (por ejemplo, que se hayan repartido al jugador una carta con valor de 10 y un as, que cuenta como 11 en esta situación), el jugador tiene “blackjack” y gana el juego de inmediato. En caso contrario, el jugador puede empezar a tomar cartas adicionales, una a la vez. Estas cartas se reparten con la cara hacia arriba, y el jugador decide cuándo dejar de tomar cartas. Si el jugador “se pasa” (es decir, si la suma de las cartas del jugador excede a 21), el juego termina y el jugador pierde. Cuando el jugador está satisfecho con su conjunto actual de cartas, “se planta” (es decir, deja de tomar cartas) y se revela la carta oculta del repartidor. Si el total del repartidor es 16 o menos, debe tomar otra carta; en caso contrario, el repartidor debe plantarse. El repartidor debe seguir tomando cartas hasta que la suma de todas sus cartas sea mayor o igual a 17. Si el repartidor se pasa de 21, el jugador gana. En caso contrario, la mano con el total de puntos que sea mayor gana. Si el repartidor y el jugador tienen el mismo total de puntos, el juego es un “empate” y nadie gana.

El servicio Web (figura 22.18) proporciona métodos para repartir una carta y determinar el valor en puntos de una mano. Representamos cada carta como un objeto `string` que consiste en un dígito (por ejemplo, del 1 al 13), representando la cara de la carta (por ejemplo del as hasta el rey), seguido por un espacio y un dígito (por ejemplo 0-3) el cual representa el palo de la carta (por ejemplo, tréboles, diamantes, corazones o espadas). Por ejemplo, el joto de corazones se representa como "11 2", y el dos de tréboles se representa como "2 0". Después de desplegar el servicio Web, creamos una aplicación para Windows que utiliza los métodos del servicio Web `ServicioBlackjack` para implementar un juego de blackjack. Para crear y desplegar este servicio Web, siga los pasos que presentamos en las secciones 22.4.2-22.4.3 para el servicio `EnteroEnorme`.

Las líneas 15-16 definen el método `RepartirCarta` como un método Web. Al establecer la propiedad `EnableSession` a `True`, se indica que debe mantenerse la información de sesión y que debe estar accesible para

```

1  // Fig. 22.18: ServicioBlackjack.cs
2  // Servicio Web de Blackjack, para repartir y contar cartas.
3  using System;
4  using System.Web;
5  using System.Web.Services;
6  using System.Web.Services.Protocols;
7  using System.Collections;
8
9  [ WebService( Namespace = "http://www.deitel.com/", Description =
10     "Un servicio Web que reparte y cuenta cartas para el juego de Blackjack" ) ]
11  [ WebServiceBinding( ConformsTo = WsiProfiles.BasicProfile1_1 ) ]
12  public class ServicioBlackjack : System.Web.Services.WebService
13  {
14      // reparte una carta que no se ha repartido todavía
15      [ WebMethod( EnableSession = true,
16         Description = "Reparte una nueva carta del mazo." ) ]
17      public string RepartirCarta()
18      {
19          string carta = "2 2";
20
21          // obtiene el mazo del cliente
22          ArrayList mazo = ( ArrayList )( Session[ "mazo" ] );
23          carta = Convert.ToString( mazo[ 0 ] );
24          mazo.RemoveAt( 0 );
25          return carta;
26      } // fin del método RepartirCarta
27
28      // crea y baraja un mazo de cartas
29      [ WebMethod( EnableSession = true,
30         Description = "Crea y baraja un mazo de cartas." ) ]
31      public void Barajar()
32      {
33          object temporal; // almacena una carta temporalmente, durante el intercambio
34          Random objetoAleatorio = new Random(); // genera números aleatorios
35          int nuevoIndice; // índice de la carta seleccionada al azar
36          ArrayList mazo = new ArrayList(); // almacena el mazo de cartas (objetos string)
37
38          // genera todas las cartas posibles
39          for ( int i = 1; i <= 13; i++ ) // itera a través de los valores de las caras
40              for ( int j = 0; j <= 3; j++ ) // itera a través de los palos
41                  mazo.Add( i + " " + j ); // agrega carta (string) al mazo
42
43          // baraja el mazo, intercambiando cada carta con otra carta al azar
44          for ( int i = 0; i < mazo.Count; i++ )
45          {

```

Figura 22.18 | Servicio Web de Blackjack. (Parte I de 2).

```

46         // obtiene índice aleatorio
47         nuevoIndice = objetoAleatorio.Next( mazo.Count - 1 );
48         temporal = mazo[ i ]; // almacena la carta actual en la variable temporal
49         mazo[ i ] = mazo[ nuevoIndice ]; // copia la carta seleccionada al azar
50         mazo[nuevoIndice] = temporal; // copia la carta actual de vuelta al mazo
51     } // fin de for
52
53     // agrega este mazo al estado de la sesión del usuario
54     Session.Add( "mazo", mazo );
55 } // fin del método Barajar
56
57 // calcula el valor de la mano
58 [ WebMethod( Description =
59     "Calcula un valor numérico para la mano actual." ) ]
60 public int ObtenerValorMano( string repartidas )
61 {
62     // divide cadena que contiene todas las cartas
63     char[] tab = { '\t' };
64     string[] cartas = repartidas.Split( tab ); // obtiene arreglo de cartas
65     int total = 0; // valor total de las cartas en la mano
66     int cara; // cara de la carta actual
67     int cuentaAses = 0; // número de ases en la mano
68
69     // itera a través de las cartas en la mano
70     foreach ( string tomada in cartas )
71     {
72         // obtiene la cara de la carta
73         cara = Int32.Parse( tomada.Substring( 0, tomada.IndexOf( " " ) ) );
74
75         switch ( cara )
76         {
77             case 1: // si es as, incrementa cuentaAses
78                 cuentaAses++;
79                 break;
80             case 11: // si es joto, suma 10
81             case 12: // si es quina, suma 10
82             case 13: // si es rey, suma 10
83                 total += 10;
84                 break;
85             default: // en caso contrario, suma el valor de la cara
86                 total += cara;
87                 break;
88         } // fin de switch
89     } // fin de foreach
90
91     // si hay ases, calcula el total óptimo
92     if ( cuentaAses > 0 )
93     {
94         // si es posible contar un as como 11, y el resto como
95         // 1 cada uno, lo hace; en caso contrario, cuenta todos los ases como 1
96         if ( total + 11 + cuentaAses - 1 <= 21 )
97             total += 11 + cuentaAses - 1;
98         else
99             total += cuentaAses;
100     } // fin de if
101
102     return total;
103 } // fin del método ObtenerValorMano
104 } // fin de la clase ServicioBlackjack

```

Figura 22.18 | Servicio Web de Blackjack. (Parte 2 de 2).

este método. Esto se requiere sólo para los métodos que deben acceder a la información de la sesión. Al hacer esto, el servicio Web puede utilizar un objeto `HttpSessionState` (ASP.NET lo llama `Session`) para mantener el mazo de cartas para cada aplicación cliente que utilice este servicio Web (línea 22). Podemos usar `Session` para almacenar objetos para un cliente específico, entre las llamadas a los métodos. En el capítulo 21 hablamos con detalle sobre el estado de una sesión.

El método `RepartirCarta` extrae una carta del mazo y la envía al cliente. Sin usar una variable de sesión, el mazo de cartas tendría que pasarse de un lado a otro, con cada llamada al método. Al utilizar el estado de la sesión se facilita la llamada al método (no requiere argumentos), y se evita la sobrecarga de enviar el mazo varias veces a través de la red.

En este punto, nuestro servicio Web contiene métodos que utilizan variables de sesión. Sin embargo, el servicio Web aún no puede determinar cuáles variables de sesión pertenecen a cada usuario. Si dos clientes llaman con éxito al método `RepartirCarta`, se manipularía el mismo mazo. Para evitar este problema, el servicio Web crea de manera automática una cookie para identificar en forma única a cada cliente. Un cliente explorador Web que tiene habilitado el manejo de cookies las almacena en forma automática. Una aplicación cliente que no sea explorador Web y consuma este servicio, deberá crear un objeto **`CookieContainer`** para almacenar las cookies que envíe el servidor. En la sección 22.5.2 hablaremos sobre esto con más detalle, cuando examinemos el cliente del servicio Web de blackjack.

El método Web `RepartirCarta` (líneas 15-26) selecciona una carta del mazo y la envía al cliente. El método obtiene primero el mazo del usuario actual, en forma de un objeto `ArrayList`, del objeto `Session` del servicio Web (línea 22). Después de obtener el mazo del usuario, `RepartirCarta` extrae la carta superior del mazo (línea 24) y devuelve el valor de esa carta como un objeto `string` (línea 25).

El método `Barajar` (líneas 29-55) genera un objeto `ArrayList` que representa un mazo de cartas, lo baraja y almacena las cartas barajadas en el objeto `Session` del cliente. Las líneas 39-41 utilizan instrucciones `for` anidadas para generar objetos `string` de la forma "*cara palo*", para representar a todas las posibles cartas en un mazo. Las líneas 44-51 barajan el mazo, intercambiando cada carta con otra carta seleccionada al azar. La línea 54 agrega el objeto `ArrayList` al objeto `Session`, para mantener el mazo entre las llamadas al método desde un cliente específico.

El método `ObtenerValorMano` (líneas 58-103) determina el valor total de las cartas en una mano, tratando de obtener la puntuación más alta posible sin pasar de 21. Recuerde que un as puede contarse como 1 o como 11, y todas las cartas con cara cuentan como 10.

Como verá en la figura 22.19, la aplicación cliente mantiene una mano de cartas como un objeto `string`, en el que cada carta se separa mediante un carácter de tabulación. La línea 64 separa la mano de cartas (representada por `repartidas`) en cartas individuales mediante una llamada al método `string Split` y le pasa un arreglo que contiene los caracteres delimitadores (en este caso, sólo un tabulador). `Split` utiliza los caracteres delimitadores para separar los tokens en el objeto `string`. Las líneas 70-89 cuentan el valor de cada carta. La línea 73 recibe el primer entero (la cara) y utiliza ese valor en la instrucción `switch` (líneas 75-88). Si la carta es un as, el método incrementa a la variable `cuentaAses`. En breve veremos cómo se utiliza esta variable. Si la carta es un 11, 12 o 13 (joto, quina o rey), el método suma 10 al valor total de la mano (línea 83). Si la carta es cualquier otra, el método incrementa el total en base a su valor (línea 86).

Como un as puede tener uno de dos valores, se requiere lógica adicional para procesar los ases. Las líneas 92-100 del método `ObtenerValorMano` procesan los ases después de todas las demás cartas. Si una mano contiene varios ases, sólo uno puede contarse como 11 (si dos ases se cuentan cada uno como 11, la mano tendría un valor perdedor de 22). La condición en la línea 96 determina si contar un as como 11 y el resto como 1 producen un total que no excede a 21. Si esto es posible, la línea 97 ajusta el total de manera acorde. En caso contrario, la línea 99 ajusta el total, contando cada as como 1.

El método `ObtenerValorMano` maximiza el valor de las cartas actuales sin exceder a 21. Por ejemplo, imagine que el repartidor tiene un 7 y recibe un as. El nuevo total podría ser 8 o 18. No obstante, `ObtenerValorMano` siempre maximiza el valor de las cartas sin pasar de 21, por lo que el nuevo total es 18.

22.5.2 Consumo del servicio Web de Blackjack

Ahora utilizaremos el servicio Web de blackjack en una aplicación para Windows (figura 22.19). Esta aplicación utiliza una instancia de `ServicioBlackjack` (se declara en la línea 18 y se crea en la línea 47) para representar al repartidor. El servicio Web mantiene la cuenta de las cartas del jugador y del repartidor (es decir, todas las cartas que se han repartido).

```

1  // Fig. 22.19: Blackjack.cs
2  // Juego de Blackjack que utiliza el servicio Web Blackjack.
3  using System;
4  using System.Collections.Generic;
5  using System.ComponentModel;
6  using System.Data;
7  using System.Drawing;
8  using System.Text;
9  using System.Windows.Forms;
10 using System.Net;
11 using System.Collections;
12
13 namespace Blackjack
14 {
15     public partial class BlackjackForm : Form
16     {
17         // referencia al servicio Web
18         private localhost.ServicioBlackjack repartidor;
19
20         // cadena que representa las cartas del repartidor
21         private string cartasRepartidor;
22
23         // cadena que representa las cartas del jugador
24         private string cartasJugador;
25         private ArrayList cuadrosCartas; // lista de controles PictureBoxes para las
            imágenes de las cartas
26         private int cartaActualJugador; // número de carta actual del jugador
27         private int cartaActualRepartidor; // número de carta actual del repartidor
28
29         // enumeración que representa los posibles resultados del juego
30         public enum EstadoJuego
31         {
32             EMPATE, // el juego termina en un empate
33             PIERDE, // el jugador pierde
34             GANA, // el jugador gana
35             BLACKJACK // el jugador tiene blackjack
36         } // fin de enumeración EstadoJuego
37
38         public BlackjackForm()
39         {
40             InitializeComponent();
41         } // fin del constructor
42
43         // prepara el juego
44         private void BlackjackForm_Load( object sender, EventArgs e )
45         {
46             // crea instancia de objeto que permite la comunicación con el servicio Web
47             repartidor = new localhost.ServicioBlackjack();
48
49             // permite el estado de la sesión
50             repartidor.CookieContainer = new CookieContainer();
51             cuadrosCartas = new ArrayList();
52
53             // coloca controles PictureBox en cuadrosCartas
54             cuadrosCartas.Add( pictureBox1 );
55             cuadrosCartas.Add( pictureBox2 );
56             cuadrosCartas.Add( pictureBox3 );
57             cuadrosCartas.Add( pictureBox4 );
58             cuadrosCartas.Add( pictureBox5 );

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte I de 8).

```

59     cuadrosCartas.Add( pictureBox6 );
60     cuadrosCartas.Add( pictureBox7 );
61     cuadrosCartas.Add( pictureBox8 );
62     cuadrosCartas.Add( pictureBox9 );
63     cuadrosCartas.Add( pictureBox10 );
64     cuadrosCartas.Add( pictureBox11 );
65     cuadrosCartas.Add( pictureBox12 );
66     cuadrosCartas.Add( pictureBox13 );
67     cuadrosCartas.Add( pictureBox14 );
68     cuadrosCartas.Add( pictureBox15 );
69     cuadrosCartas.Add( pictureBox16 );
70     cuadrosCartas.Add( pictureBox17 );
71     cuadrosCartas.Add( pictureBox18 );
72     cuadrosCartas.Add( pictureBox19 );
73     cuadrosCartas.Add( pictureBox20 );
74     cuadrosCartas.Add( pictureBox21 );
75     cuadrosCartas.Add( pictureBox22 );
76 } // fin del método BlackjackForm_Load
77
78 // reparte cartas al repartidor, mientras su total sea menor que 17,
79 // después calcula el valor de cada mano y determina el ganador
80 private void RepartidorJuega()
81 {
82     // mientras el valor de la mano del repartidor sea menor a 17,
83     // el repartidor debe tomar cartas
84     while ( repartidor.ObtenerValorMano( cartasRepartidor ) < 17 )
85     {
86         cartasRepartidor += '\t' + repartidor.RepartirCarta(); // reparte
            nueva carta
87
88         // actualiza GUI para mostrar nueva carta
89         MostrarCarta( cartaActualRepartidor, "" );
90         cartaActualRepartidor++;
91         MessageBox.Show( "El repartidor toma una carta" );
92     } // fin de while
93
94     int totalRepartidor = repartidor.ObtenerValorMano( cartasRepartidor );
95     int totalJugador = repartidor.ObtenerValorMano( cartasJugador );
96
97     // si el repartidor se pasó, el jugador gana
98     if ( totalRepartidor > 21 )
99     {
100         JuegoTerminado( EstadoJuego.GANA );
101         return;
102     } // fin de if
103
104     // si el repartidor y el jugador no se han pasado de 21,
105     // el que tenga más puntos gana; si tienen la misma puntuación es empate.
106     if ( totalRepartidor > totalJugador )
107         JuegoTerminado( EstadoJuego.PIERDE );
108     else if ( totalJugador > totalRepartidor )
109         JuegoTerminado( EstadoJuego.GANA );
110     else
111         JuegoTerminado( EstadoJuego.EMPATE );
112 } // fin del método RepartidorJuega
113
114 // muestra la carta representada por valorCarta en el control PictureBox
    especificado
115 public void MostrarCarta( int carta, string valorCarta )

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 2 de 8).


```

116     {
117         // obtiene el control PictureBox apropiado del objeto ArrayList
118         PictureBox mostrarBox = ( PictureBox )( cuadrosCartas[ carta ] );
119
120         // si la cadena que representa a carta está vacía,
121         // establece mostrarBox para que muestre la parte posterior de carta
122         if ( valorCarta == "" )
123         {
124             mostrarBox.Image =
125                 Image.FromFile( "imagenes_blackjack/cartaposterior.png" );
126             return;
127         } // fin de if
128
129         // obtiene de valorCarta el valor de la cara de la carta
130         string cara = valorCarta.Substring( 0, valorCarta.IndexOf( " " ) );
131
132         // obtiene de valorCarta el palo de la carta
133         string palo =
134             valorCarta.Substring( valorCarta.IndexOf( " " ) + 1 );
135
136         char letraPalo; // letra del palo que se usa para formar el nombre del
            archivo de imagen
137
138         // determina la letra del palo de la carta
139         switch ( Convert.ToInt32( palo ) )
140         {
141             case 0: // tréboles
142                 letraPalo = 't';
143                 break;
144             case 1: // diamantes
145                 letraPalo = 'd';
146                 break;
147             case 2: // corazones
148                 letraPalo = 'c';
149                 break;
150             default: // espadas
151                 letraPalo = 'e';
152                 break;
153         } // fin de switch
154
155         // establece mostrarBox para mostrar la imagen apropiada
156         mostrarBox.Image = Image.FromFile(
157             "imagenes_blackjack/" + cara + letraPalo + ".png" );
158     } // fin del método MostrarCarta
159
160     // muestra todas las cartas del jugador y el
161     // mensaje apropiado del estado del juego
162     public void JuegoTerminado( EstadoJuego ganador )
163     {
164         char[] tab = { '\t' };
165         string[] cartas = cartasRepartidor.Split( tab );
166
167         // muestra todas las cartas del repartidor
168         for ( int i = 0; i < cartas.Length; i++ )
169             MostrarCarta( i, cartas[ i ] );
170
171         // muestra la imagen apropiada del estado
172         if ( ganador == EstadoJuego.EMPATE ) // empate
173             estadoPictureBox.Image =

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 3 de 8).


```

174         Image.FromFile( "imagenes_blackjack/empate.png" );
175     else if ( ganador == EstadoJuego.PIERDE ) // el jugador pierde
176         estadoPictureBox.Image =
177             Image.FromFile( "imagenes_blackjack/pierde.png" );
178     else if ( ganador == EstadoJuego.BLACKJACK )
179         // el jugador tiene blackjack
180         estadoPictureBox.Image =
181             Image.FromFile( "imagenes_blackjack/blackjack.png" );
182     else // el jugador gana
183         estadoPictureBox.Image =
184             Image.FromFile( "imagenes_blackjack/gana.png" );
185
186     // muestra los totales finales para repartidor y jugador
187     repartidorTotalLabel.Text =
188         "Repartidor: " + repartidor.ObtenerValorMano( cartasRepartidor );
189     jugadorTotalLabel.Text =
190         "Jugador: " + repartidor.ObtenerValorMano( cartasJugador );
191
192     // restablece los controles para un nuevo juego
193     plantarButton.Enabled = false;
194     pedirButton.Enabled = false;
195     repartirButton.Enabled = true;
196 } // fin del método JuegoTerminado
197
198 // reparte dos cartas al repartidor y dos al jugador
199 private void repartirButton_Click( object sender, EventArgs e )
200 {
201     string carta; // almacena una carta temporalmente, hasta que se agrega a una mano
202
203     // borra las imágenes de las cartas
204     foreach ( PictureBox imagenCarta in cuadrosCartas )
205         imagenCarta.Image = null;
206
207     estadoPictureBox.Image = null; // borra imagen de estado
208     repartidorTotalLabel.Text = ""; // borra el total final para el repartidor
209     jugadorTotalLabel.Text = ""; // borra el total final para el jugador
210
211     // crea un nuevo mazo barajado en el equipo remoto
212     repartidor.Barajar();
213
214     // reparte dos cartas al jugador
215     cartasJugador = repartidor.RepartirCarta(); // reparte una carta a la mano
        del jugador
216
217     // actualiza GUI para mostrar la nueva carta
218     MostrarCarta( 11, cartasJugador );
219     carta = repartidor.RepartirCarta(); // reparte una segunda carta
220     MostrarCarta( 12, carta ); // actualiza GUI para mostrar la nueva carta
221     cartasJugador += '\t' + carta; // agrega la segunda carta a la mano del jugador
222
223     // reparte dos cartas al repartidor, sólo muestra la cara de la primera carta
224     cartasRepartidor = repartidor.RepartirCarta(); // reparte una carta a la mano
        del repartidor
225     MostrarCarta( 0, cartasRepartidor ); // actualiza GUI para mostrar la
        nueva carta
226     carta = repartidor.RepartirCarta(); // reparte una segunda carta
227     MostrarCarta( 1, "" ); // actualiza GUI para mostrar la carta cara abajo
228     cartasRepartidor += '\t' + carta; // agrega la segunda carta a la mano del
        repartidor

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web Blackjack. (Parte 4 de 8).

```

229
230     plantarButton.Enabled = true; // permite al jugador plantarse
231     pedirButton.Enabled = true; // permite al jugador pedir otra carta
232     repartirButton.Enabled = false; // deshabilita el botón Repartir
233
234     // determina el valor de las dos manos
235     int totalRepartidor = repartidor.ObtenerValorMano( cartasRepartidor );
236     int totalJugador = repartidor.ObtenerValorMano( cartasJugador );
237
238     // si las manos son iguales a 21, es un empate
239     if ( totalRepartidor == totalJugador && totalRepartidor == 21 )
240         JuegoTerminado( EstadoJuego.EMPATE );
241     else if ( totalRepartidor == 21 ) // si el repartidor tiene 21, gana
242         JuegoTerminado( EstadoJuego.PIERDE );
243     else if ( totalJugador == 21 ) // el jugador tiene blackjack
244         JuegoTerminado( EstadoJuego.BLACKJACK );
245
246     // la siguiente carta del repartidor tiene el índice 2 en cuadrosCartas
247     cartaActualRepartidor = 2;
248
249     // la siguiente carta del jugador tiene el índice 13 en cuadrosCartas
250     cartaActualJugador = 13;
251 } // fin del método repartirButton
252
253 // reparte otra carta al jugador
254 private void pedirButton_Click( object sender, EventArgs e )
255 {
256     // da otra carta al jugador
257     string carta = repartidor.RepartirCarta(); // reparte una nueva carta
258     cartasJugador += '\t' + carta; // agrega la nueva carta a la mano del jugador
259
260     // actualiza GUI para mostrar la nueva carta
261     MostrarCarta( cartaActualJugador, carta );
262     cartaActualJugador++;
263
264     // determina el valor de la mano del jugador
265     int total = repartidor.ObtenerValorMano( cartasJugador );
266
267     // si el jugador excede a 21, la casa gana
268     if ( total > 21 )
269         JuegoTerminado( EstadoJuego.PIERDE );
270
271     // si el jugador tiene 21,
272     // no pueden tomar más cartas, y el repartidor juega
273     if ( total == 21 )
274     {
275         pedirButton.Enabled = false;
276         RepartidorJuega();
277     } // fin de if
278 } // fin del método pedirButton_Click
279
280 // juega la mano del repartidor después de que el jugador elige plantarse
281 private void plantarButton_Click( object sender, EventArgs e )
282 {
283     plantarButton.Enabled = false; // deshabilita el botón Plantar
284     pedirButton.Enabled = false; // deshabilita el botón Pedir
285     repartirButton.Enabled = true; // rehabilita el botón Repartir
286     RepartidorJuega(); // el jugador elige plantarse, por lo que se juega la mano
    del repartidor

```

Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 5 de 8).

```

287         } // fin del método plantarButton_Click
288     } // fin de la clase BlackjackForm
289 } // fin del espacio de nombres Blackjack

```

a) Se reparten las cartas iniciales al jugador y al repartidor cuando el usuario oprime el botón **Repartir**.

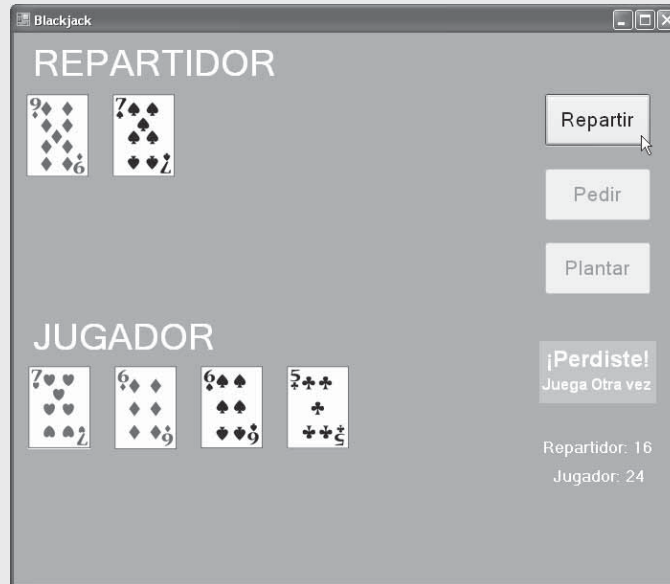


b) Las cartas, después de que el jugador oprime el botón **Pedir** dos veces, y después el botón **Plantar**. En este caso, el jugador ganó el juego debido a que el repartidor se pasó (se excedió de 21).



Figura 22.19 | Juego de blackjack que utiliza el servicio Web B1ackJack. (Parte 6 de 8).

c) Las cartas, después de que el jugador oprime el botón **Pedir** una vez, y luego el botón **Plantar**. En este caso, el jugador se pasó (se excedió de 21) y el repartidor ganó el juego.



d) Las cartas, después de que el jugador oprime el botón **Repartir**. En este caso, el jugador ganó con Blackjack, ya que las primeras dos cartas fueron un as y una carta con un valor de 10 (una quina en este caso).



Figura 22.19 | Juego de blackjack que utiliza el servicio Web B1ackJack. (Parte 7 de 8).

e) Las cartas, después de que el jugador oprime el botón **Plantar**. En este caso, el jugador y el repartidor empatan; tienen el mismo total de puntos.

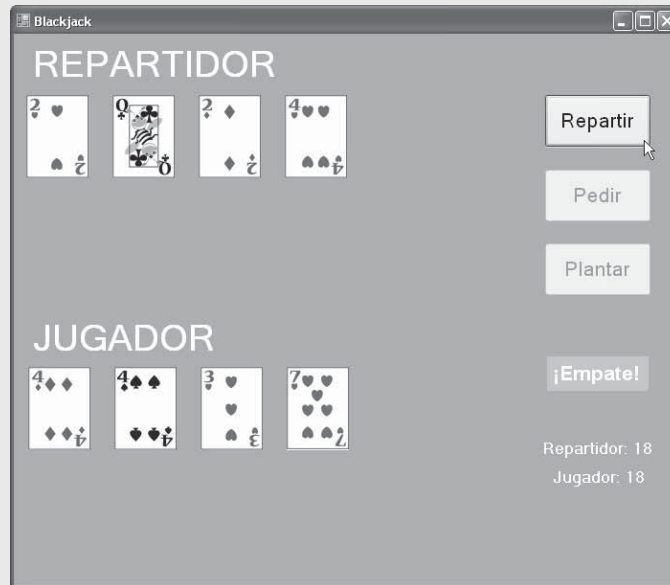


Figura 22.19 | Juego de blackjack que utiliza el servicio Web BlackJack. (Parte 8 de 8).

Cada jugador tiene 11 controles `PictureBox`; el número máximo de cartas que pueden repartirse sin excederse automáticamente de 21 (es decir, cuatro ases, cuatro números dos y tres números tres). Estos controles `PictureBox` se colocan en un objeto `ArrayList` (líneas 54-75), por lo que podemos indexar el objeto `ArrayList` durante el juego, para determinar el control `PictureBox` que mostrará la imagen de una carta específica.

En la sección 22.5.1, mencionamos que el cliente debe proporcionar una forma de aceptar las cookies creadas por el servicio Web, para identificar a los usuarios en forma única. La línea 50 en el manejador de eventos `Load` de `BlackjackForm` crea un nuevo objeto `CookieContainer` para la propiedad `CookieContainer` de `repartidor`. Un objeto `CookieContainer` (espacio de nombres `System.Net`) almacena la información de una cookie (creada por el servicio Web) en un objeto `Cookie`, en el objeto `CookieContainer`. El objeto `Cookie` contiene un identificador único, que el servicio Web puede usar para reconocer al cliente cuando éste realice solicitudes a futuro. Como parte de cada solicitud, la cookie se envía de vuelta al servidor en forma automática. Si el cliente no crea un objeto `CookieContainer`, el servicio Web crearía un nuevo objeto `Session` para cada solicitud, y la información de estado del usuario no persistiría de solicitud en solicitud.

El método `JuegoTerminado` (líneas 162-196) muestra todas las cartas del repartidor, muestra el mensaje apropiado en el control `PictureBox` de estado y muestra el total final de puntos del repartidor y del jugador. El método `JuegoTerminado` recibe como argumento a un miembro de la enumeración `EstadoJuego` (que se define en las líneas 30-36). La enumeración representa si el jugador empató, perdió o ganó el juego; sus cuatro miembros son `EMPATE`, `PIERDE`, `GANAR` y `BLACKJACK`.

Cuando el jugador hace clic en el botón **Repartir** (cuyo manejador de eventos aparece en las líneas 199-251), se borran todos los controles `PictureBox` y `Label` que muestran el total final de puntos. A continuación se baraja el mazo y tanto el jugador como el repartidor reciben dos cartas cada uno. Si ambos obtienen puntuaciones de 21, el programa llama al método `JuegoTerminado` y le pasa `EstadoJuego.PUSH`. Si sólo el jugador tiene 21 después de repartir las dos cartas, el programa pasa `EstadoJuego.BLACKJACK` al método `JuegoTerminado`. Si sólo el repartidor tiene 21, el programa pasa `EstadoJuego.PIERDE` al método `JuegoTerminado`.

Si `repartirButton_Click` no llama a `JuegoTerminado`, el jugador puede tomar más cartas, haciendo clic en el botón **Pedir**. El manejador de eventos para este botón se encuentra en las líneas 254-278. Cada vez que un jugador hace clic en **Pedir**, el programa reparte una carta más al jugador y la muestra en la GUI. Si el jugador se excede de 21, el juego se acaba y el jugador pierde. Si el jugador tiene exactamente 21, no puede tomar más cartas y

se hace una llamada al método `RepartidorJuega` (líneas 80-112) para que el repartidor continúe tomando cartas hasta que su mano tenga un valor de 17 o más (líneas 84-92). Si el repartidor se excede de 21, el jugador gana (línea 100); en cualquier otro caso se comparan los valores de las manos, y se hace una llamada a `JuegoTerminado` con el argumento apropiado (líneas 106-111).

Al hacer clic en el botón **Plantar** se indica que un jugador no quiere que se le reparta otra carta. El manejador de eventos para este botón (líneas 281-287) deshabilita los botones **Pedir** y **Plantar**, y después llama al método `RepartidorJuega`.

El método `MostrarCarta` (líneas 115-158) actualiza la GUI para mostrar una carta recién repartida. El método recibe como argumentos un entero que representa el índice del control `PictureBox` en el objeto `ArrayList` que debe establecer su imagen, y un objeto `string` que representa a la carta. Un objeto `string` vacío indica que deseamos mostrar la carta hacia abajo. Si el método `MostrarCarta` recibe un objeto `string` que no está vacío, el programa extrae la cara y el palo del objeto `string`, y utiliza esta información para buscar la imagen correcta. La instrucción `switch` (líneas 139-153) convierte el número que representa al palo en un entero, y asigna el carácter apropiado a `letraPalo` (t para tréboles, d para diamantes, c para corazones y e para espadas). El carácter en `letraPalo` se utiliza para completar el nombre del archivo de la imagen (líneas 156-157).

22.6 Uso de formularios Web Forms y servicios Web

Nuestros ejemplos anteriores acceden a los servicios Web desde aplicaciones Windows en Visual C# 2005. No obstante, podemos utilizarlos con la misma facilidad en aplicaciones Web creadas con Visual Web Developer. De hecho, como los comercios basados en Web están prevaleciendo cada vez más, es común que las aplicaciones Web consuman servicios Web. La figura 22.20 presenta un servicio Web de reservación de un aerolínea, el cual recibe información relacionada con el tipo de asiento que desea reservar un cliente, y hace una reservación si está disponible dicho asiento. Más adelante en esta sección, presentaremos una aplicación Web que permite a un cliente especificar una solicitud de reservación, y después utiliza el servicio Web de reservación de la aerolínea para tratar de ejecutar la solicitud.

El servicio Web de reservación de la aerolínea tiene un solo método Web: `Reservar` (líneas 24-42), el cual busca en su base de datos de asientos (`Boletos.mdf`) para localizar un asiento que coincida con la solicitud del usuario. Si encuentra un asiento apropiado, `Reservar` actualiza la base de datos, hace la reservación y devuelve `true`; en caso contrario, no se hace la reservación y el método devuelve `false`. Observe que las instrucciones en las líneas 28-29 y en la línea 37, que consultan y actualizan la base de datos, usan objetos de las clases `BoletosDataSet` y `BoletosDataSetTableAdapters.AsientosTableAdapter`. En el capítulo 20 vimos que las clases `DataSet` y `TableAdapter` se crean por usted cuando utiliza el **Diseñador de DataSet** para agregar un objeto `DataSet` a un proyecto. En la sección 22.6.1 hablaremos sobre los pasos para agregar el conjunto `BoletosDataSet`.

`Reservar` recibe dos argumentos: un objeto `string` que representa el tipo de asiento deseado (es decir, *Ventana*, *Intermedio* o *Pasillo*) y un objeto `string` que representa el tipo de clase deseada (es decir, *Económica*

```

1  // Fig. 22.20: ServicioReservacion.cs
2  // Servicio Web de reservación de una aerolínea.
3  using System;
4  using System.Web;
5  using System.Web.Services;
6  using System.Web.Services.Protocols;
7
8  [ WebService( Namespace = "http://www.deitel.com/", Description =
9    "Servicio que permite a un usuario reservar un asiento en un avión." ) ]
10 [ WebServiceBinding( ConformsTo = WsiProfiles.BasicProfile1_1 ) ]
11 public class ServicioReservacion : System.Web.Services.WebService
12 {
13     // crea objeto BoletosDataSet para colocar en caché los datos
14     // de la base de datos Boletos
15     private BoletosDataSet boletosDataSet = new BoletosDataSet();
16

```

Figura 22.20 | Servicio Web de reservación de una aerolínea. (Parte 1 de 2).

```

17 // crea AsientosTableAdapter para interactuar con la base de datos
18 private BoletosDataSetTableAdapters.AsientosTableAdapter
19     AsientosTableAdapter =
20     new BoletosDataSetTableAdapters.AsientosTableAdapter();
21
22 // comprueba la base de datos par determinar si hay un asiento disponible que
    coincide
23 [ WebMethod( Description = "Método para reservar un asiento." ) ]
24 public bool Reservar( string tipoAsiento, string tipoClase )
25 {
26     // llena BoletosDataSet.Asientos con filas que representan los asientos
27     // desocupados que coinciden con el tipoAsiento y tipoClase especificados
28     AsientosTableAdapter.LlenarPorTipoYClase(
29         boletosDataSet.Asientos, tipoAsiento, tipoClase );
30
31     // si el número de asientos devuelto es distinto de cero,
32     // obtiene el primer número de asiento que coincida y lo marca como ocupado
33     if ( boletosDataSet.Asientos.Count != 0 )
34     {
35         string numeroAsiento = boletosDataSet.Asientos[ 0 ].Numero;
36
37         AsientosTableAdapter.ActualizarAsientoComoOcupado( numeroAsiento );
38         return true; // el asiento se reservó
39     } // fin de if
40
41     return false; // el asiento no se reservó
42 } // fin del método Reservar
43 } // fin de la clase ServicioReservación

```

Figura 22.20 | Servicio Web de reservación de una aerolínea. (Parte 2 de 2).

o Primera). Nuestra base de datos contiene cuatro columnas: el número de asiento (es decir, 1-10), el tipo de asiento (Ventana, Intermedio o Pasillo), el tipo de clase (Económica o Primera) y una columna que contiene 1 (verdadero) o 0 (falso) para indicar si el asiento está ocupado. Las líneas 28-29 obtienen los números de asiento de cualquier asiento disponible que coincida con el asiento y tipo de clase solicitados. Esta instrucción llena la tabla Asientos en `boletosDataSet` con los resultados de la consulta

```

SELECT Numero
FROM Asientos
WHERE (Ocupado = 0) AND (Tipo = @tipo) AND (Clase = @clase)

```

Los parámetros `@tipo` y `@clase` en la consulta se sustituyen con los valores de los argumentos `tipoAsiento` y `tipoClase` del método `LlenarPorTipoYClase` de `AsientosTableAdapter`. En la línea 33, si el número de filas en la tabla Asientos (`boletosDataSet.Asientos.Count`) no es cero, hubo cuando menos un asiento que coincidió con la solicitud del usuario. En este caso, el servicio Web reserva el primer número de asiento que coincida. En la línea 35 obtenemos el número de asiento al acceder al primer elemento de la tabla Asientos (es decir, `Asientos[ 0 ]`: la primera fila en la tabla), y después obtener el valor de la columna `Numero` de esa fila. La línea 37 invoca al método `ActualizarAsientoComoOcupado` de `AsientosTableAdapter` y lo pasa a `numeroAsiento`, el asiento a reservar. El método `ActualizarAsientoComoOcupado` utiliza la instrucción UPDATE

```

UPDATE Asientos
SET Ocupado = 1
WHERE (Numero = @numero)

```

para marcar el asiento como ocupado en la base de datos, sustituyendo el parámetro `@numero` con el valor de `numeroAsiento`. El método `Reservar` devuelve `true` (línea 38) para indicar que la reservación fue exitosa. Si no hay asientos que coincidan (línea 33), `Reservar` devuelve `false` (línea 41) para indicar que ningún asiento coincidió con la solicitud del usuario.

22.6.1 Agregar componentes de datos a un servicio Web

Ahora usaremos las herramientas de Visual Web Developer para configurar un objeto DataSet que permita a nuestro servicio Web interactuar con el archivo de base de datos SQL Server Boletos.mdf que se proporciona en la carpeta de ejemplo para la figura 22.20. Agregaremos un nuevo objeto DataSet al proyecto, y después configuraremos el objeto TableAdapter de DataSet mediante el **Asistente para la configuración de TableAdapter**. Utilizaremos el asistente para seleccionar el origen de datos (Boletos.mdf) y para crear las instrucciones SQL necesarias para soportar las operaciones en la base de datos que vimos en la descripción de la figura 22.20. Los siguientes pasos para configurar el DataSet y su correspondiente TableAdapter son similares a los que vimos en los capítulos 20-21.

Paso 1: crear ServicioReservacion y agregar un DataSet al proyecto

Empiece por crear un proyecto tipo Servicio Web ASP.NET llamado ServicioReservacion. Cambie el nombre del archivo Service.cs por el de ServicioReservacion.cs y sustituya su código con el código de la figura 22.20. A continuación, agregue un DataSet llamado BoletosDataSet al proyecto. Haga clic con el botón derecho del ratón en la carpeta App_Code dentro del **Explorador de soluciones** y seleccione la opción **Agregar nuevo elemento...** del menú contextual. En el cuadro de diálogo **Agregar nuevo elemento**, seleccione DataSet, especifique BoletosDataSet.xsd en el campo **Nombre** y haga clic en **Agregar**. A continuación se mostrará el conjunto BoletosDataSet en modo de diseño y se abrirá el **Asistente para la configuración de TableAdapter**. Cuando agregamos un DataSet a un proyecto, el IDE crea las clases TableAdapter apropiadas para interactuar con las tablas de la base de datos.

Paso 2: seleccionar el origen de datos y crear una conexión

En los siguientes pasos usaremos el **Asistente para la configuración de Table Adapter**, para configurar un TableAdapter de manera que manipule la tabla Asientos en la base de datos Boletos.mdf. Ahora, debe seleccionar la base de datos. En el **Asistente para la configuración de TableAdapter**, haga clic en el botón **Nueva conexión...** para mostrar el cuadro de diálogo **Agregar conexión**. En este cuadro de diálogo, especifique **Archivo de base de datos de Microsoft SQL Server** como el **Origen de datos** y después haga clic en el botón **Examinar...** para mostrar el cuadro de diálogo **Seleccione el archivo de base de datos de SQL Server**. Localice el archivo Boletos.mdf en su computadora, selecciónelo y haga clic en el botón **Abrir** para regresar al cuadro de diálogo **Agregar conexión**. Haga clic en el botón **Probar conexión** para probar la conexión con la base de datos, y después haga clic en **Aceptar** para regresar al **Asistente para la configuración de TableAdapter**. Haga clic en el botón **Siguiente >** y después en **Sí** cuando el asistente le pregunte si desea agregar el archivo a su proyecto y modificar la conexión. Haga clic en **Siguiente >** para guardar la cadena de conexión en el archivo de configuración de la aplicación.

Paso 3: abrir el Generador de consultas y agregar la tabla Asientos de Boletos.mdf

Ahora debemos especificar cómo accederá el TableAdapter a la base de datos. Como en este ejemplo utilizaremos instrucciones SQL, elija **Usar instrucciones SQL** y después haga clic en **Siguiente >**. Ahora haga clic en **Generador de consultas...** para mostrar los cuadros de diálogo **Generador de consultas** y **Agregar tabla**. Antes de generar una consulta SQL, debemos especificar la(s) tabla(s) que se va(n) a utilizar en la consulta. La base de datos Boletos.mdf sólo contiene una tabla, llamada Asientos. Seleccione esta tabla de la ficha **Tablas** y haga clic en **Agregar**. Haga clic en **Cerrar** para cerrar el cuadro de diálogo **Agregar tabla**.

Paso 4: configurar una consulta SELECT para obtener los asientos disponibles

Ahora crearemos una consulta que seleccione los asientos que no estén ya reservados y que coincidan con un tipo y una clase específicos. Empiece por seleccionar **Numero** de la tabla **Asientos** en la parte superior del cuadro de diálogo **Generador de consultas**. A continuación, debemos especificar los criterios para seleccionar asientos. En la parte media del cuadro de diálogo **Generador de consultas**, haga clic en la celda debajo de **Numero** en la columna **Columna** y seleccione **Ocupado**. En la columna **Filtro** de esta fila, escriba 0 (es decir, falso) para indicar que debemos seleccionar sólo los números de los asientos que no estén ocupados. En la siguiente fila, seleccione **Tipo** en la columna **Columna** y especifique @tipo como el **Filtro**, para indicar que el valor del filtro se especificará en un argumento para el método que implementa esta consulta. En la siguiente fila, seleccione **Clase** en la columna **Columna** y especifique @clase como el **Filtro**, para indicar que el valor de este filtro también se especificará como un argumento para el método. Deseleccione las casillas de verificación en la columna **Resultados** para las

filas **Ocupado**, **Tipo** y **Clase**. Ahora el cuadro de diálogo **Generador de consultas** debe aparecer como se muestra en la figura 22.21. Haga clic en **Aceptar** para cerrar el cuadro de diálogo **Generador de consultas**. Haga clic en el botón **Siguiente >** para elegir los métodos que se van a generar. Para el nombre del método debajo de **Rellenar un DataTable**, escriba **LlenarPorTipoYClase**. Para el nombre del método debajo de **Devolver un DataTable**, escriba **ObtenerDatosPorTipoYClase**. Haga clic en el botón **Finalizar** para generar estos métodos.

Paso 5: agregar otra consulta a AsientosTableAdapter para el BoletosDataSet

Los últimos dos pasos que necesitamos realizar crean una consulta UPDATE que reserva un asiento. En el área de diseño para el **BoletosDataSet**, haga clic en **AsientosTableAdapter** para seleccionarlo, después haga clic con el botón derecho del ratón en este objeto y seleccione **Agregar consulta...** para mostrar el **Asistente para la configuración de consultas de TableAdapter**. Seleccione **Usar instrucciones SQL** y haga clic en el botón **Siguiente >**. Seleccione UPDATE como el tipo de la consulta y haga clic en el botón **Siguiente >**. Elimine la consulta UPDATE existente. Haga clic en **Generador de consultas...** para mostrar los cuadros de diálogo **Generador de consultas** y **Agregar tabla**. Después agregue la tabla **Asientos**, como hicimos en el *paso 3*, y haga clic en **Cerrar** para regresar al cuadro de diálogo **Generador de consultas**.

Paso 6: configurar una instrucción UPDATE para reservar un asiento

En el cuadro de diálogo **Generador de consultas**, seleccione la columna **Ocupada** de la tabla **Asientos** en la parte superior del cuadro de diálogo. En la parte media del cuadro de diálogo, coloque el valor 1 (es decir, verdadero) en la columna **Valor nuevo** para la fila **Ocupada**. En la fila debajo de **Ocupada**, seleccione **Numero**, desactive la casilla de verificación en la columna **Set** y especifique @numero como el valor de **Filtro**, para indicar que el número de asiento se especificará como argumento para el método que implementa esta consulta. Ahora el cuadro de diálogo **Generador de consultas** debe aparecer como se muestra en la figura 22.22. Haga clic en el botón **Aceptar** del cuadro de diálogo **Generador de consultas** para regresar al **Asistente para la configuración de consultas de TableAdapter**. Después haga clic en el botón **Siguiente >** para elegir el nombre del método que ejecutará la consulta

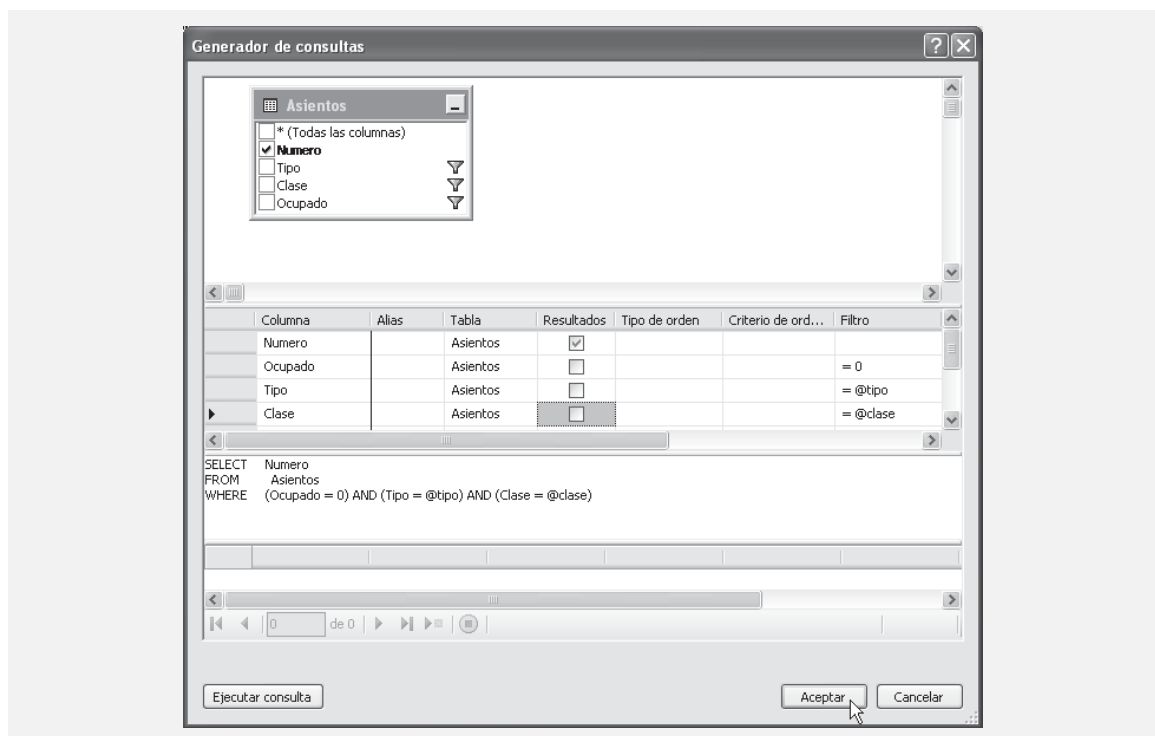


Figura 22.21 | El cuadro de diálogo **Generador de consultas** especifica una consulta SELECT, que selecciona los asientos que no están ya reservados y que coinciden con un tipo y una clase específicos.

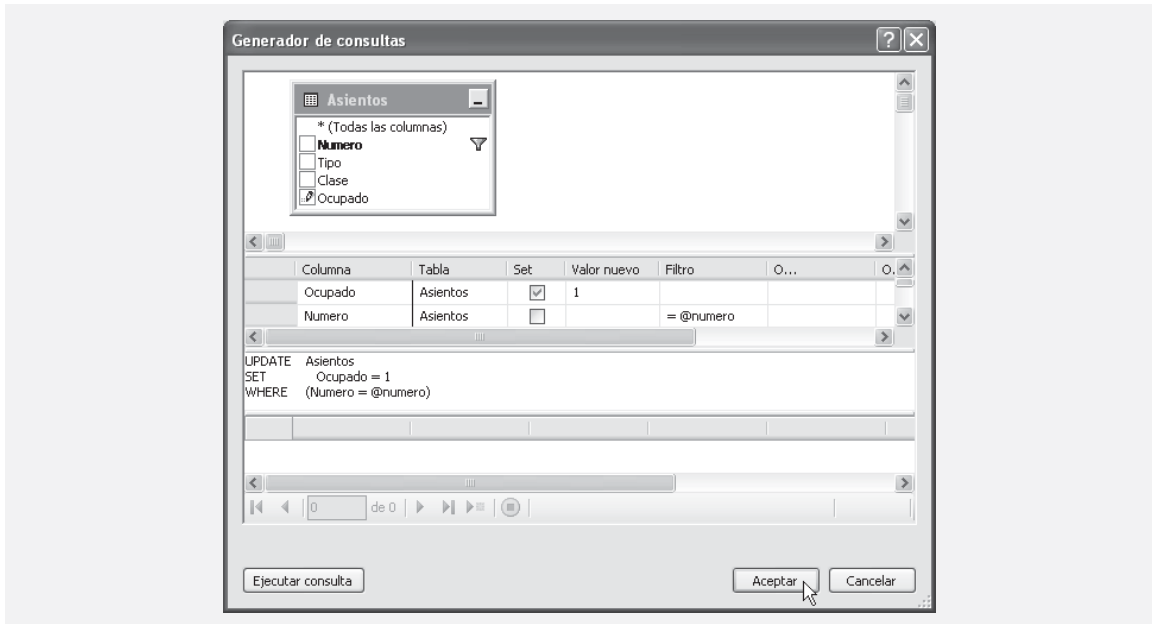


Figura 22.22 | El cuadro de diálogo **Generador de consultas** especifica una instrucción UPDATE que reserva un asiento.

UPDATE. Cambie el nombre del método por `ActualizarAsientoComoOcupado`, y después haga clic en **Finalizar** para cerrar el **Asistente para la configuración de consultas de TableAdapter**. En este punto, puede usar la página `ServicioReservacion.aspx` para probar el método `Reservar` del servicio Web. Para ello, seleccione **Iniciar sin depuración** del menú **Depurar**. En la sección 22.6.2 generaremos un formulario Web Forms para que consuma este servicio Web.

22.6.2 Creación de un formulario Web Forms para interactuar con el servicio Web de reservaciones de una aerolínea

La figura 22.23 presenta el listado ASPX para un formulario Web Forms, a través del cual los usuarios pueden seleccionar los tipos de asientos. Esta página permite a los usuarios reservar un asiento en base a su clase (Económica o Primera) y ubicación (Pasillo, Intermedio o Ventana) en una fila de asientos. Después, la página utiliza el servicio Web de reservación de la aerolínea para llevar a cabo las solicitudes del usuario. Si la solicitud de la base de datos no tiene éxito, se instruye al usuario para que modifique la solicitud e intente de nuevo.

Esta página define dos objetos `DropDownList` y un objeto `Button`. Un `DropDownList` (líneas 22-27) muestra todos los tipos de asientos que pueden seleccionar los usuarios. El segundo (líneas 29-32) proporciona opciones

```

1  <!-- Fig. 22.23: ClienteReservacion.aspx1 -->
2  <!-- Formulario Web que permite a los usuarios reservar asientos en un avión. -->
3  <%@ Page Language="C#" AutoEventWireup="true"
4     CodeFile="ClienteReservacion.aspx.cs"
5     Inherits="ClienteReservacion" %>
6
7  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
8     "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
9
10 <html xmlns="http://www.w3.org/1999/xhtml" >

```

Figura 22.23 | Archivo ASPX que recibe la información de la reservación. (Parte I de 2).

[illegible]

Figura 22.23 | Archivo ASPX que recibe la información de la reservación. (Parte 2 de 2).

para el tipo de clase. Los usuarios hacen clic en el control `Button` llamado `reservarButton` (líneas 34-36) para enviar solicitudes después de realizar selecciones de los objetos `DropDownList`. La página también define un objeto `Label` llamado `errorLabel`, que en un principio está en blanco (líneas 38-39) y que muestra un mensaje apropiado si no hay un asiento que coincida con la selección del usuario. El archivo de código subyacente (figura 22.24) adjunta un manejador de eventos a `reservarButton`.

```
1 // Fig. 22.24: ClienteReservacion.aspx.cs
2 // Archivo de código subyacente de ClienteReservacion.
3 using System;
4 using System.Data;
5 using System.Configuration;
6 using System.Web;
7 using System.Web.Security;
8 using System.Web.UI;
9 using System.Web.UI.WebControls;
10 using System.Web.UI.WebControls.WebParts;
11 using System.Web.UI.HtmlControls;
12
13 public partial class ClienteReservacion : System.Web.UI.Page
```

Figura 22.24 | Archivo de código subyacente para la página de reservación. (Parte I de 2).

```

14 {
15     // objeto de tipo proxy, que se utiliza para conectarse al servicio
        Web de reservación
16     private localhost.ServicioReservacion agenteBoletos =
17         new localhost.ServicioReservacion();
18
19     // trata de reservar el tipo de asiento seleccionado
20     protected void reservarButton_Click( object sender, EventArgs e )
21     {
22         // si el método Web devolvió verdadero, indica éxito
23         if ( agenteBoletos.Reservar( asientoList.SelectedItem.Text,
24             claseList.SelectedItem.Text.ToString() ) )
25         {
26             // oculta otros controles
27             instruccionesLabel.Visible = false;
28             asientoList.Visible = false;
29             claseList.Visible = false;
30             reservarButton.Visible = false;
31             errorLabel.Visible = false;
32
33             // muestra mensaje indicando éxito
34             Response.Write( "Su reservación está hecha. Gracias." );
35         } // fin de if
36         else // el método Web devolvió falso, por lo que indica falla
37         {
38             // muestra mensaje en el control errorLabel que al principio estaba en blanco
39             errorLabel.Text = "Este tipo de asiento no está disponible. " +
40                 "Modifique su solicitud e intente otra vez.";
41         } // fin de else
42     } // fin del método reservarButton_Click
43 } // fin de la clase ClienteReservacion

```

Figura 22.24 | Archivo de código subyacente para la página de reservación. (Parte 2 de 2).

Las líneas 16-17 de la figura 22.24 crean un objeto `ServicioReservacion`. (Recuerde que debe agregar una referencia Web a este servicio Web). Cuando el usuario hace clic en **Reservar** (figura 22.25), se ejecuta el manejador de eventos `reservarButton_Click` (líneas 20-42 de la figura 22.24) y se vuelve a cargar la página. El manejador de eventos llama al método `Reservar` del servicio Web y le pasa como argumentos el asiento seleccionado y el tipo de clase (líneas 23-24). Si `Reservar` devuelve `true`, la aplicación muestra un mensaje agradeciendo al usuario por hacer la reservación (línea 34); en caso contrario, `errorLabel` notifica al usuario que el tipo de asiento solicitado no está disponible, y le pide que intente de nuevo (líneas 39-40). Use las técnicas que presentamos en el capítulo 21 para generar el formulario Web Forms ASP.NET.

a) Seleccionar un asiento.

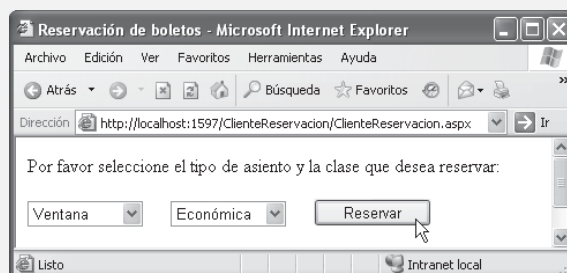
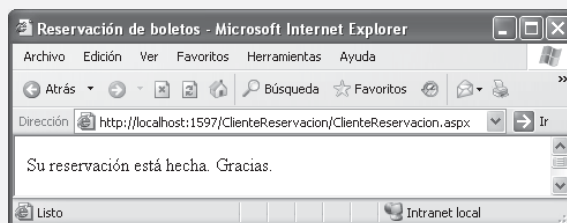
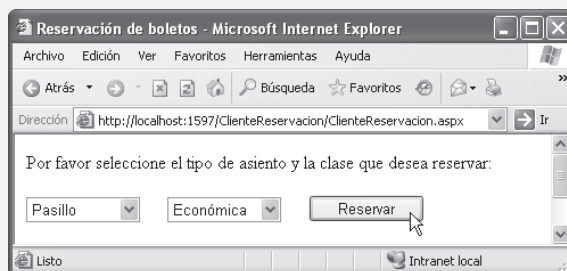


Figura 22.25 | Ejemplo de ejecución del formulario Web Forms para reservación de boletos. (Parte 1 de 2).

b) El asiento se reservó con éxito.



c) Tratar de reservar otro asiento.



d) No coincidió ningún asiento con el tipo y la clase solicitados.

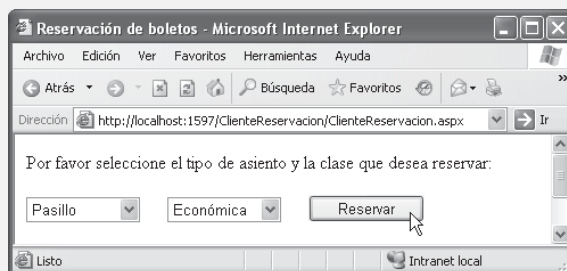


Figura 22.25 | Ejemplo de ejecución del formulario Web Forms para reservación de boletos. (Parte 2 de 2).

22.7 Tipos definidos por el usuario en los servicios Web

Los métodos Web que hemos demostrado hasta ahora reciben y devuelven valores de tipos simples. También es posible procesar tipos definidos por el usuario (conocidos como *tipos personalizados*) en un servicio Web. Estos tipos pueden pasarse a los métodos Web, o éstos pueden devolverlos. Los clientes de un servicio Web también pueden usar estos tipos definidos por el usuario, ya que la clase proxy creada para el cliente contiene las definiciones de los tipos.

Esta sección presenta un servicio Web llamado *GeneradorEcuaciones*, que genera al azar ecuaciones aritméticas de tipo *Ecuacion*. El cliente es una aplicación para enseñar matemáticas, que recibe como entrada la información acerca de la pregunta matemática que desea hacer el usuario (suma, resta o multiplicación) y el nivel de habilidad del mismo (1 especifica ecuaciones que utilizan números de un dígito, 2 especifica ecuaciones con números de dos dígitos y 3 especifica ecuaciones que contienen números de tres dígitos). Después, el servicio Web genera una ecuación que consiste en números aleatorios con el número apropiado de dígitos. La aplicación cliente recibe la *Ecuacion* y muestra la pregunta de ejemplo al usuario en un formulario Windows Forms.

Serialización de los tipos definidos por el usuario

Anteriormente mencionamos que SOAP debe soportar todos los tipos que se pasan desde/hacia los servicios Web. Entonces, ¿cómo es que SOAP puede soportar un tipo que ni siquiera se ha creado aún? Los tipos personalizados que se envían hacia o desde un servicio Web se serializan, con lo cual se pueden pasar en formato XML. A este proceso se le conoce como *serialización XML*.

Requerimientos para los tipos definidos por el usuario que se utilizan con métodos Web

Las clases que se utilizan para especificar tipos de retorno y tipos de parámetros para los métodos Web deben cumplir ciertos requerimientos:

1. Deben proporcionar un constructor `public` predeterminado, o uno sin parámetros. Cuando un servicio Web o el consumidor de un servicio Web reciben un objeto serializado en XML, el .NET Framework debe ser capaz de llamar a este constructor como parte del proceso de deserialización del objeto (es decir, convertirlo de vuelta en un objeto en C#).
2. Las propiedades y las variables de instancia que deben serializarse en formato XML deben declararse `public`. (Tenga en cuenta que las propiedades `public` pueden usarse para proporcionar acceso a las variables de instancia `private`.)
3. Las propiedades que se van a serializar deben proporcionar descriptores de acceso `get` y `set` (incluso aunque tengan cuerpos vacíos). Las propiedades de sólo lectura no se serializan.

Cualquier información que no esté serializada simplemente recibe su valor predeterminado (o el valor proporcionado por el constructor predeterminado o sin parámetros) cuando se deserializa un objeto de la clase.

**Error común de programación 22.3**

Si no se define un constructor `public` predeterminado o sin parámetros para un tipo que se va a pasar a un método Web, o un tipo que este método va a devolver, se produce un error en tiempo de ejecución.

**Error común de programación 22.4**

Si se define sólo el descriptor de acceso `get` o el descriptor de acceso `set` de una propiedad, para un tipo definido por el usuario que se va a pasar a un método Web, o un tipo que este método va a devolver, la propiedad será inaccesible para el cliente.

**Observación de ingeniería de software 22.1**

Los clientes de un servicio Web sólo pueden acceder a los miembros `public` de ese servicio. El programador puede proporcionar propiedades `public` para permitir el acceso a los datos `private`.

Definición de la clase Ecuacion

En la figura 22.26 definimos la clase `Ecuacion`. Las líneas 28-46 definen un constructor que recibe tres argumentos: dos valores `int` que representan a los operandos izquierdo y derecho, y un objeto `string` que representa la operación aritmética a realizar. El constructor establece las variables de instancia `operandoIzquierdo`, `operandoDerecho` y `tipoOperacion`, y después calcula el resultado apropiado. El constructor sin parámetros (líneas 21-25) llama al constructor de tres argumentos (líneas 28-46) y le pasa algunos valores predeterminados. No utilizamos el constructor sin parámetros en forma explícita, pero el mecanismo de serialización XML lo utiliza cuando se deserializan objetos de esta clase. Debido a que proporcionamos un constructor con parámetros, debemos definir explícitamente el constructor sin parámetros en esta clase, para que los objetos de la misma puedan pasarse a los métodos Web, o que estos métodos puedan devolverlos.

```

1  // Fig. 22.26: Ecuacion.cs
2  // Clase Ecuacion que contiene información acerca de una ecuación.
3  using System;
4  using System.Data;
5  using System.Configuration;
6  using System.Web;
7  using System.Web.Security;
8  using System.Web.UI;
9  using System.Web.UI.WebControls;
10 using System.Web.UI.WebControls.WebParts;

```

Figura 22.26 | Clase que almacena la información de la ecuación. (Parte I de 4).

```

11 using System.Web.UI.HtmlControls;
12
13 public class Ecuacion
14 {
15     private int operandoIzquierdo; // número a la izquierda del operador
16     private int operandoDerecho; // número a la derecha del operador
17     private int valorResultado; // resultado de la operación
18     private string tipoOperacion; // tipo de la operación
19
20     // constructor predeterminado requerido
21     public Ecuacion()
22         : this( 0, 0, "+" )
23     {
24         // cuerpo vacío
25     } // fin del constructor predeterminado
26
27     // constructor con tres argumentos para la clase Ecuacion
28     public Ecuacion( int valorIzq, int valorDer, string tipo )
29     {
30         operandoIzquierdo = valorIzq;
31         operandoDerecho = valorDer;
32         tipoOperacion = tipo;
33
34         switch ( tipoOperacion ) // realiza la operación apropiada
35         {
36             case "+": // suma
37                 valorResultado = operandoIzquierdo + operandoDerecho;
38                 break;
39             case "-": // resta
40                 valorResultado = operandoIzquierdo - operandoDerecho;
41                 break;
42             case "*": // multiplicación
43                 valorResultado = operandoIzquierdo * operandoDerecho;
44                 break;
45         } // fin de switch
46     } // fin del constructor con tres argumentos
47
48     // devuelve representación string del objeto Ecuacion
49     public override string ToString()
50     {
51         return operandoIzquierdo.ToString() + " " + tipoOperacion + " " +
52             operandoDerecho.ToString() + " = " + valorResultado.ToString();
53     } // fin del método ToString
54
55     // propiedad que devuelve un objeto string que representa el lado izquierdo
56     public string LadoIzquierdo
57     {
58         get
59         {
60             return operandoIzquierdo.ToString() + " " + tipoOperacion + " " +
61                 operandoDerecho.ToString();
62         } // fin de get
63
64         set // descriptor de acceso set requerido
65         {
66             // cuerpo vacío
67         } // fin de set
68     } // fin de la propiedad LadoIzquierdo
69

```

Figura 22.26 | Clase que almacena la información de la ecuación. (Parte 2 de 4).

```

70 // propiedad que devuelve un objeto string que representa el lado derecho
71 public string LadoDerecho
72 {
73     get
74     {
75         return valorResultado.ToString();
76     } // fin de get
77
78     set // descriptor de acceso set requerido
79     {
80         // cuerpo vacío
81     } // fin de set
82 } // fin de la propiedad LadoDerecho
83
84 // propiedad para acceder al operando izquierdo
85 public int Izquierdo
86 {
87     get
88     {
89         return operandoIzquierdo;
90     } // fin de get
91
92     set
93     {
94         operandoIzquierdo = value;
95     } // fin de set
96 } // fin de la propiedad Izquierdo
97
98 // propiedad para acceder al operando derecho
99 public int Derecho
100 {
101     get
102     {
103         return operandoDerecho;
104     } // fin de get
105
106     set
107     {
108         operandoDerecho = value;
109     } // fin de set
110 } // fin de la propiedad Derecho
111
112 // propiedad para acceder al resultado de aplicar
113 // una operación a los operandos izquierdo y derecho
114 public int Resultado
115 {
116     get
117     {
118         return valorResultado;
119     } // fin de get
120
121     set
122     {
123         valorResultado = value;
124     } // fin de set
125 } // fin de la propiedad Resultado
126
127 // propiedad para acceder a la operación
128 public string Operacion

```

Figura 22.26 | Clase que almacena la información de la ecuación. (Parte 3 de 4).


```

129     {
130         get
131         {
132             return tipoOperacion;
133         } // fin de get
134
135         set
136         {
137             tipoOperacion = value;
138         } // fin de set
139     } // fin de la propiedad Operacion
140 } // fin de la clase Ecuacion

```

Figura 22.26 | Clase que almacena la información de la ecuación. (Parte 4 de 4).

La clase *Ecuacion* define las propiedades *LadoIzquierdo* (líneas 56-68), *LadoDerecho* (líneas 71-82), *Izquierdo* (líneas 85-96), *Derecho* (líneas 99-110), *Resultado* (líneas 114-125) y *Operacion* (líneas 128-139). El cliente del servicio Web no necesita modificar los valores de las propiedades *LadoIzquierdo* y *LadoDerecho*. No obstante, recuerde que una propiedad puede serializarse sólo si tiene los descriptores de acceso *get* y *set*; esto es cierto aun cuando el descriptor de acceso *set* tiene un cuerpo vacío. *LadoIzquierdo* (líneas 56-68) devuelve un objeto *string* que representa todo lo que hay a la izquierda del signo igual (=) en la ecuación, y *LadoDerecho* (líneas 71-82) devuelve un objeto *string* que representa todo lo que hay a la derecha del signo igual (=). *Izquierdo* (líneas 85-96) devuelve el objeto *int* a la izquierda del operador (que se denomina operador izquierdo), y *Derecho* (líneas 99-110) devuelve el objeto *int* a la derecha del operador (que se denomina operando derecho). *Resultado* (líneas 114-125) devuelve la solución a la ecuación, y *Operacion* (líneas 128-139) devuelve el operador en la ecuación. El cliente en este caso de estudio no utiliza la propiedad *LadoDerecho*, pero la incluimos en caso de que otros clientes elijan usarla a futuro.

Creación del servicio Web *GeneradorEcuaciones*

La figura 22.27 presenta el servicio Web *GeneradorEcuaciones*, el cual crea al azar objetos *Ecuacion* personalizados. Este servicio Web sólo contiene el método *GenerarEcuacion* (líneas 16-32), el cual recibe dos parámetros: un objeto *string* que representa a la operación matemática (suma, resta o multiplicación) y un objeto *int* que representa el nivel de dificultad.

```

1  // Fig. 22.27: Generador.cs
2  // Servicio Web para generar ecuaciones al azar, con base en una
3  // operación y un nivel de dificultad especificados.
4  using System;
5  using System.Web;
6  using System.Web.Services;
7  using System.Web.Services.Protocols;
8
9  [ WebService( Namespace = "http://www.deitel.com/", Description =
10     "Servicio Web que genera una ecuación matemática." ) ]
11  [ WebServiceBinding( ConformsTo = WsiProfiles.BasicProfile1_1 ) ]
12  public class Generador : System.Web.Services.WebService
13  {
14      // Método para generar una ecuación matemática
15      [ WebMethod( Description = "Método para generar una ecuación matemática." ) ]
16      public Ecuacion GenerarEcuacion( string operacion, int nivel )
17      {
18          // encuentra los números máximo y mínimo a utilizar
19          int maximo = Convert.ToInt32(Math.Pow( 10, nivel));
20          int minimo = Convert.ToInt32(Math.Pow( 10, nivel - 1 ) );

```

Figura 22.27 | Servicio Web que genera ecuaciones al azar. (Parte 1 de 2).

```

21
22 // objeto para generar números aleatorios
23 Random objetoAleatorio = new Random();
24
25 // crea una ecuación que consiste en dos números
26 // aleatorios, entre los parámetros mínimo y máximo
27 Ecuacion ecuacion = new Ecuacion(
28     objetoAleatorio.Next( minimo, maximo ),
29     objetoAleatorio.Next( minimo, maximo ), operacion );
30
31 return ecuacion;
32 } // fin del método GenerarEcuacion
33 } // fin de la clase Generador

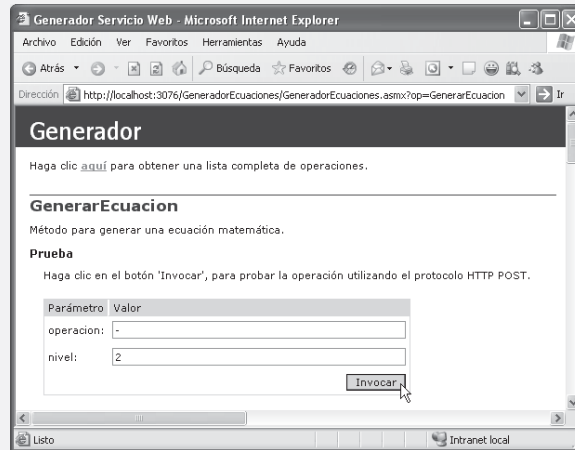
```

Figura 22.27 | Servicio Web que genera ecuaciones al azar. (Parte 2 de 2).

Prueba del servicio Web GeneradorEcuaciones

La figura 22.28 muestra el resultado de evaluar el servicio Web GeneradorEcuaciones. Observe que el valor de retorno de nuestro método Web está codificado en XML. No obstante, este ejemplo difiere de los anteriores, en cuanto a que el XML especifica los valores para todas las propiedades y datos públicos del objeto que se va a devolver. El objeto de retorno está serializado en XML. Nuestra clase proxy toma este valor de retorno y lo deserializa en un objeto de la clase Ecuacion, para después pasarlo al cliente.

a) Invocación del método **GenerarEcuacion** para crear una ecuación de resta con números de dos dígitos.



b) Resultados codificados en XML de la invocación del método **GenerarEcuacion**, para crear una ecuación de resta con números de dos dígitos.

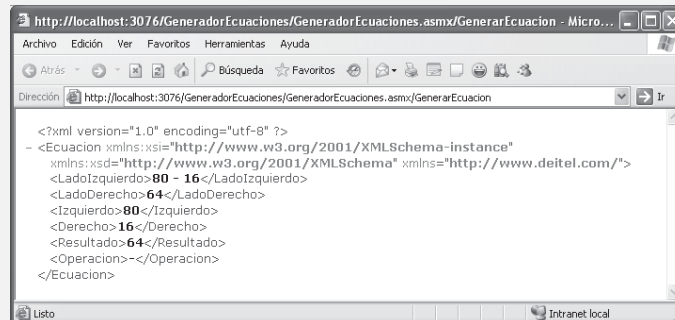


Figura 22.28 | Devolución de un objeto serializado XML de un método Web.

Observe que el objeto *Ecuacion* *no* se pasa entre el servicio Web y el cliente, sino que la información en el objeto se envía como datos codificados en XML. Los clientes creados en .NET recibirán la información y crearán un nuevo objeto *Ecuacion*. Sin embargo, los clientes creados en otras plataformas pueden utilizar la información de manera distinta. Los lectores que creen clientes en otras plataformas deben revisar la documentación de los servicios Web para la plataforma específica que estén usando, para ver cómo sus clientes pueden procesar los tipos personalizados.

Examinaremos el método Web *GenerarEcuacion* más de cerca. Las líneas 19-20 de la figura 22.27 definen los límites superior e inferior para los números aleatorios que utiliza el método para generar una *Ecuacion*. Para establecer estos límites, el programa llama primero al método *Static Pow* de la clase *Math*; este método eleva su primer argumento a la potencia de su segundo argumento. Para calcular el valor de *maximo* (el límite superior para cualquier número generado al azar, que se utilice para formar una *Ecuacion*), el programa eleva el número 10 a la potencia del argumento *nivel* especificado (línea 19). Si *nivel* es 1, *maximo* es 10; si *nivel* es 2, *maximo* es 100; y si *nivel* es 3, *maximo* es 1000. El valor de la variable *minimo* se determina elevando 10 a una potencia menor que *nivel* (línea 20). Esto calcula el número más pequeño con un número de dígitos determinado por *nivel*. Si *nivel* es 1, *minimo* es 1; si *nivel* es 2, *minimo* es 10; y si *nivel* es 3, *minimo* es 100.

Las líneas 27-29 crean un nuevo objeto *Ecuacion*. El programa llama al método *Next* de *Random*, el cual devuelve un *int* que es mayor o igual que el límite inferior especificado, pero menor que el límite superior especificado. Este método genera un valor del operando izquierdo que es mayor o igual a *minimo*, pero menor que *maximo* (es decir, un número con *nivel* dígitos). El operando derecho es otro número aleatorio con las mismas características. La línea 29 pasa el objeto *string* *operacion*, que recibe *GenerarEcuacion*, al constructor de *Ecuacion*. La línea 31 devuelve el nuevo objeto *Ecuacion* al cliente.

Consumo del servicio Web *GeneradorEcuaciones*

La aplicación *TutorMatematicas* (figura 22.29) utiliza el servicio Web *GeneradorEcuaciones*. La aplicación llama al método *GenerarEcuacion* del servicio Web para crear un objeto *Ecuacion*. Después, el tutor muestra el lado izquierdo de la *Ecuacion* y espera la entrada del usuario. Este ejemplo accede a las clases *Generador* y *Ecuacion* del espacio de nombres *localhost*; ambas se colocan en este espacio de nombres de manera predeterminada, cuando se genera el proxy. En las líneas 22-23 declaramos variables de estos tipos. La línea 23 también crea el proxy de *Generador*.

```

1  // Fig. 22.29: TutorMatematicas.cs
2  // Programa para enseñar matemáticas, que utiliza un servicio Web para generar
   ecuaciones al azar.
3  using System;
4  using System.Collections.Generic;
5  using System.ComponentModel;
6  using System.Data;
7  using System.Drawing;
8  using System.Text;
9  using System.Windows.Forms;
10
11 namespace TutorMatematicas
12 {
13     public partial class TutorMatematicasForm : Form
14     {
15         public TutorMatematicasForm()
16         {
17             InitializeComponent();
18         } // fin del constructor
19
20         private string operacion = "+";
21         private int nivel = 1;
22         localhost.Ecuacion ecuacion;
23         localhost.Generador generador = new localhost.Generador();

```

Figura 22.29 | Aplicación para enseñar matemáticas. (Parte I de 4).

```

24
25 // genera una nueva ecuación cuando el usuario hace clic en el botón
26 private void generarButton_Click( object sender, EventArgs e )
27 {
28     // genera ecuación usando la operación y el nivel actuales
29     ecuacion = generador.GenerarEcuacion( operacion, nivel );
30
31     // muestra el lado izquierdo de la ecuación
32     preguntaLabel.Text = ecuacion.LadoIzquierdo;
33
34     aceptarButton.Enabled = true;
35     respuestaTextBox.Enabled = true;
36 } // fin del método generarButton_Click
37
38 // verifica la respuesta del usuario
39 private void aceptarButton_Click( object sender, EventArgs e )
40 {
41     // determina el resultado correcto del objeto Ecuacion
42     int respuesta = ecuacion.Resultado;
43
44     if ( respuestaTextBox.Text == "" )
45         return;
46
47     // obtiene la respuesta del usuario
48     int respuestaUsuario = Int32.Parse( respuestaTextBox.Text );
49
50     // determina si la respuesta del usuario es correcta
51     if ( respuesta == respuestaUsuario )
52     {
53         preguntaLabel.Text = ""; // borra la pregunta
54         respuestaTextBox.Text = ""; // borra la respuesta
55         aceptarButton.Enabled = false; // deshabilita botón Aceptar
56         MessageBox.Show( "¡Correcto! ¡Buen trabajo!" );
57     } // fin de if
58     else
59         MessageBox.Show( "Incorrecto. Intente de nuevo." );
60 } // fin del método aceptarButton_Click
61
62 // establece el nivel de dificultad a 1
63 private void nivelUnoRadioButton_CheckedChanged( object sender,
64     EventArgs e )
65 {
66     nivel = 1;
67 } // fin del método nivelUnoRadioButton_CheckedChanged
68
69 // establece el nivel de dificultad a 2
70 private void nivelDosRadioButton_CheckedChanged( object sender,
71     EventArgs e )
72 {
73     nivel = 2;
74 } // fin del método nivelDosRadioButton_CheckedChanged
75
76 // establece el nivel de dificultad a 3
77 private void nivelTresRadioButton_CheckedChanged( object sender,
78     EventArgs e )
79 {
80     nivel = 3;
81 } // fin del método nivelTresRadioButton_CheckedChanged
82

```

Figura 22.29 | Aplicación para enseñar matemáticas. (Parte 2 de 4).

```

83 // establece la operación a suma
84 private void sumaRadioButton_CheckedChanged( object sender,
85     EventArgs e )
86 {
87     operacion = "+";
88     generarButton.Text =
89         "Generar ejemplo de " + sumaRadioButton.Text;
90 } // fin del método sumaRadioButton_CheckedChanged
91
92 // establece la operación a resta
93 private void restaRadioButton_CheckedChanged( object sender,
94     EventArgs e )
95 {
96     operacion = "-";
97     generarButton.Text = "Generar ejemplo de " +
98         restaRadioButton.Text;
99 } // fin del método restaRadioButton_CheckedChanged
100
101 // establece la operación a multiplicación
102 private void multiplicacionRadioButton_CheckedChanged(
103     object sender, EventArgs e )
104 {
105     operacion = "*";
106     generarButton.Text = "Generar ejemplo de " +
107         multiplicacionRadioButton.Text;
108 } // fin del método multiplicacionRadioButton_CheckedChanged
109 } // fin de la clase TutorMatematicasForm
110 } // fin del espacio de nombres TutorMatematicas

```

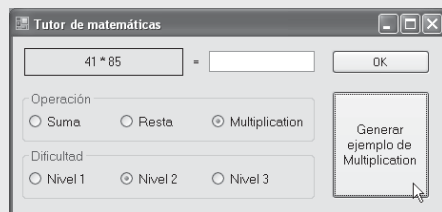
a) Generación de una ecuación de suma, nivel 1

b) Respuesta incorrecta a la ecuación.

c) Respuesta correcta a la ecuación.

Figura 22.29 | Aplicación para enseñar matemáticas. (Parte 3 de 4).

d) Generación de una ecuación de multiplicación, nivel 2.

**Figura 22.29** | Aplicación para enseñar matemáticas. (Parte 4 de 4).

La aplicación tutor de matemáticas muestra una ecuación y espera a que el usuario escriba una respuesta. La opción predeterminada para el nivel de dificultad es **1**, pero el usuario puede modificarla eligiendo un nivel de los controles `RadioButton` que están dentro del control `GroupBox` con la leyenda **Dificultad**. Al hacer clic en cualquiera de los niveles se invoca el manejador de eventos `CheckedChanged` del correspondiente control `RadioButton` (líneas 63-81), el cual establece el entero `nivel` al nivel seleccionado por el usuario. Aunque la opción predeterminada para el tipo de pregunta es **Suma**, el usuario también puede modificar esto si selecciona uno de los controles `RadioButton` dentro del control `GroupBox` con la leyenda **Operación**. Al hacer esto, invoca a los correspondientes manejadores de eventos en las líneas 84-108, que asignan al objeto `string` llamado `operacion` el símbolo correspondiente a la selección del usuario. Cada manejador de eventos también actualiza la propiedad `Texto` del botón **Generar** para que coincida con la operación recién seleccionada.

El manejador de eventos `generarButton_Click` (líneas 26-36) invoca al método `GenerarEcuacion` de `GeneradorEcuaciones` (línea 29). Después de recibir un objeto `Ecuación` del servicio Web, el manejador muestra el lado izquierdo de la ecuación en `preguntaLabel` (línea 32) y habilita `acceptarButton`, de manera que el usuario pueda escribir una respuesta. Cuando el usuario hace clic en **Aceptar**, `acceptarButton_Click` (líneas 39-60) verifica que el usuario haya proporcionado la respuesta correcta.

22.8 Conclusión

En este capítulo presentamos los servicios Web ASP.NET: una tecnología que permite a los usuarios solicitar y recibir datos a través de Internet, y que promueve la reutilización de software en sistemas distribuidos. En este capítulo aprendió que un servicio Web es una clase que permite a los equipos cliente llamar a los métodos del servicio Web en forma remota, a través de formatos de datos y protocolos comunes, como XML, HTTP y SOAP. Hablamos sobre varios beneficios de este tipo de computación distribuida; por ejemplo, los clientes pueden acceder a ciertos datos en equipos remotos, y los clientes que carecen del poder de procesamiento necesario para realizar cálculos específicos pueden aprovechar los recursos de los equipos remotos.

Explicamos cómo Visual C# 2005, Visual Web Developer 2005 y el .NET Framework facilitan la creación y el consumo de los servicios Web. Aprendió a definir los servicios Web y los métodos Web, así como a consumirlos desde aplicaciones Windows y aplicaciones Web ASP.NET. Después de explicar la mecánica de los servicios Web a través de nuestro ejemplo `EnteroEnorme`, demostramos servicios Web más sofisticados, que utilizan rastreo de sesiones y tipos definidos por el usuario.

En el siguiente capítulo, hablaremos sobre los detalles de bajo nivel relacionados con las redes computacionales. Le mostraremos cómo implementar servidores y clientes que se comunican entre sí, cómo enviar y recibir datos mediante sockets (que simplifican dichas transmisiones de tal forma que es como si se realizaran operaciones de lectura/escritura con archivos, respectivamente) y cómo crear un servidor con subprocesamiento múltiple para jugar una versión en red del popular juego Tres en raya.

22.9 Recursos Web

Además de los recursos Web que se muestran aquí, es conveniente que consulte los recursos Web relacionados con ASP.NET, que se proporcionan al final del capítulo 21.

msdn.microsoft.com/webservices

El Centro para desarrolladores de servicios Web de Microsoft incluye las especificaciones y hojas técnicas sobre la tecnología de los servicios Web .NET, así como artículos, columnas y vínculos sobre XML/SOAP.

www.webservices.org

Este sitio proporciona noticias, artículos, recursos y vínculos relacionados con la industria, acerca de los servicios Web.

www-130.ibm.com/developerworks/webservices

El sitio de IBM para la arquitectura orientada a servicios (SOA) y los servicios Web incluye artículos, descargas, demos y grupos de discusión, en relación con la tecnología de los servicios Web.

www.w3.org/TR/wsdl

Este sitio proporciona una gran cantidad de información acerca de WSDL, incluyendo una discusión detallada sobre las tecnologías relacionadas con los servicios Web, como XML, SOAP, HTTP y los tipos MIME, dentro del contexto de WSDL.

www.w3.org/TR/soap

Este sitio proporciona una gran cantidad de información sobre los mensajes SOAP, el uso de SOAP con HTTP y cuestiones de seguridad relacionadas con SOAP.

www.uddi.com

El sitio de Integración, descubrimiento y descripción universal proporciona discusiones, especificaciones, hojas técnicas e información general acerca de UDDI.

www.ws-i.org

El sitio Web de la Organización de interoperabilidad de servicios Web proporciona integración detallada en relación con los esfuerzos para generar servicios Web basados en estándares que promuevan la interoperabilidad y una verdadera independencia de plataformas.

webservices.xml.com/security

Este sitio contiene artículos acerca de las cuestiones de seguridad relacionadas con los servicios Web, y los protocolos estándar de seguridad.

23

Redes: sockets basados en flujos y datagramas

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Implementar aplicaciones de red que utilicen sockets y datagramas.
- Implementar clientes y servidores que se comuniquen entre sí.
- Implementar aplicaciones de colaboración basada en red.
- Construir un servidor con subprocesamiento múltiple.
- Utilizar el control WebBrowser para agregar herramientas de exploración Web a cualquier aplicación.
- Utilizar la tecnología .NET Remoting para permitir que una aplicación que se ejecuta en una computadora invoque a los métodos de una aplicación que se ejecuta en otra computadora.

Si la presencia de electricidad puede hacerse visible en cualquier parte de un circuito, no veo por qué razón la inteligencia no pueda transmitirse en forma instantánea mediante la electricidad.

—Samuel F. B. Moore

El protocolo es todo.

—Francois Giuliani

Las redes de telecomunicaciones, información y computación son hoy lo que las redes de ferrocarriles, autopistas y canales fueron en otra época.

—Bruno Kreisky

*El puerto está cerca,
escucho las campanas,
toda la gente se regocija.*

—Walt Whitman

- 23.1 Introducción
- 23.2 Comparación entre la comunicación orientada a la conexión y la comunicación sin conexión
- 23.3 Protocolos para transportar datos
- 23.4 Establecimiento de un servidor TCP simple (mediante el uso de sockets de flujo)
- 23.5 Establecimiento de un cliente TCP simple (mediante el uso de sockets de flujo)
- 23.6 Interacción entre cliente/servidor mediante conexiones de sockets de flujo
- 23.7 Interacción entre cliente/servidor sin conexión mediante datagramas
- 23.8 Juego de Tres en raya cliente/servidor mediante el uso de un servidor con subprocesamiento múltiple
- 23.9 Control WebBrowser
- 23.10 .NET Remoting
- 23.11 Conclusión

23.1 Introducción

Hay mucho entusiasmo acerca de Internet y Web, ya que Internet enlaza toda la información a nivel mundial y la Web facilita el uso de Internet y le proporciona el toque atractivo del contenido multimedia. Las organizaciones ven a Internet y a la Web como algo crucial para las estrategias de sus sistemas de información. La FCL de .NET cuenta con una variedad de herramientas de red integradas, que facilitan el desarrollo de aplicaciones basadas en Internet y Web. Los programas pueden buscar en todo el mundo información y colaborar con programas que se ejecutan en otras computadoras a nivel internacional, nacional o sólo dentro de una organización.

En el capítulo 21, ASP.NET 2.0, formularios Web Forms y controles Web, y en el capítulo 22, Servicios Web, empezamos nuestra presentación de las herramientas de redes y computación distribuida en C#. Hablamos sobre ASP.NET, los formularios Web Forms y los servicios Web: tecnologías de red de alto nivel que permiten a los programadores desarrollar aplicaciones distribuidas. En este capítulo nos enfocamos en las tecnologías de red subyacentes que dan soporte a las herramientas de C#, ASP.NET y los servicios Web.

Este capítulo comienza con las generalidades de las técnicas y tecnologías de computación que se utilizan para transmitir datos a través de Internet. A continuación presentaremos los conceptos básicos acerca de cómo establecer una conexión entre dos aplicaciones que utilizan flujos de datos, similares a la E/S de archivos. Este enfoque orientado a las conexiones permite que los programas se comuniquen entre sí con la misma facilidad que se realizan las operaciones de escritura y lectura en los archivos en disco. Después presentaremos una aplicación simple de chat, en la que se utilizan estas técnicas para enviar mensajes entre un cliente y un servidor. El capítulo continuará con una presentación y un ejemplo de las técnicas sin conexión para transmitir datos entre aplicaciones, lo cual es menos confiable que establecer una conexión entre aplicaciones, pero mucho más eficiente. Dichas técnicas se utilizan por lo general en aplicaciones como audio y video en tiempo real a través de Internet. A continuación presentaremos un ejemplo de un juego de Tres en raya tipo cliente/servidor, el cual demuestra cómo crear un servidor simple con subprocesamiento múltiple. Después mostraremos el nuevo control WebBrowser para agregar herramientas de exploración Web a cualquier aplicación. El capítulo termina con una breve introducción a la tecnología .NET Remoting que, al igual que los servicios Web (capítulo 22), permite la computación distribuida a través de redes.

23.2 Comparación entre la comunicación orientada a la conexión y la comunicación sin conexión

Hay dos enfoques principales para las comunicaciones entre aplicaciones: *orientadas a la conexión* y *sin conexión*. Las comunicaciones orientadas a la conexión son similares al sistema telefónico, en el cual se establece una conexión y se mantiene el tiempo que dure la sesión. Los servicios sin conexión son similares al servicio postal, en el cual dos cartas enviadas desde el mismo lugar y hacia el mismo destino pueden tomar dos rutas drásticamente distintas a través del sistema, e incluso llegar a horas distintas, o no llegar.

En un enfoque orientado a la conexión, las computadoras se envían una a otra información de control (a través de una técnica conocida como *intercambio*, o *handshaking*) para iniciar una conexión de un extremo al otro.

Internet es una *red no confiable*, lo cual significa que los datos que se envían a través de esta red pueden dañarse o perderse. Los datos se envían en *paquetes*, que contienen piezas de los datos, junto con información que ayuda a Internet a enrutar los paquetes hacia el destino apropiado. Internet no garantiza nada acerca de los paquetes enviados; podrían llegar corruptos o desordenados, duplicados o incluso podrían no llegar. Internet sólo hace su “mejor esfuerzo” por entregar los paquetes. Un enfoque orientado a la conexión asegura que las comunicaciones sean confiables en redes no confiables, garantizando de esta forma que los paquetes enviados llegarán al receptor destinado sin daños, y se ensamblarán en la secuencia correcta.

En un enfoque sin conexión, las dos computadoras no realizan el intercambio antes de la transmisión, por lo que no se garantiza la confiabilidad; los datos enviados tal vez nunca lleguen al receptor destinado. Sin embargo, un enfoque sin conexión evita la sobrecarga asociada con el intercambio y con el aseguramiento de la confiabilidad; a menudo se requiere pasar menos información entre un host y otro.

23.3 Protocolos para transportar datos

Existen muchos protocolos para la comunicación entre aplicaciones. Los *protocolos* son conjuntos de reglas que gobiernan la forma en que deben interactuar dos entidades. En este capítulo nos enfocaremos en el Protocolo de control de transmisión (TCP) y en el Protocolo de datagramas de usuario (UDP). Las herramientas de red TCP y UDP de .NET se definen en el espacio de nombres **System.Net.Sockets**.

El *Protocolo de control de transmisión (TCP)* es un protocolo de comunicación orientado a la conexión, el cual garantiza que los paquetes enviados llegarán al receptor destinado sin daños y en la secuencia correcta. TCP permite que protocolos como HTTP (capítulo 21) envíen información a través de una red, de una manera tan simple y confiable como escribir a un archivo en una computadora local. Si los paquetes de información no llegan al recipiente, TCP se asegura que los paquetes se envíen de nuevo. Si los paquetes llegan fuera de orden, TCP los vuelve a ensamblar en el orden correcto, en forma transparente para la aplicación receptora. Si llegan paquetes duplicados, TCP los descarta.

Las aplicaciones que no requieren de la garantía de transmisión confiable de un extremo a otro que ofrece TCP, por lo general utilizan el *Protocolo de datagramas de usuario (UDP)* sin conexión. UDP incurre en la mínima sobrecarga necesaria para la comunicación entre aplicaciones; no garantiza que los paquetes, llamados *datagramas*, lleguen a su destino o que lleguen en el orden original.

Existen ciertas ventajas en cuanto al uso de UDP en comparación con TCP. UDP tiene poca sobrecarga, ya que los datagramas no necesitan llevar la información que llevan los paquetes TCP para asegurar la confiabilidad. Además, reduce el tráfico de red relativo a TCP, debido a la ausencia de intercambio, retransmisiones, etcétera.

La comunicación no confiable es aceptable en muchas situaciones. En primer lugar, la confiabilidad no es necesaria para algunas aplicaciones, por lo que puede evitarse la sobrecarga impuesta por un protocolo que garantice la confiabilidad. En segundo lugar, algunas aplicaciones como el audio y video en tiempo real pueden tolerar la pérdida ocasional de datagramas. Esto por lo general produce una pequeña pausa (o “hipo”) en el audio o video que se está reproduciendo. Si la misma aplicación se ejecutara sobre TCP, un segmento perdido podría producir una pausa considerable, ya que el protocolo esperaría a que se retransmitiera el segmento perdido y se entregara en forma correcta antes de continuar. Por último, las aplicaciones que necesitan implementar sus propios mecanismos de confiabilidad distintos a los de TCP pueden crear tales mecanismos sobre UDP.

23.4 Establecimiento de un servidor TCP simple (mediante el uso de sockets de flujo)

Por lo general, con TCP un servidor “espera” una solicitud de conexión de un cliente. A menudo, el programa servidor contiene una instrucción de control o un bloque de código que se ejecuta en forma continua, hasta que el servidor recibe una solicitud. Al recibir esta solicitud, el servidor establece una conexión con el cliente. Después, utiliza esta conexión (administrada por un objeto de la clase **Socket**) para manejar las próximas solicitudes de ese cliente y enviarle datos. Como los programas que se comunican a través de TCP procesan los datos que envían y reciben como flujos de bytes, en ocasiones los programadores se refieren a los objetos Socket como “Sockets de flujo”.

Para establecer un servidor simple con TCP y sockets de flujo, se requieren cinco pasos. Primero, es necesario crear un objeto de la clase **TcpListener** del espacio de nombres **System.Net.Sockets**. Esta clase representa a

un socket de flujo TCP, a través del cual un servidor puede escuchar en espera de solicitudes. Al crear un nuevo objeto `TcpListener`, como en

```
TcpListener servidor = new TcpListener( direcciónIP, puerto );
```

se **enlaza** (asigna) la aplicación servidor al **número de puerto** especificado. Un número de puerto es un identificador numérico que utiliza una aplicación para identificarse a sí misma, en una **dirección de red** dada, también conocida como **Dirección de protocolo Internet (Dirección IP)**. Las direcciones IP identifican a las computadoras en Internet. De hecho, los nombres de los sitios Web como `www.deitel.com` son alias para las direcciones IP. Una dirección IP se representa mediante un objeto de la clase **IPAddress**, del espacio de nombres **System.Net**. Cualquier aplicación que utiliza redes se identifica a sí misma a través de un *par dirección IP/número de puerto*; dos aplicaciones no pueden tener el mismo número de puerto, en una dirección IP dada. Por lo general, no es necesario enlazar en forma explícita un socket a un puerto de conexión (usando el método `Bind` de la clase `Socket`), ya que la clase `TcpListener` y otras clases que veremos en este capítulo lo hacen en forma automática, junto con otras operaciones de inicialización de sockets.



Observación de ingeniería de software 23.1

Los números de puerto pueden tener valores entre 0 y 65535. Muchos sistemas operativos reservan los números de puerto menores a 1024 para los servicios del sistema (como los servidores Web y de correo electrónico). Las aplicaciones deben recibir privilegios especiales para poder usar estos números de puerto reservados.

Para recibir solicitudes, `TcpListener` primero debe escuchar en espera de ellas. El segundo paso en el proceso de conexión es llamar al método `Start` de `TcpListener`, el cual hace que el objeto `TcpListener` empiece a escuchar en espera de solicitudes de conexión. El servidor escucha en forma indefinida en espera de una solicitud; es decir, la ejecución de la aplicación del lado servidor espera hasta que algún cliente trata de conectarse con ella. El servidor crea una conexión con el cliente cuando recibe una solicitud de conexión. Un objeto de la clase `Socket` (espacio de nombres `System.Net.Sockets`) administra una conexión con un cliente. El método `AcceptSocket` de la clase `TcpListener` acepta una solicitud de conexión. Este método devuelve un objeto `Socket` al momento de la conexión, como en la siguiente instrucción:

```
Socket conexion = servidor.AcceptSocket();
```

Cuando el servidor recibe una solicitud, `AcceptSocket` llama al método `Accept` del `Socket` subyacente de `TcpListener` para realizar la conexión. Éste es un ejemplo de cómo se oculta al programador la complejidad del trabajo en red. Usted simplemente coloca la instrucción anterior en un programa del lado servidor; las clases del espacio de nombres `System.Net.Sockets` se encargan de los detalles relacionados con la aceptación de solicitudes y el establecimiento de conexiones.

El tercer paso establece los flujos que se utilizan para la comunicación con el cliente. En este paso, creamos un objeto `NetworkStream` que utiliza el objeto `Socket` (que representa la conexión) para realizar el envío y la recepción de los datos. En nuestro siguiente ejemplo, utilizamos este objeto `NetworkStream` para crear un `BinaryWriter` y un `BinaryReader`, que se utilizarán para enviar información hacia, y recibir información desde el cliente, respectivamente.

El paso cuatro es la fase de procesamiento, en la cual el servidor y el cliente se comunican mediante la conexión establecida en el tercer paso. En esta fase, el cliente utiliza el método `Write` de `BinaryWriter` y el método `ReadString` de `BinaryReader` para realizar las comunicaciones apropiadas.

El quinto paso es la fase de terminación de la conexión. Cuando el cliente y el servidor terminan de comunicarse, el servidor llama al método `Close` de `BinaryReader`, `BinaryWriter`, `NetworkStream` y `Socket` para terminar la conexión. Entonces, el servidor puede regresar al paso dos para esperar la siguiente solicitud de conexión. La documentación para la clase `Socket` recomienda que se llame al método `Shutdown` antes del método `Close`, para asegurar que todos los datos se envíen y se reciban antes de que se cierre el `Socket`.

Un problema asociado con el esquema de servidor descrito en esta sección es que el paso cuatro **bloquea** las demás solicitudes mientras procesa la solicitud del cliente conectado, por lo que ningún otro cliente puede conectarse con el servidor mientras se esté ejecutando el código que define a la fase de procesamiento. La técnica más común para tratar este problema es utilizar servidores con subprocesamiento múltiple, los cuales colocan el

código de la fase de procesamiento en un subproceso separado. Para cada solicitud de conexión que recibe el servidor, crea un objeto `Thread` para procesar la conexión, dejando a su objeto `TcpListener` (o `Socket`) libre para recibir otras conexiones. En la sección 23.8 demostramos un servidor con subprocesamiento múltiple.



Observación de ingeniería de software 23.2

Los servidores con subprocesamiento múltiple pueden administrar con eficiencia conexiones simultáneas con varios clientes. Esta arquitectura es precisamente la que utilizan los populares servidores UNIX y Windows para trabajo en red.



Observación de ingeniería de software 23.3

Puede implementarse un servidor con subprocesamiento múltiple para crear un subproceso que administre la E/S de red a través de un objeto `Socket` devuelto por el método `AcceptSocket`. Un servidor con subprocesamiento múltiple también puede implementarse para mantener una reserva de subprocesos que administren la E/S de red a través de objetos `Socket` recién creados.



Tip de rendimiento 23.1

En los sistemas de alto rendimiento con mucha memoria, puede implementarse un servidor con subprocesamiento múltiple para crear una reserva de subprocesos. Estos subprocesos pueden asignarse rápidamente para manejar la E/S de red a través de varios objetos `Socket`. Por ende, cuando se recibe una conexión, el servidor no incurre en la sobrecarga de la creación de subprocesos.

23.5 Establecimiento de un cliente TCP simple (mediante el uso de sockets de flujo)

Hay cuatro pasos para crear un cliente TCP simple. Primero es necesario crear un objeto de la clase `TcpClient` (espacio de nombres `System.Net.Sockets`) para conectarse al servidor. La conexión se establece llamando al método `Connect` de `TcpClient`. Una versión sobrecargada de este método recibe dos argumentos: la dirección IP del servidor y su número de puerto, como en:

```
TcpClient cliente = new TcpClient();
cliente.Connect( direcciónServidor, puertoServidor );
```

El `puertoServidor` es un objeto `int` que representa el número de puerto al que está enlazada la aplicación servidor para escuchar en espera de las solicitudes de conexión. La `direcciónServidor` puede ser una instancia de `IPAddress` (que encapsula la dirección IP del servidor) o un objeto `string` que especifique el nombre de host del servidor o su dirección IP. El método `Connect` también tiene una versión sobrecargada, a la cual se le puede pasar un objeto `EndPoint` que representa un par direcciónIP/número de puerto. El método `Connect` de `TcpClient` llama al método `Connect` de `Socket` para establecer la conexión. Si hubo éxito, el método `Connect` de `TcpClient` devuelve un entero positivo; en caso contrario, devuelve 0.

En el paso dos, el objeto `TcpClient` usa su método `GetStream` para obtener un objeto `NetworkStream`, de manera que pueda realizar operaciones de lectura y escritura con el servidor. Después usamos el objeto `NetworkStream` para crear un objeto `BinaryWriter` y un objeto `BinaryReader`, los cuales se utilizarán para enviar información hacia, y recibir información desde el servidor, respectivamente.

El tercer paso es la fase de procesamiento, en la que se comunican el cliente y el servidor. En esta fase de nuestro ejemplo, el cliente utiliza el método `Write` de `BinaryWriter` y el método `ReadString` de `BinaryReader` para realizar las comunicaciones apropiadas. Mediante el uso de un proceso similar al que utilizan los servidores, un cliente puede emplear subprocesos para evitar el bloqueo de la comunicación con otros servidores, mientras se procesan los datos de una conexión.

Una vez que se completa la transmisión, el paso cuatro requiere que el cliente cierre la conexión, llamando al método `Close` de `BinaryReader`, `BinaryWriter`, `NetworkStream` y `TcpClient`. Esto cierra cada uno de los flujos y el objeto `Socket` de `TcpClient` para terminar la conexión con el servidor. En este punto, puede establecerse una nueva conexión a través del método `Connect`, como describimos antes.

23.6 Interacción entre cliente/servidor mediante conexiones de sockets de flujo

Las aplicaciones de las figuras 23.1 y 23.2 utilizan las clases y las técnicas descritas en las dos secciones anteriores para crear una *aplicación de chat cliente/servidor* simple. El servidor espera una solicitud del cliente para hacer una conexión. Cuando una aplicación cliente se conecta al servidor, la aplicación servidor envía, al cliente, un arreglo de bytes, indicando que la conexión tuvo éxito. Después, el cliente muestra un mensaje, notificando al usuario que se estableció una conexión.

Tanto la aplicación cliente como la aplicación servidor contienen controles TextBox que permiten a los usuarios escribir mensajes y enviarlos a la otra aplicación. Cuando el cliente o el servidor envían el mensaje “TERMINAR”, termina la conexión entre el cliente y el servidor. Después, el servidor espera a que otro cliente solicite una conexión. Las figuras 23.1 y 23.2 proporcionan el código para las clases Servidor y Cliente, respectivamente. La figura 23.2 también contiene capturas de pantalla que muestran la ejecución entre el cliente y el servidor.

Clase ServidorChatForm

Empezaremos por ver la clase ServidorChatForm (figura 23.1). En el constructor, la línea 27 crea un objeto Thread que acepta conexiones de los clientes. El objeto delegado ThreadStart que se pasa como argumento para el constructor especifica el método que el objeto Thread debe ejecutar. La línea 28 empieza el objeto Thread, el cual utiliza al delegado ThreadStart para invocar al método EjecutarServidor (líneas 104-179). Este método inicializa el servidor para que reciba solicitudes de conexión y las procese. La línea 115 crea una instancia de un objeto TcpListener, para que escuche en espera de una solicitud de conexión de un cliente en el puerto 50000 (*paso 1*). La línea 118 llama después al método Start de TcpListener, lo cual hace que el objeto TcpListener empiece a esperar solicitudes (*paso 2*).

```

1  // Fig. 23.1: ServidorChat.cs
2  // Establece un servidor que recibe una conexión de un cliente, le envía
3  // una cadena, chatea con él y cierra la conexión.
4  using System;
5  using System.Windows.Forms;
6  using System.Threading;
7  using System.Net;
8  using System.Net.Sockets;
9  using System.IO;
10
11 public partial class ServidorChatForm : Form
12 {
13     public ServidorChatForm()
14     {
15         InitializeComponent();
16     } // fin del constructor
17
18     private Socket conexion; // Socket para aceptar una conexión
19     private Thread lecturaThread; // Thread para procesar los mensajes entrantes
20     private NetworkStream socketStream; // flujo de datos de red
21     private BinaryWriter escritor; // facilita la escritura en el flujo
22     private BinaryReader lector; // facilita la lectura del flujo
23
24     // inicializa el subproceso para la lectura
25     private void ServidorChatForm_Load( object sender, EventArgs e )
26     {
27         lecturaThread = new Thread( new ThreadStart( EjecutarServidor ) );
28         lecturaThread.Start();
29     } // fin del método ServidorChatForm_Load

```

Figura 23.1 | Parte correspondiente al servidor de una conexión cliente/servidor con sockets de flujo. (Parte 1 de 4).

```

30
31 // cierra todos los subprocesos asociados con esta aplicación
32 private void ServidorChatForm_FormClosing( object sender,
33     FormClosingEventArgs e )
34 {
35     System.Environment.Exit( System.Environment.ExitCode );
36 } // fin del método CharServerForm_FormClosing
37
38 // delegado que permite que se haga una llamada al método MostrarMensaje
39 // en el subproceso que crea y mantiene la GUI
40 private delegate void DisplayDelegate( string mensaje );
41
42 // el método DisplayDelegate establece la propiedad Text de mostrarTextBox
43 // en forma segura para los subprocesos
44 private void MostrarMensaje( string mensaje )
45 {
46     // si la modificación de mostrarTextBox no es segura para el subproceso
47     if ( mostrarTextBox.InvokeRequired )
48     {
49         // usa el método Invoke heredado para ejecutar MostrarMensaje
50         // a través de un delegado
51         Invoke( new DisplayDelegate( MostrarMensaje ),
52             new object[] { mensaje } );
53     } // fin de if
54     else // Se puede modificar mostrarTextBox en el subproceso actual
55         mostrarTextBox.Text += mensaje;
56 } // fin del método MostrarMensaje
57
58 // delegado que permite llamar al método DeshabilitarSalida
59 // en el subprofeso que crea y mantiene la GUI
60 private delegate void DisableInputDelegate( bool value );
61
62 // el método DeshabilitarSalida establece la propiedad ReadOnly de entradaTextBox
63 // de una manera segura para los subprocesos
64 private void DeshabilitarEntrada( bool valor )
65 {
66     // si la modificación de entrarTextBox no es segura para el subproceso
67     if ( entradaTextBox.InvokeRequired )
68     {
69         // usa el método heredado Invoke para ejecutar DeshabilitarEntrada
70         // a través de un delegado
71         Invoke( new DisableInputDelegate( DeshabilitarEntrada ),
72             new object[] { valor } );
73     } // fin de if
74     else // Se puede modificar entradaTextBox en el subproceso actual
75         entradaTextBox.ReadOnly = valor;
76 } // fin del método DeshabilitarEntrada
77
78 // envía al cliente el texto escrito en el servidor
79 private void entradaTextBox_KeyDown( object sender, KeyEventArgs e )
80 {
81     // envía el texto al cliente
82     try
83     {
84         if ( e.KeyCode == Keys.Enter && entradaTextBox.ReadOnly == false )
85         {
86             escritor.Write( "SERVIDOR>>> " + entradaTextBox.Text );
87             mostrarTextBox.Text += "\r\nSERVIDOR>>> " + entradaTextBox.Text;
88         }
89     }
90 }

```

Figura 23.1 | Parte correspondiente al servidor de una conexión cliente/servidor con sockets de flujo. (Parte 2 de 4).


```

89         // si el usuario en el servidor indicó la terminación
90         // de la conexión con el cliente
91         if ( entradaTextBox.Text == "TERMINAR" )
92             conexion.Close();
93
94         entradaTextBox.Clear(); // borra la entrada del usuario
95     } // fin de if
96 } // fin de try
97 catch ( SocketException )
98 {
99     mostrarTextBox.Text += "\nError al escribir objeto";
100 } // fin de catch
101 } // fin del método entradaTextBox_KeyDown
102
103 // permite que un cliente se conecte; muestra el texto que envía el cliente
104 public void EjecutarServidor()
105 {
106     TcpListener oyente;
107     int contador = 1;
108
109     // espera la conexión de un cliente y muestra el texto
110     // que envía el cliente
111     try
112     {
113         // Paso 1: crea TcpListener
114         IPAddress local = IPAddress.Parse( "127.0.0.1" );
115         oyente = new TcpListener( local, 50000 );
116
117         // Paso 2: TcpListener espera la solicitud de conexión
118         oyente.Start();
119
120         // Paso 3: establece la conexión con base en la solicitud del cliente
121         while ( true )
122         {
123             MostrarMensaje( "Esperando una conexión\r\n" );
124
125             // acepta una conexión entrante
126             conexion = oyente.AcceptSocket();
127
128             // crea objeto NetworkStream asociado con el socket
129             socketStream = new NetworkStream( conexion );
130
131             // crea objetos para transferir datos a través de un flujo
132             escritor = new BinaryWriter( socketStream );
133             lector = new BinaryReader( socketStream );
134
135             MostrarMensaje( "Conexión " + contador + " recibida.\r\n" );
136
137             // informa al cliente que la conexión fue exitosa
138             escritor.Write( "SERVIDOR>>> Conexión exitosa" );
139
140             DeshabilitarEntrada( false ); // habilita entradaTextBox
141
142             string laRespuesta = "";
143
144             // Paso 4: lee los datos de cadena que envía el cliente
145             do
146             {
147                 try

```

Figura 23.1 | Parte correspondiente al servidor de una conexión cliente/servidor con sockets de flujo. (Parte 3 de 4).

```

148         {
149             // lee la cadena que se envía al cliente
150             laRespuesta = lector.ReadString();
151
152             // muestra el mensaje
153             MostrarMensaje( "\r\n" + laRespuesta );
154         } // fin de try
155         catch ( Exception )
156         {
157             // maneja la excepción si hay error al leer los datos
158             break;
159         } // fin de catch
160     } while ( laRespuesta != "CLIENTE>>> TERMINAR" &&
161             conexion.Connected );
162
163     MostrarMensaje( "\r\nEl usuario terminó la conexión\r\n" );
164
165     // Paso 5: cierra la conexión
166     escritor.Close();
167     lector.Close();
168     socketStream.Close();
169     conexion.Close();
170
171     DeshabilitarEntrada( true ); // deshabilita entradaTextBox
172     contador++;
173 } // fin de while
174 } // fin de try
175 catch ( Exception error )
176 {
177     MessageBox.Show( error.ToString() );
178 } // fin de catch
179 } // fin del método EjecutarServidor
180 } // fin de la clase ServidorChatForm

```

Figura 23.1 | Parte correspondiente al servidor de una conexión cliente/servidor con sockets de flujo. (Parte 4 de 4).

Aceptar la conexión y establecer los flujos

Las líneas 121-173 declaran un ciclo infinito que empieza estableciendo la conexión solicitada por el cliente (*paso 3*). La línea 126 llama al método `AcceptSocket` del objeto `TcpListener`, el cual devuelve un objeto `Socket` al momento en que se realiza la conexión. El subproceso en el que se hace la llamada al método `Accept` se bloquea (es decir, se deja de ejecutar) hasta que se establece una conexión. El objeto `Socket` devuelto administra la conexión. La línea 129 pasa este objeto `Socket` como un argumento para el constructor de un objeto `NetworkStream`, el cual proporciona acceso a los flujos a través de una red. En este ejemplo, el objeto `NetworkStream` utiliza los flujos del `Socket` especificado. Las líneas 132-133 crean instancias de las clases `BinaryWriter` y `BinaryReader` para escribir y leer datos. Pasamos el objeto `NetworkStream` como un argumento para cada constructor; `BinaryWriter` puede escribir bytes en el objeto `NetworkStream`, y `BinaryReader` puede leer bytes del objeto `NetworkStream`. La línea 135 llama a `MostrarMensaje`, indicando que se recibió una conexión. A continuación, enviamos un mensaje al cliente indicando que se recibió la conexión. El método `Write` de `BinaryWriter` tiene muchas versiones sobrecargadas que escriben datos de varios tipos a un flujo. La línea 138 utiliza el método `Write` para enviar un objeto `string` al cliente, notificándole acerca de la conexión exitosa. Esto completa el *paso 3*.

Recibir mensajes del cliente

A continuación empezaremos la fase de procesamiento (*paso 4*). Las líneas 145-161 declaran una instrucción `do...while` que se ejecuta hasta que el servidor recibe un mensaje indicando la terminación de la conexión (es decir, `CLIENTE>>> TERMINAR`). La línea 150 utiliza el método `ReadString` de `BinaryReader` para leer un objeto

string del flujo. El método `ReadString` se bloquea hasta que se lea un objeto `string`. Ésta es la razón por la que ejecutamos el método `EjecutarServidor` en un subproceso separado (el cual se crea en las líneas 27-28, cuando se carga el formulario). Este subproceso asegura que el usuario de nuestra aplicación `Servidor` pueda seguir interactuando con la GUI para enviar mensajes al cliente, aun cuando este subproceso esté bloqueado, esperando un mensaje del cliente.

Modificar los controles de la GUI desde subprocesos separados

Los controles `Windows Forms` no son seguros para los subprocesos; no se garantiza que un control que se modifica desde varios subprocesos se modifique en forma correcta. La documentación de `Visual Studio 2005`¹ recomienda que sólo el subproceso que creó la GUI es el que debe modificar los controles. La clase `Control` proporciona el método ***Invoke*** para ayudar a asegurar esto. `Invoke` recibe dos argumentos: un `delegate` que representa a un método que modificará la GUI, y un arreglo de objetos `object` que representan los parámetros del método. En algún punto después de llamar a `Invoke`, el subproceso que creó originalmente la GUI ejecutará (cuando no esté ejecutando algún otro código) el método representado por el objeto `delegate`, y le pasará el contenido del arreglo `object` como argumentos para el método.

La línea 40 declara un tipo `delegate` llamado `DisplayDelegate`, el cual representa a los métodos que toman un argumento `string` y que no devuelven un valor. El método `MostrarMensaje` (líneas 44-56) cumple con esos requerimientos: recibe un parámetro `string` llamado `mensaje` y no devuelve un valor. La instrucción `if` en la línea 47 evalúa la propiedad ***InvokeRequired*** (heredada de la clase `Control`) de `mostrarTextBox`, la cual devuelve `true` si el subproceso actual no puede modificar este control en forma directa, y devuelve `false` en caso contrario. Si el subproceso actual que ejecuta al método `MostrarMensaje` no es el subproceso que creó la GUI, entonces la condición `if` se evalúa a `true` y las líneas 51-52 llaman al método `Invoke`, al cual le pasan un nuevo `DisplayDelegate`, que representa al método `MostrarMensaje` *en sí* y a un nuevo arreglo `object`, que consiste en el argumento `string` `mensaje`. Esto hace que el subproceso que creó la GUI llame nuevamente al método `MostrarMensaje`, con el mismo argumento `string` que el de la llamada original. Cuando esa llamada ocurre desde el subproceso que creó la GUI, el método *puede* modificar `mostrarTextBox` en forma directa, por lo que se ejecuta el cuerpo `else` (línea 55) y se adjunta `mensaje` a la propiedad `Text` de `mostrarTextBox`.

Las líneas 60-76 proporcionan una definición de `delegate`, `DisableInputDelegate` y un método llamado `DeshabilitarEntrada`, para permitir que cualquier subproceso modifique la propiedad `ReadOnly` de `entradaTextBox`, utilizando las mismas técnicas. Un subproceso llama a `DeshabilitarEntrada` con un argumento `bool` (`true` para deshabilitar; `false` para habilitar). Si `DeshabilitarEntrada` no puede modificar el control desde el subproceso actual, `DeshabilitarEntrada` llama al método `Invoke`. Esto hace que el subproceso que creó la GUI llame posteriormente a `DeshabilitarEntrada` y establezca `entradaTextBox.ReadOnly` al valor del argumento `bool`.

Terminar la conexión con el cliente

Cuando se completa el chat, las líneas 166-169 cierran los objetos `BinaryWriter`, `BinaryReader`, `NetworkStream` y `Socket` (*paso 5*), invocando a sus respectivos métodos `Close`. Después, el servidor espera otra solicitud de conexión de un cliente; para ello, regresa al principio del ciclo `while` (línea 121).

Enviar mensajes al cliente

Cuando el usuario de la aplicación servidor introduce un objeto `string` en el control `TextBox` y oprime la tecla *Intro*, el manejador de eventos `entradaTextBox_KeyDown` (líneas 79-101) lee el objeto `string` y lo envía a través del método `Write` de la clase `BinaryWriter`. Si un usuario termina la aplicación servidor, la línea 92 llama al método `Close` del objeto `Socket` para cerrar la conexión.

Terminar la aplicación servidor

Las líneas 32-36 definen el manejador de eventos `ServidorChatForm_FormClosing` para el evento `FormClosing`. El evento cierra la aplicación y llama al método `Exit` de la clase `Environment` con el parámetro `ExitCode`, para terminar todos los subprocesos. El método `Exit` de la clase `Environment` cierra todos los subprocesos asociados con la aplicación.

1. El artículo de MSDN “Cómo: Realizar llamadas seguras para subprocesos en controles de formularios `Windows Forms`” se encuentra en [msdn2.microsoft.com/es-es/library/ms171728\(VS.80\).aspx](http://msdn2.microsoft.com/es-es/library/ms171728(VS.80).aspx).

Clase ClienteChatForm

La figura 23.2 lista el código para la clase `ClienteChatForm`. Al igual que el objeto `ServidorChatForm`, el objeto `ClienteChatForm` crea un objeto `Thread` (líneas 26-27) en su constructor para manejar todos los mensajes entrantes. El método `EjecutarCliente` de `ClienteChatForm` (líneas 96-151) se conecta al objeto `ServidorChatForm`, recibe datos de `ServidorChatForm` y envía los datos a `ServerChatForm`. Las líneas 106-107 crean una instancia de un objeto `TcpClient` y después llaman al método `Connect` para establecer una conexión (*paso 1*). El primer argumento para el método `Connect` es el nombre del servidor; en nuestro caso, el nombre del servidor es "localhost", lo que significa que el servidor está ubicado en la misma máquina que el cliente. El nombre `localhost` también se conoce como *dirección de retorno de bucle (loopback) IP*, y es equivalente a la dirección IP 127.0.0.1. Este valor envía la transmisión de datos de vuelta a la dirección IP del emisor. [Nota: Elegimos demostrar la relación cliente/servidor mediante la conexión entre programas que se ejecutan en la misma computadora (`localhost`). Por lo general, este argumento contiene la dirección de Internet de otra computadora.] El segundo argumento para el método `Connect` es el número de puerto del servidor. Este número debe coincidir con el número de puerto en el que el servidor espera las conexiones.

El formulario `ClienteChatForm` usa un `NetworkStream` para enviar datos hacia, y recibir datos del servidor. El cliente obtiene el objeto `NetworkStream` en la línea 110, a través de una llamada al método `GetStream` de `TcpClient` (*paso 2*). La instrucción `do...while` en las líneas 120-134 itera hasta que el cliente reciba el mensaje de terminación de la conexión (`SERVIDOR>>> TERMINAR`). La línea 126 utiliza el método `ReadString` de `BinaryReader` para obtener el siguiente mensaje del servidor (*paso 3*). La línea 127 muestra el mensaje, y las líneas 137-140 cierran los objetos `BinaryWriter`, `BinaryReader`, `NetworkStream` y `TcpClient` (*paso 4*).

Las líneas 39-75 declaran los métodos `DisplayDelegate`, `MostrarMensaje`, `DisableInputDelegate` y `DeshabilitarEntrada`, exactamente igual que en las líneas 40-76 de la figura 23.1. Estos métodos se utilizan una vez más para asegurar que la GUI se modifique sólo por el subproceso que creo los controles de la GUI.

Cuando el usuario de la aplicación cliente escribe una cadena en el control `TextBox` y oprime la tecla *Intro*, el manejador de eventos `entrada_TextBox_KeyDown` (líneas 78-93) lee el objeto `string` del control `TextBox` y lo envía a través del método `Write` de `BinaryWriter`. Observe aquí que `ServidorChatForm` recibe una conexión, la procesa, la cierra y espera a la siguiente. En una aplicación real, lo más común es que un servidor reciba una conexión, la configure para que se procese como un subproceso separado de ejecución, y espere nuevas conexiones. Así, los subprocesos separados que procesan las conexiones existentes podrían seguir ejecutándose mientras el servidor se concentra en las nuevas solicitudes de conexión.

```

1  // Fig. 23.2: ClienteChat.cs
2  // Establece un cliente que enviará información hacia,
3  // y leerá información de, un servidor.
4  using System;
5  using System.Windows.Forms;
6  using System.Threading;
7  using System.Net.Sockets;
8  using System.IO;
9
10 public partial class ClienteChatForm : Form
11 {
12     public ClienteChatForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     private NetworkStream salida; // flujo para recibir datos
18     private BinaryWriter escritor; // facilita la escritura en el flujo
19     private BinaryReader lector; // facilita la lectura del flujo
20     private Thread lecturaThread; // Thread para procesar mensajes entrantes
21     private string mensaje = "";
22

```

Figura 23.2 | Parte relacionada al cliente de una conexión cliente/servidor con sockets de flujo. (Parte 1 de 5).

```

23 // inicializa el subproceso para lectura
24 private void ClienteChatForm_Load( object sender, EventArgs e )
25 {
26     lecturaThread = new Thread( new ThreadStart( EjecutarCliente ) );
27     lecturaThread.Start();
28 } // fin del método ClienteChatForm_Load
29
30 // cierra todos los subprocesos asociados con esta aplicación
31 private void ClienteChatForm_FormClosing( object sender,
32     FormClosingEventArgs e )
33 {
34     System.Environment.Exit( System.Environment.ExitCode );
35 } // fin del método ClienteChatForm_FormClosing
36
37 // delegado que permite llamar al método MostrarMensaje
38 // en el subproceso que crea y mantiene la GUI
39 private delegate void DisplayDelegate( string message );
40
41 // el método MostrarMensaje establece la propiedad Text de mostrarTextBox
42 // en una manera segura para el subproceso
43 private void MostrarMensaje( string mensaje )
44 {
45     // si la modificación de mostrarTextBox no es segura para el subproceso
46     if ( mostrarTextBox.InvokeRequired )
47     {
48         // usa el método heredado Invoke para ejecutar MostrarMensaje
49         // a través de un delegado
50         Invoke( new DisplayDelegate( MostrarMensaje ),
51             new object[] { mensaje } );
52     } // fin de if
53     else // Se puede modificar mostrarTextBox en el subproceso actual
54         mostrarTextBox.Text += mensaje;
55 } // fin del método MostrarMensaje
56
57 // delegado que permite llamar al método DeshabilitarSalida
58 // en el subproceso que crea y mantiene la GUI
59 private delegate void DisableInputDelegate( bool value );
60
61 // el método DeshabilitarSalida establece la propiedad ReadOnly de entradaTextBox
62 // de una manera segura para el subproceso
63 private void DeshabilitarSalida( bool valor )
64 {
65     // si la modificación de entradaTextBox no es segura para el subproceso
66     if ( entradaTextBox.InvokeRequired )
67     {
68         // usa el método heredado Invoke para ejecutar a DeshabilitarSalida
69         // a través de un delegado
70         Invoke( new DisableInputDelegate( DeshabilitarSalida ),
71             new object[] { valor } );
72     } // fin de if
73     else // Se puede modificar entradaTextBox en el subproceso actual
74         entradaTextBox.ReadOnly = valor;
75 } // fin del método DeshabilitarSalida
76
77 // envía al servidor el texto que escribe el usuario
78 private void entradaTextBox_KeyDown( object sender, KeyEventArgs e )
79 {
80     try
81     {

```

Figura 23.2 | Parte relacionada al cliente de una conexión cliente/servidor con sockets de flujo. (Parte 2 de 5).

```

82         if ( e.KeyCode == Keys.Enter && entradaTextBox.ReadOnly == false )
83         {
84             escritor.Write( "CLIENTE>>> " + entradaTextBox.Text );
85             mostrarTextBox.Text += "\r\nCLIENTE>>> " + entradaTextBox.Text;
86             entradaTextBox.Clear();
87         } // fin de if
88     } // fin de try
89     catch ( SocketException )
90     {
91         mostrarTextBox.Text += "\nError al escribir el objeto";
92     } // fin de catch
93 } // fin del método entradaTextBox_KeyDown
94
95 // se conecta al servidor y muestra el texto generado por el servidor
96 public void EjecutarCliente()
97 {
98     TcpClient cliente;
99
100    // crea instancia de TcpClient para enviar datos al servidor
101    try
102    {
103        MostrarMensaje( "Tratando de conectar\r\n" );
104
105        // Paso 1: crear TcpClient y conectar al servidor
106        cliente = new TcpClient();
107        cliente.Connect( "127.0.0.1", 50000 );
108
109        // Paso 2: obtener NetworkStream asociado con TcpClient
110        salida = cliente.GetStream();
111
112        // crea objetos para escribir y leer a través del flujo
113        escritor = new BinaryWriter( salida );
114        lector = new BinaryReader( salida );
115
116        MostrarMensaje( "\r\nSe recibieron flujos de E/S\r\n" );
117        DeshabilitarSalida( false ); // habilita entradaTextBox
118
119        // itera hasta que el servidor indica la terminación
120        do
121        {
122            // Paso 3: fase de procesamiento
123            try
124            {
125                // lee mensaje del servidor
126                mensaje = lector.ReadString();
127                MostrarMensaje( "\r\n" + mensaje );
128            } // fin de try
129            catch ( Exception )
130            {
131                // maneja excepcion si hay error al leer datos del servidor
132                System.Environment.Exit( System.Environment.ExitCode );
133            } // fin de catch
134        } while ( mensaje != "SERVIDOR>>> TERMINAR" );
135
136        // Paso 4: cierra la conexión
137        escritor.Close();
138        lector.Close();
139        salida.Close();
140        cliente.Close();

```

Figura 23.2 | Parte relacionada al cliente de una conexión cliente/servidor con sockets de flujo. (Parte 3 de 5).

```

141
142     Application.Exit();
143 } // fin de try
144 catch ( Exception error )
145 {
146     // maneja excepción si hay error al establecer la conexión
147     MessageBox.Show( error.ToString(), "Error en la conexión",
148         MessageBoxButtons.OK, MessageBoxIcon.Error );
149     System.Environment.Exit( System.Environment.ExitCode );
150 } // fin de catch
151 } // fin del método EjecutarCliente
152 } // fin de la clase ClienteChatForm

```

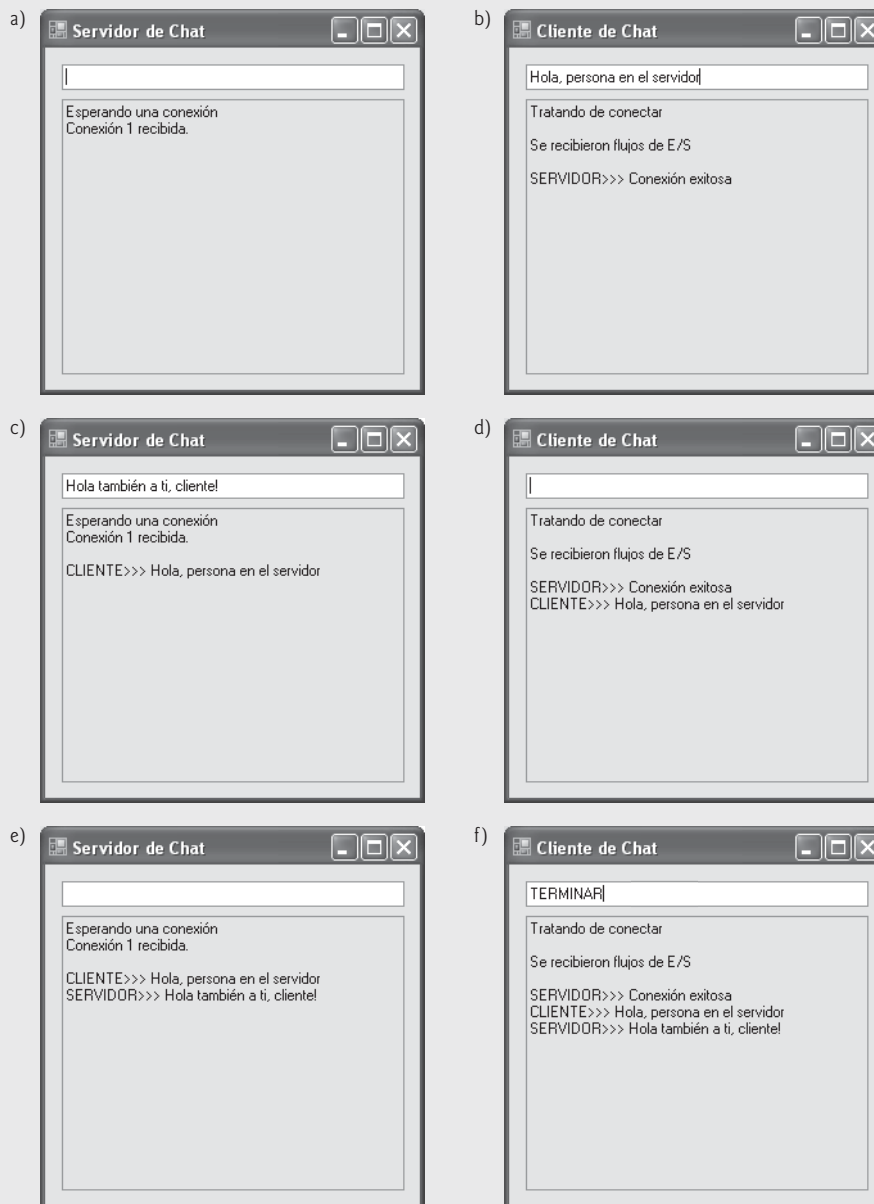


Figura 23.2 | Parte relacionada al cliente de una conexión cliente/servidor con sockets de flujo. (Parte 4 de 5).

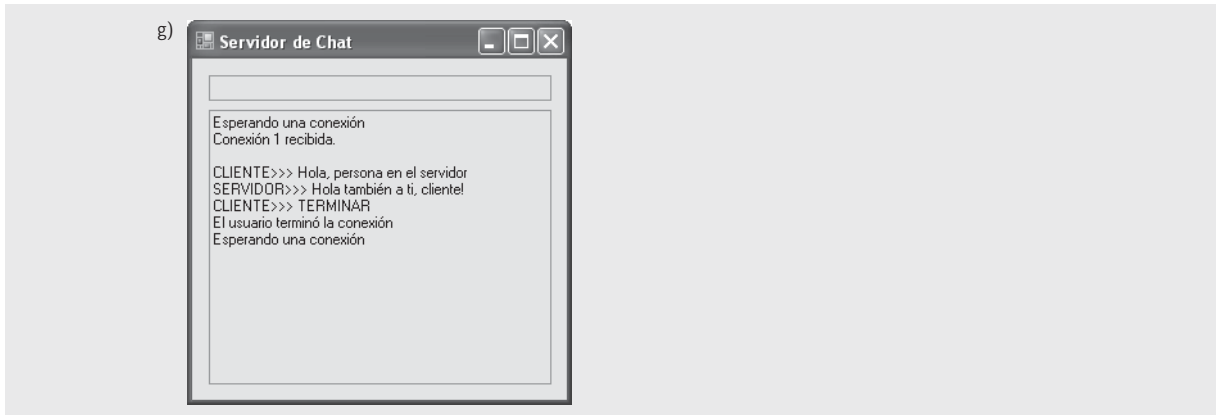


Figura 23.2 | Parte relacionada al cliente de una conexión cliente/servidor con sockets de flujo. (Parte 5 de 5).

23.7 Interacción entre cliente/servidor sin conexión mediante datagramas

Hasta este punto, hemos hablado sobre las transmisiones orientadas a la conexión y basadas en flujos que utilizan el protocolo TCP para asegurar que los paquetes de datos se transmitan en forma confiable. Ahora veremos las transmisiones sin conexión que utilizan datagramas y UDP.

La transmisión sin conexión a través de datagramas se asemeja al método mediante el cual el servicio postal transporta y entrega el correo. La transmisión sin conexión empaqueta y envía la información en paquetes conocidos como datagramas, que se pueden considerar como algo similar a las cartas que se envían por el correo convencional. Si un mensaje grande no cabe en un solo sobre, ese mensaje se descompone en piezas separadas y se coloca en sobres separados, numerados en secuencia. Todas las cartas se envían por correo al mismo tiempo, y podrían llegar en orden, desordenadas o tal vez no lleguen. La persona en el extremo receptor vuelve a ensamblar las piezas del mensaje en orden secuencial, antes de tratar de interpretar el mensaje. Si éste es lo bastante pequeño como para caber en un sobre, se elimina el problema de secuencia, pero aún es posible que el mensaje nunca llegue. (A diferencia del correo postal, los duplicados de los datagramas podrían llegar a las computadoras receptoras.) C# cuenta con la clase `UdpClient` para la transmisión sin conexión. Al igual que `TcpListener` y `TcpClient`, `UdpClient` usa métodos de la clase `Socket`. Los métodos `Send` y `Receive` de `UdpClient` transmiten datos con el método `SendTo` de `Socket` y leen datos con el método `ReceiveFrom` de `Socket`, respectivamente.

Los programas en las figuras 23.3 y 23.4 utilizan datagramas para enviar paquetes de información entre las aplicaciones cliente y servidor. En la aplicación `ClientePaquetes`, el usuario escribe un mensaje en un control `TextBox` y oprime *Intro*. El cliente convierte el mensaje en un arreglo `byte` y lo envía al servidor. El servidor recibe el paquete y muestra la información que contiene; después *hace eco* y devuelve el paquete al cliente. Cuando el cliente recibe el paquete, muestra la información que contiene. En este ejemplo, las implementaciones de las clases `ClientePaquetesForm` y `ServidorPaquetesForm` son similares.

Clase `ServidorPaquetesForm`

El código de la figura 23.3 define el formulario `ServidorPaquetesForm` para esta aplicación. La línea 23 en el manejador de eventos para la clase `ServidorPaquetesForm` crea una instancia de la clase `UdpClient`, la cual recibe datos en el puerto 50000. Esto inicializa el `Socket` subyacente para las conexiones. La línea 24 crea una instancia de la clase `IpEndPoint` para almacenar la dirección IP y número de(l) cliente(s) que transmiten a `ServidorPaquetesForm`. El primer argumento para el constructor `IpEndPoint` es un objeto `IPAddress`; el segundo argumento es el número de puerto del punto final. Estos dos valores son 0, ya que sólo necesitamos instanciar un objeto `IpEndPoint` vacío. Las direcciones IP y los números de puerto de los clientes se copian en el objeto `IpEndPoint` cuando se reciben los datagramas de los clientes.

Las líneas 39-55 definen a `DisplayDelegate` y `MostrarMensaje`, lo cual permite que cualquier subproceso pueda modificar la propiedad `Text` de `mostrarTextBox`.

```

1  // Fig. 23.3: ServidorPaquetes.cs
2  // Establece un servidor que recibe paquetes de un
3  // cliente y los envía de vuelta al cliente.
4  using System;
5  using System.Windows.Forms;
6  using System.Net;
7  using System.Net.Sockets;
8  using System.Threading;
9
10 public partial class ServidorPaquetesForm : Form
11 {
12     public ServidorPaquetesForm()
13     {
14         InitializeComponent();
15     } // fin de constructor
16
17     private UdpClient cliente;
18     private IPEndPoint puntoRecepcion;
19
20     // inicializa las variables y el subproceso para recibir paquetes
21     private void ServidorPaquetesForm_Load( object sender, EventArgs e )
22     {
23         cliente = new UdpClient( 50000 );
24         puntoRecepcion = new IPEndPoint( new IPAddress( 0 ), 0 );
25         Thread lecturaThread =
26             new Thread( new ThreadStart( EsperarPaquetes ) );
27         lecturaThread.Start();
28     } // fin del método ServidorPaquetesForm_Load
29
30     // cierra el servidor
31     private void ServidorPaquetesForm_FormClosing( object sender,
32         FormClosingEventArgs e )
33     {
34         System.Environment.Exit( System.Environment.ExitCode );
35     } // fin del método ServidorPaquetesForm_FormClosing
36
37     // delegado que permite llamar al método MostrarMensaje
38     // en el subproceso que crea y mantiene la GUI
39     private delegate void DisplayDelegate( string message );
40
41     // el método MostrarMensaje establece la propiedad Text de mostrarTextBox
42     // de una manera segura para el proceso
43     private void MostrarMensaje( string mensaje )
44     {
45         // si la modificación de mostrarTextBox no es segura para el subproceso
46         if ( mostrarTextBox.InvokeRequired )
47         {
48             // usa el método heredado Invoke para ejecutar MostrarMensaje
49             // a través de un delegado
50             Invoke( new DisplayDelegate( MostrarMensaje ),
51                 new object[] { mensaje } );
52         } // fin de if
53         else // si se puede modificar mostrarTextBox en el subproceso actual
54             mostrarTextBox.Text += mensaje;
55     } // fin del método MostrarMensaje
56
57     // espera a que llegue un paquete
58     public void EsperarPaquetes()
59     {

```

Figura 23.3 | Parte correspondiente al lado servidor de una aplicación cliente/servidor sin conexión. (Parte I de 2).


```

60     while ( true )
61     {
62         // prepara el paquete
63         byte[] datos = cliente.Receive( ref puntoRecepcion );
64         MostrarMensaje( "\r\nSe recibió paquete:" +
65             "\r\nLongitud: " + datos.Length +
66             "\r\nContenido: " +
67             System.Text.Encoding.ASCII.GetString( datos ) );
68
69         // devuelve (eco) la información del paquete de vuelta al cliente
70         MostrarMensaje( "\r\n\r\nEnviando de vuelta datos al cliente..." );
71         cliente.Send( datos, datos.Length, puntoRecepcion );
72         MostrarMensaje( "\r\nPaquete enviado\r\n" );
73     } // fin de while
74 } // fin del método EsperarPaquetes
75 } // fin de la clase ServidorPaquetesForm

```

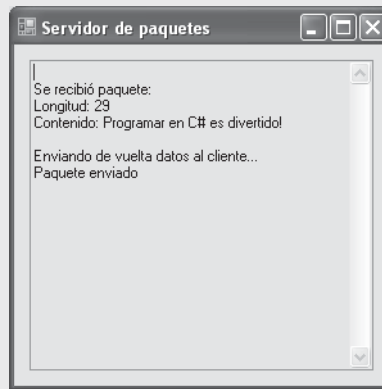


Figura 23.3 | Parte correspondiente al lado servidor de una aplicación cliente/servidor sin conexión. (Parte 2 de 2).

El método `EsperarPaquetes` de `ServidorPaquetesForm` (líneas 58-74) ejecuta un ciclo infinito mientras espera a que lleguen los datos al formulario. Cuando llega la información, el método `Receive` de `UdpClient` (línea 63) recibe un arreglo `byte` del cliente. A `Receive` le pasamos el objeto `IPEndPoint` creado en el constructor; esto proporciona al método un valor de `IPEndPoint` al que el programa copia la dirección IP y el número de puerto del cliente. Este programa se compila y ejecuta sin una excepción, aun si la referencia al objeto `IPEndPoint` es `null`, ya que el método `Receive` inicializa el objeto `IPEndPoint` si es `null`.

Las líneas 64-67 actualizan la pantalla de `ServidorPaquetesForm` para incluir la información del paquete y su contenido. La línea 71 envía los datos de vuelta al cliente, usando el método `Send` de `UdpClient`. Esta versión de `Send` recibe tres argumentos: el arreglo `byte` a enviar, un `int` que representa la longitud del arreglo y el `IPEndPoint` al que se van a enviar los datos. Utilizamos el arreglo `datos` devuelto por el método `Receive` como los datos, la longitud del arreglo `datos` como la longitud y el `IPEndPoint` que se pasa al método `Receive` como el destino de los datos. La dirección IP y el número de puerto del cliente que envió los datos se almacenan en `puntoRecepcion`, por lo que el hecho de sólo pasar `puntoRecepcion` a `Send` permite que `ServidorPaquetesForm` responda al cliente.

Clase `ClientePaquetesForm`

La clase `ClientePaquetesForm` (figura 23.4) funciona de manera similar a `ServidorPaquetesForm`, excepto que el objeto `Cliente` envía paquetes sólo cuando el usuario escribe un mensaje en un control `TextBox` y oprime *Intro*. Cuando esto ocurre, el programa llama al manejador de eventos `entradaTextBox_KeyDown` (líneas 58-75). La línea 68 convierte el objeto `string`, que el usuario introdujo en el control `TextBox`, en un arreglo `byte`. La línea 71 llama al método `Send` de `UdpClient` para enviar el arreglo `byte` al formulario `ServidorPaquetesForm` que se encuentra en `localhost` (es decir, en el mismo equipo). Especificamos el puerto como 50000, para lo cual sabemos que es el puerto de `ServidorPaquetesForm`.

Las líneas 39-55 definen a `DisplayDelegate` y `MostrarMensaje`, con lo que se permite a cualquier subproceso que modifique la propiedad `Text` del control `mostrarTextBox`.

La línea 24 crea una instancia de un objeto `UdpClient` para recibir paquetes en el puerto 50001; elegimos este puerto ya que el objeto `ServidorPaquetesForm` está ocupando el puerto 50000. El método `EsperarPaquetes` de la clase `ClientePaquetesForm` (líneas 78-90) utiliza un ciclo infinito para esperar estos paquetes. El método `Receive` de `UdpClient` se bloquea hasta que se reciba un paquete de datos (línea 83). El bloqueo realizado por

```

1  // Fig. 23.4: ClientePaquetes.cs
2  // Establece un cliente que envía paquetes a un servidor y recibe
3  // paquetes de ese servidor.
4  using System;
5  using System.Windows.Forms;
6  using System.Net;
7  using System.Net.Sockets;
8  using System.Threading;
9
10 public partial class ClientePaquetesForm : Form
11 {
12     public ClientePaquetesForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     private UdpClient cliente;
18     private IPEndPoint puntoRecepcion;
19
20     // inicializa las variables y el subproceso para recibir paquetes
21     private void ClientePaquetesForm_Load( object sender, EventArgs e )
22     {
23         puntoRecepcion = new IPEndPoint( new IPAddress( 0 ), 0 );
24         cliente = new UdpClient( 50001 );
25         Thread subproceso =
26             new Thread( new ThreadStart( EsperarPaquetes ) );
27         subproceso.Start();
28     } // fin del método ClientePaquetesForm_Load
29
30     // cierra el cliente
31     private void ClientePaquetesForm_FormClosing( object sender,
32         FormClosingEventArgs e )
33     {
34         System.Environment.Exit( System.Environment.ExitCode );
35     } // fin del método ClientePaquetesForm_FormClosing
36
37     // delegado que permite llamar al método MostrarMensaje
38     // en el subproceso que crea y mantiene la GUI
39     private delegate void DisplayDelegate( string message );
40
41     // el método MostrarMensaje establece la propiedad Text de mostrarTextBox
42     // de una manera segura para el subproceso
43     private void MostrarMensaje( string mensaje )
44     {
45         // si la modificación de mostrarTextBox no es segura para el subproceso
46         if ( mostrarTextBox.InvokeRequired )
47         {
48             // usa el método heredado Invoke para ejecutar MostrarMensaje
49             // a través de un delegado
50             Invoke( new DisplayDelegate( MostrarMensaje ),

```

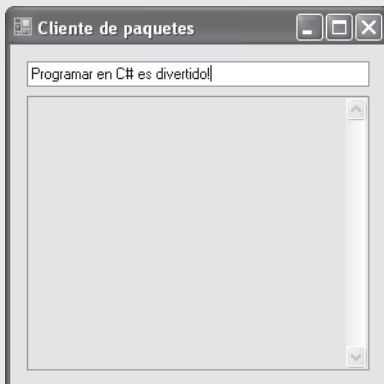
Figura 23.4 | Parte correspondiente al cliente de una aplicación cliente/servidor sin conexión. (Parte I de 2).

```

51         new object[] { mensaje } );
52     } // fin de if
53     else // sí se puede modificar mostrarTextBox en el subproceso actual
54         mostrarTextBox.Text += mensaje;
55 } // fin del método MostrarMensaje
56
57 // envía un paquete
58 private void entradaTextBox_KeyDown( object sender, EventArgs e )
59 {
60     if ( e.KeyCode == Keys.Enter )
61     {
62         // crea un paquete (datagrama) como objeto string
63         string paquete = entradaTextBox.Text;
64         mostrarTextBox.Text +=
65             "\r\nEnviando paquete que contiene: " + paquete;
66
67         // convierte el paquete en arreglo de bytes
68         byte[] datos = System.Text.Encoding.ASCII.GetBytes( paquete );
69
70         // envía el paquete al servidor en el puerto 50000
71         cliente.Send( datos, datos.Length, "127.0.0.1", 50000 );
72         mostrarTextBox.Text += "\r\nPaquete enviado\r\n";
73         entradaTextBox.Clear();
74     } // fin de if
75 } // fin del método entradaTextBox_KeyDown
76
77 // espera a que lleguen los paquetes
78 public void EsperarPaquetes()
79 {
80     while ( true )
81     {
82         // recibe arreglo de bytes del servidor
83         byte[] datos = cliente.Receive( ref puntoRecepcion );
84
85         // envía el paquete de datos al control TextBox
86         MostrarMensaje( "\r\nPaquete recibido:" +
87             "\r\nLongitud: " + datos.Length + "\r\nContenido: " +
88             System.Text.Encoding.ASCII.GetString( datos ) + "\r\n" );
89     } // fin de while
90 } // fin del método EsperarPaquetes
91 } // fin de la clase ClientePaquetesForm

```

a) La ventana Cliente de paquetes, antes de enviar un paquete al servidor



b) La ventana Cliente de paquetes, después de enviar un paquete al servidor y recibirlo de vuelta

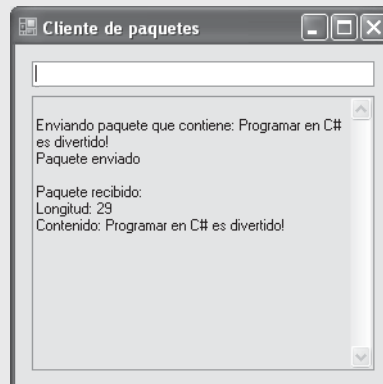


Figura 23.4 | Parte correspondiente al cliente de una aplicación cliente/servidor sin conexión. (Parte 2 de 2).

el método `Receive` no evita que la clase `ClientePaquetesForm` realice otros servicios (por ejemplo, manejar la entrada del usuario), ya que un subproceso separado ejecuta el método `EsperarPaquetes`.

Cuando llega un paquete, las líneas 86-88 muestran su contenido en el control `TextBox`. El usuario puede escribir información en el control `TextBox` de la ventana `ClientePaquetesForm` y oprimir *Intro* en cualquier momento, incluso mientras se recibe un paquete. El manejador de eventos para el control `TextBox` procesa el evento y envía los datos al servidor.

23.8 Juego de Tres en raya cliente/servidor mediante el uso de un servidor con subprocesamiento múltiple

En esta sección presentamos una versión en red del popular juego Tres en raya, implementado con sockets de flujo y técnicas de cliente/servidor. El programa consiste en una aplicación `ServidorTresEnRaya` (figura 23.5) y una aplicación `ClienteTresEnRaya` (figura 23.6). `ServidorTresEnRaya` permite que dos instancias de `ClienteTresEnRaya` se conecten al servidor y jueguen Tres en raya, una contra la otra. En la figura 23.6 se muestran los resultados. Cuando el servidor recibe una conexión de un cliente, las líneas 78-87 de la figura 23.5 crean instancias de la clase `Jugador`, para procesar a cada uno de los clientes en un subproceso de ejecución separado. Esto permite al servidor manejar las solicitudes de ambos clientes. El servidor asigna el valor "X" al primer cliente que se conecta (el jugador X hace el primer movimiento), y después asigna el valor "O" al segundo cliente. Durante el juego, el servidor mantiene información acerca del estado del tablero, de manera que pueda validar los movimientos solicitados por los jugadores. Sin embargo, ni el servidor ni el cliente pueden establecer si uno de los jugadores ganó el juego; en esta aplicación, el método `JuegoTerminado` (líneas 139-143) siempre devuelve `false`. Cada cliente mantiene su propia versión de la GUI del tablero Tres en raya para visualizar el juego. Los clientes sólo pueden colocar marcas en los lugares vacíos del tablero. La clase `Cuadro` (figura 23.7) se utiliza para definir los cuadros en el tablero de Tres en raya.

Clase `ServidorTresEnRaya`

`ServidorTresEnRaya` (figura 23.5) utiliza su manejador de eventos `Load` (líneas 27-37) para crear un arreglo `byte` que almacene los movimientos realizados por los jugadores (línea 29). El programa crea un arreglo de dos referencias a objetos `Jugador` (línea 30) y un arreglo de dos referencias a objetos `Thread` (línea 31). Cada elemento en ambos arreglos corresponde a un jugador de Tres en raya. La variable `jugadorActual` se establece a 0 (línea 32), que corresponde al jugador "X". En nuestro programa, el jugador "X" hace el primer movimiento. Las líneas 35-36 crean e inician el objeto `Thread` `obtenerJugadores`, utilizado por `ServidorTresEnRayaForm` para aceptar conexiones, de manera que el objeto `Thread` actual no se bloquee mientras espera a los jugadores.

```

1  // Fig. 23.5: ServidorTresEnRaya.cs
2  // Esta clase mantiene un juego de Tres en raya para
3  // dos aplicaciones cliente.
4  using System;
5  using System.Windows.Forms;
6  using System.Net;
7  using System.Net.Sockets;
8  using System.Threading;
9  using System.IO;
10
11 public partial class ServidorTresEnRayaForm : Form
12 {
13     public ServidorTresEnRayaForm()
14     {
15         InitializeComponent();
16     } // fin del constructor
17
18     private byte[] tablero; // la representación local del tablero del juego
19     private Jugador[] jugadores; // dos objetos Jugador

```

Figura 23.5 | Lado servidor del programa Tres en raya cliente/servidor. (Parte I de 5).

```

20  private Thread[] subprocessosJugadores; // subprocessos para la interacción con los
                                     clientes
21  private TcpListener oyente; // escucha en espera de la conexión del cliente
22  private int jugadorActual; // lleva la cuenta de quién sigue
23  private Thread obtenerJugadores; // subprocesso para adquirir las conexiones de los
                                     clientes
24  internal bool desconectado = false; // verdadero si el servidor se cierra
25
26  // inicializa las variables y el subprocesso para recibir los clientes
27  private void ServidorTresEnRayaForm_Load( object sender, EventArgs e )
28  {
29      tablero = new byte[ 9 ];
30      jugadores = new Jugador[ 2 ];
31      subprocessosJugadores = new Thread[ 2 ];
32      jugadorActual = 0;
33
34      // acepta conexiones en un subprocesso distinto
35      obtenerJugadores = new Thread( new ThreadStart( Establecer ) );
36      obtenerJugadores.Start();
37  } // fin del método ServidorTresEnRayaForm_Load
38
39  // notifica a los Jugadores para que dejen de ejecutarse
40  private void ServidorTresEnRayaForm_FormClosing( object sender,
41      FormClosingEventArgs e )
42  {
43      desconectado = true;
44      System.Environment.Exit( System.Environment.ExitCode );
45  } // fin del método ServidorTresEnRayaForm_FormClosing
46
47  // delegado que permite llamar al método MostrarMensaje
48  // en el subprocesso que crea y mantiene la GUI
49  private delegate void DisplayDelegate( string mensaje );
50
51  // el método MostrarMensaje establece la propiedad Text de mostrarTextBox
52  // de una manera segura para el subprocesso
53  internal void MostrarMensaje( string mensaje )
54  {
55      // si la modificación de mostrarTextBox no es segura para el subprocesso
56      if ( mostrarTextBox.InvokeRequired )
57      {
58          // usa el método heredado Invoke para ejecutar MostrarMensaje
59          // a través de un delegado
60          Invoke( new DisplayDelegate( MostrarMensaje ),
61              new object[] { mensaje } );
62      } // fin de if
63      else // sí se puede modificar mostrarTextBox en el subprocesso actual
64          mostrarTextBox.Text += mensaje;
65  } // fin del método MostrarMensaje
66
67  // acepta conexiones de 2 jugadores
68  public void Establecer()
69  {
70      MostrarMensaje( "Esperando a los jugadores...\r\n" );
71
72      // establece Socket
73      oyente =
74          new TcpListener( IPAddress.Parse( "127.0.0.1" ), 50000 );
75      oyente.Start();
76
77      // acepta el primer jugador e inicia un subprocesso jugador

```

Figura 23.5 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 2 de 5).

```

78 jugadores[ 0 ] = new Jugador( oyente.AcceptSocket(), this, 0 );
79 subprocesosJugadores[ 0 ] =
80     new Thread( new ThreadStart( jugadores[ 0 ].Ejecutar ) );
81 subprocesosJugadores[ 0 ].Start();
82
83 // acepta el segundo jugador e inicia otro subproceso jugador
84 jugadores[ 1 ] = new Jugador( oyente.AcceptSocket(), this, 1 );
85 subprocesosJugadores[ 1 ] =
86     new Thread( new ThreadStart( jugadores[ 1 ].Ejecutar ) );
87 subprocesosJugadores[ 1 ].Start();
88
89 // hace saber al primer jugador que el otro jugador se conectó
90 lock ( jugadores[ 0 ] )
91 {
92     jugadores[ 0 ].subprocesoSuspendido = false;
93     Monitor.Pulse( jugadores[ 0 ] );
94 } // fin de lock
95 } // fin del método Establecer
96
97 // determina si un movimiento es válido
98 public bool MovimientoValido( int ubicacion, int jugador )
99 {
100     // evita que otro subproceso haga un movimiento
101     lock ( this )
102     {
103         // mientras no sea el turno del jugador actual, espera
104         while ( jugador != jugadorActual )
105             Monitor.Wait( this );
106
107         // si el cuadro deseado no está ocupado
108         if ( !EstaOcupado( ubicacion ) )
109         {
110             // establece el tablero para que contenga la marca del jugador actual
111             tablero[ ubicacion ] = ( byte ) ( jugadorActual == 0 ?
112                 'X' : 'O' );
113
114             // establece jugadorActual para que sea el otro jugador
115             jugadorActual = ( jugadorActual + 1 ) % 2;
116
117             // notifica el movimiento al otro jugador
118             jugadores[ jugadorActual ].OtroJugadorMovio( ubicacion );
119
120             // alerta al otro jugador que es tiempo de moverse
121             Monitor.Pulse( this );
122             return true;
123         } // fin de if
124         else
125             return false;
126     } // fin de lock
127 } // fin del método MovimientoValido
128
129 // determina si el cuadro especificado está ocupado
130 public bool EstaOcupado( int ubicacion )
131 {
132     if ( tablero[ ubicacion ] == 'X' || tablero[ ubicacion ] == 'O' )
133         return true;
134     else
135         return false;
136 } // fin del método EstaOcupado

```

Figura 23.5 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 3 de 5).

```

137
138 // determina si el juego terminó
139 public bool JuegoTerminado()
140 {
141     // colocar aquí el código para evaluar si alguien ganó el juego
142     return false;
143 } // fin del método JuegoTerminado
144 } // fin de la clase ServidorTresEnRayaForm
145
146 // la clase Jugador representa a un jugador de tres en raya
147 public class Jugador
148 {
149     internal Socket conexion; // Socket para aceptar una conexión
150     private NetworkStream socketStream; // flujo de datos de red
151     private ServidorTresEnRayaForm servidor; // referencia al servidor
152     private BinaryWriter escritor; // facilita la escritura en el flujo
153     private BinaryReader lector; // facilita la lectura del flujo
154     private int numero; // número de jugador
155     private char marca; // marca del jugador en el tablero
156     internal bool subprocesoSuspendido = true; // si está esperando al otro jugador
157
158     // constructor que requiere objetos Socket, ServidorTresEnRayaForm
159     // e int como argumentos
160     public Jugador( Socket socket, ServidorTresEnRayaForm valorServidor,
161         int nuevoNumero )
162     {
163         marca = (nuevoNumero == 0 ? 'X' : 'O');
164         conexion = socket;
165         servidor = valorServidor;
166         numero = nuevoNumero;
167
168         // crea objeto NetworkStream para Socket
169         socketStream = new NetworkStream( conexion );
170
171         // crea flujos para escribir/leer bytes
172         escritor = new BinaryWriter( socketStream );
173         lector = new BinaryReader( socketStream );
174     } // fin del constructor
175
176     // indica la jugada al otro jugador
177     public void OtroJugadorMovio( int ubicacion )
178     {
179         // signal that opponent moved
180         escritor.Write( "El oponente movió." );
181         escritor.Write( ubicacion ); // envía la ubicación del movimiento
182     } // fin del método OtroJugadorMovio
183
184     // permite a los jugadores hacer movimientos y recibir los movimientos
185     // del otro jugador
186     public void Ejecutar()
187     {
188         bool listo = false;
189
190         // muestra en el servidor que se hizo una conexión
191         servidor.MostrarMensaje( "Jugador " + ( numero == 0 ? 'X' : 'O' )
192             + " conectado\r\n" );
193
194         // envía la marca del jugador actual al cliente
195         escritor.Write( marca );

```

Figura 23.5 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 4 de 5).

```

196
197 // si el número es igual a 0, entonces este jugador es X,
198 // en caso contrario, 0 debe esperar el primer movimiento de X
199 escritor.Write( "Jugador " + ( numero == 0 ?
200     "X conectado.\r\n" : "0 conectado, por favor espere.\r\n" ) );
201
202 // X debe esperar a que llegue otro jugador
203 if ( marca == 'X' )
204 {
205     escritor.Write( "Esperando al otro jugador." );
206
207     // espera la notificación del servidor, de que
208     // se conectó el otro jugador
209     lock ( this )
210     {
211         while ( subprocesoSuspendido )
212             Monitor.Wait( this );
213     } // fin de lock
214
215     escritor.Write( "Se conectó el otro jugador. Es su turno." );
216 } // fin de if
217
218 // iniciar el juego
219 while ( !listo )
220 {
221     // espera a que haya datos disponibles
222     while ( conexion.Available == 0 )
223     {
224         Thread.Sleep( 1000 );
225
226         if ( servidor.desconectado )
227             return;
228     } // fin de while
229
230     // recibe los datos
231     int ubicacion = lector.ReadInt32();
232
233     // si el movimiento es válido, lo muestra en el
234     // servidor e indica que el movimiento es válido
235     if ( servidor.MovimientoValido( ubicacion, numero ) )
236     {
237         servidor.MostrarMensaje( "ubic: " + ubicacion + "\r\n" );
238         escritor.Write( "Movimiento válido." );
239     } // fin de if
240     else // indica que el movimiento es inválido
241         escritor.Write( "Movimiento inválido, intente de nuevo." );
242
243     // si el juego terminó, establece listo a true para salir del ciclo while
244     if ( servidor.JuegoTerminado() )
245         listo = true;
246 } // fin de ciclo while
247
248 // cierra la conexión de los sockets
249 escritor.Close();
250 lector.Close();
251 socketStream.Close();
252 conexion.Close();
253 } // fin del método Ejecutar
254 } // fin de la clase Jugador

```

Figura 23.5 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 5 de 5).

Las líneas 49-65 definen a `DisplayDelegate` y `MostrarMensaje`, lo cual permite que cualquier subproceso pueda modificar la propiedad `Text` de `mostrarTextBox`. Esta vez, el método `MostrarMensaje` se declara como `internal`, por lo que puede llamarse dentro de un método de la clase `Jugador`, a través de una referencia a `ServidorTresEnRayaForm`.

El subproceso `obtenerJugadores` ejecuta el método `Establecer` (líneas 68-95), el cual crea un objeto `TcpListener` para escuchar solicitudes en el puerto 50000 (líneas 73-75). Después, este objeto escucha en espera de solicitudes de conexión de los jugadores primero y segundo. Las líneas 78 y 84 crean instancias de objetos `Jugador`, los cuales representan a los jugadores, y las líneas 79-81 y 85-87 crean dos objetos `Thread` que ejecutan los métodos `Ejecutar` de cada objeto `Jugador`.

El constructor de `Jugador` (figura 23.5, líneas 160-174) recibe como argumentos una referencia al objeto `Socket` (es decir, la conexión con el cliente), una referencia al objeto `ServidorTresEnRayaForm` y un objeto `int` que indica el número de jugador (de donde el constructor infiere la marca "X" o "O" utilizada por ese jugador). En este caso de estudio, `ServidorTresEnRayaForm` llama al método `Ejecutar` (líneas 186-253) después de instanciar un objeto `Jugador`. Las líneas 191-200 notifican al servidor de una conexión exitosa y envían al cliente el char que éste colocará en el tablero, cuando haga un movimiento. Si `Ejecutar` se está ejecutando para el Jugador "X", se ejecutan las líneas 205-215 y el Jugador "X" tiene que esperar a que se conecte un segundo jugador. Las líneas 211-212 definen una instrucción `while` que suspende el subproceso del Jugador "X" hasta que el servidor indica que el Jugador "O" se ha conectado. El servidor notifica la conexión al Jugador estableciendo la variable `subprocesoSuspendido` de `Jugador` a `false` (línea 92). Cuando `subprocesoSuspendido` se vuelve `false`, `Jugador` sale de la instrucción `while` en las líneas 211-212.

El método `Ejecutar` ejecuta la instrucción `while` en las líneas 219-246, para permitir al usuario que juegue. Cada iteración de esta instrucción espera a que el cliente envíe un valor `int` que especifica en dónde se colocará la "X" o la "O" en el tablero; después, el Jugador coloca la marca en el tablero, si la ubicación es válida (es decir, que la ubicación no contenga ya una marca). Observe que la instrucción `while` continúa su ejecución sólo si la variable `bool listo` es `false`. Esta variable se establece a `true` mediante el manejador de eventos `ServidorTresEnRayaForm_FormClosing` de la clase `ServidorTresEnRayaForm`, el cual se invoca cuando el servidor cierra la conexión.

La línea 222 de la figura 23.5 empieza una instrucción `while` que itera hasta que la propiedad `Available` de `Socket` indica que hay información proveniente del `Socket` (o hasta que el servidor se desconecte del cliente). Si no hay información, el objeto `Thread` pasa al estado inactivo durante un segundo. Al despertar, el objeto `Thread` utiliza la propiedad `Disconnected` para verificar si la variable `desconectado` del servidor es `true` (línea 226). Si el valor es `true`, el objeto `Thread` sale del método (con lo cual termina la ejecución del subproceso); en caso contrario, el objeto `Thread` entra de nuevo al ciclo. No obstante, si la propiedad `Available` indica que hay datos para recibir, termina la instrucción `while` de las líneas 222-228, permitiendo que se procese la información.

Esta información contiene un valor `int`, que representa la ubicación en la que el cliente desea colocar una marca. La línea 231 llama al método `ReadInt32` del objeto `BinaryReader` (que lee del objeto `NetworkStream` creado con el `Socket`) para leer este `int`. Después, la línea 235 pasa el `int` al método `MovimientoValido` de `ServidorTresEnRayaForm`. Si este método valida el movimiento, el Jugador coloca la marca en la ubicación deseada.

El método `MovimientoValido` (líneas 98-127) envía al cliente un mensaje que indica si el movimiento fue válido. Las ubicaciones en el tablero corresponden a los números del 0 al 8 (0-2 para la fila superior, 3-5 para la fila intermedia y 6-8 para la fila superior). Todas las instrucciones en el método `MovimientoValido` se encierran en una instrucción `lock`, que permite realizar sólo un movimiento a la vez. Esto evita que dos jugadores modifiquen la información de estado del juego al mismo tiempo. Si el Jugador que intenta validar un movimiento no es el jugador actual (es decir, el que tiene permitido hacer un movimiento), ese Jugador se coloca en un estado de *Espera* hasta que sea su turno para hacer el movimiento. Si el usuario trata de colocar una marca en una ubicación que ya contenga una marca, el método `MovimientoValido` devuelve `false`. No obstante, si el usuario seleccionó una ubicación desocupada (línea 108), las líneas 111-112 colocan la marca en la representación local del tablero. La línea 118 notifica al otro Jugador que se realizó un movimiento, y la línea 121 invoca al método `Pulse`, para que el Jugador en espera pueda validar un movimiento. Después, el método devuelve `true` para indicar que el movimiento es válido.

Cuando se ejecuta una aplicación `ClienteTresEnRayaForm` (figura 23.6), crea un control `TextBox` para mostrar los mensajes del servidor y la representación del tablero de Tres en raya. El tablero se crea a partir de nueve objetos `Cuadro` (figura 23.7), el cual contiene controles `Panel` en los que el usuario puede hacer clic, indicando


```

1  // Fig. 23.6: ClienteTresEnRaya.cs
2  // Cliente para el programa de Tres en raya.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6  using System.Net.Sockets;
7  using System.Threading;
8  using System.IO;
9
10 public partial class ClienteTresEnRayaForm : Form
11 {
12     public ClienteTresEnRayaForm()
13     {
14         InitializeComponent();
15     } // fin del constructor
16
17     private Cuadro[ , ] tablero; // representación local del tablero del juego
18     private Cuadro cuadroActual; // el Cuadro que eligió este jugador
19     private Thread subprocesoSalida; // subproceso para recibir datos del servidor
20     private TcpClient conexion; // cliente para establecer la conexión
21     private NetworkStream flujo; // flujo de datos de red
22     private BinaryWriter escritor; // facilita la escritura en el flujo
23     private BinaryReader lector; // facilita la lectura del flujo
24     private char miMarca; // la marca del jugador en el tablero
25     private bool miTurno; // ¿es el turno de este jugador?
26     private SolidBrush brocha; // brocha para dibujar Xs y Os
27     private bool listo = false; // verdadero cuando se termina el juego
28
29     // inicializa variables y subproceso para conectarse al servidor
30     private void ClienteTresEnRayaForm_Load( object sender, EventArgs e )
31     {
32         tablero = new Cuadro[ 3, 3 ];
33
34         // crea 9 objetos Cuadro y los coloca en el tablero
35         tablero[ 0, 0 ] = new Cuadro( tablero0Panel, ' ', 0 );
36         tablero[ 0, 1 ] = new Cuadro( tablero1Panel, ' ', 1 );
37         tablero[ 0, 2 ] = new Cuadro( tablero2Panel, ' ', 2 );
38         tablero[ 1, 0 ] = new Cuadro( tablero3Panel, ' ', 3 );
39         tablero[ 1, 1 ] = new Cuadro( tablero4Panel, ' ', 4 );
40         tablero[ 1, 2 ] = new Cuadro( tablero5Panel, ' ', 5 );
41         tablero[ 2, 0 ] = new Cuadro( tablero6Panel, ' ', 6 );
42         tablero[ 2, 1 ] = new Cuadro( tablero7Panel, ' ', 7 );
43         tablero[ 2, 2 ] = new Cuadro( tablero8Panel, ' ', 8 );
44
45         // crea un objeto SolidBrush para escribir en los cuadros
46         brocha = new SolidBrush( Color.Black );
47
48         // hace conexión con el servidor y obtiene el flujo
49         // de red asociado
50         conexion = new TcpClient( "127.0.0.1", 50000 );
51         flujo = conexion.GetStream();
52         escritor = new BinaryWriter( flujo );
53         lector = new BinaryReader( flujo );
54
55         // inicia un nuevo subproceso para enviar y recibir mensajes
56         subprocesoSalida = new Thread( new ThreadStart( Ejecutar ) );
57         subprocesoSalida.Start();
58     } // fin del método ClienteTresEnRayaForm_Load
59

```

Figura 23.6 | Lado cliente del programa Tres en raya cliente/servidor. (Parte I de 6).

```

60 // vuelve a pintar los cuadros
61 private void ClienteTresEnRayaForm_Paint( object sender,
62     PaintEventArgs e )
63 {
64     PintarCuadros();
65 } // fin del método ClienteTresEnRayaForm_Load
66
67 // el juego terminó
68 private void ClienteTresEnRayaForm_FormClosing( object sender,
69     FormClosingEventArgs e )
70 {
71     listo = true;
72     System.Environment.Exit( System.Environment.ExitCode );
73 } // fin de ClienteTresEnRayaForm_FormClosing
74
75 // delegado que permite llamar al método MostrarMensaje
76 // en el subproceso que crea y mantiene la GUI
77 private delegate void DisplayDelegate( string message );
78
79 // el método MostrarMensaje establece la propiedad Text de mostrarTextBox
80 // de una manera segura para el subproceso
81 private void MostrarMensaje( string mensaje )
82 {
83     // si la modificación de mostrarTextBox no es segura para el subproceso
84     if ( mostrarTextBox.InvokeRequired )
85     {
86         // usa el método heredado Invoke para ejecutar MostrarMensaje
87         // a través de un delegado
88         Invoke( new DisplayDelegate( MostrarMensaje ),
89             new object[] { mensaje } );
90     } // fin de if
91     else // sí se puede modificar mostrarTextBox en el subproceso actual
92         mostrarTextBox.Text += mensaje;
93 } // fin del método MostrarMensaje
94
95 // delegado que permite llamar al método CambiarIdLabel
96 // en el subproceso que crea y mantiene la GUI
97 private delegate void ChangeIdLabelDelegate( string message );
98
99 // el método CambiarIdLabel establece la propiedad Text de mostrarTextBox
100 // de una manera segura para el subproceso
101 private void CambiarIdLabel( string etiqueta )
102 {
103     // si la modificación de idLabel no es segura para el subproceso
104     if ( idLabel.InvokeRequired )
105     {
106         // usa el método heredado Invoke para ejecutar CambiarIdLabel
107         // a través de un delegado
108         Invoke( new ChangeIdLabelDelegate( CambiarIdLabel ),
109             new object[] { etiqueta } );
110     } // fin de if
111     else // sí se puede modificar idLabel en el subproceso actual
112         idLabel.Text = etiqueta;
113 } // fin del método CambiarIdLabel
114
115 // dibuja la marca de cada cuadro
116 public void PintarCuadros()
117 {
118     Graphics g;

```

Figura 23.6 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 2 de 6).

```

119
120 // dibuja la marca apropiada en cada panel
121 for ( int fila = 0; fila < 3; fila++ )
122 {
123     for ( int columna = 0; columna < 3; columna++ )
124     {
125         // obtiene el objeto Graphics para cada Panel
126         g = tablero[ fila, columna ].PanelCuadros.CreateGraphics();
127
128         // dibuja la letra apropiada en el panel
129         g.DrawString( tablero[ fila, columna ].Marca.ToString(),
130             tablero0Panel.Font, brocha, 10, 8 );
131     } // fin de for
132 } // fin de for
133 } // fin del método PintarCuadros
134
135 // envía al servidor la ubicación del cuadro en el que se hizo clic
136 private void cuadro_MouseUp( object sender,
137     System.Windows.Forms.MouseEventArgs e )
138 {
139     // para cada cuadro, verifica si se hizo clic en él
140     for ( int fila = 0; fila < 3; fila++ )
141     {
142         for ( int columna = 0; columna < 3; columna++ )
143         {
144             if ( tablero[ fila, columna ].PanelCuadros == sender )
145             {
146                 CuadroActual = tablero[ fila, columna ];
147
148                 // envía el movimiento al servidor
149                 EnviarCuadradoClic( tablero[ fila, columna ].Ubicacion );
150             } // fin de if
151         } // fin de for
152     } // fin de for
153 } // fin del método cuadro_MouseUp
154
155 // subproceso de control que permite la actualización continua
156 // del control TextBox en la pantalla
157 public void Ejecutar()
158 {
159     // primero obtiene la marca del jugador (X o O)
160     miMarca = lector.ReadChar();
161     CambiarIdLabel( "Usted es el jugador \"\" + miMarca + \"\" );
162     miTurno = ( miMarca == 'X' ? true : false );
163
164     // procesa los mensajes entrantes
165     try
166     {
167         // recibe los mensajes que se envían al cliente
168         while ( !listo )
169             ProcesarMensaje( lector.ReadString() );
170     } // fin de try
171     catch ( IOException )
172     {
173         MessageBox.Show( "Servidor desconectado, el juego terminó", "Error",
174             MessageBoxButtons.OK, MessageBoxIcon.Error );
175     } // fin de catch
176 } // fin del método Ejecutar
177

```

Figura 23.6 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 3 de 6).

```

178 // procesa los mensajes enviados al cliente
179 public void ProcesarMensaje( string mensaje )
180 {
181     // si el movimiento que envió el jugador al servidor es válido
182     // actualiza la pantalla, establece la marca de ese cuadro para que sea
183     // la marca del jugador actual y vuelve a pintar el tablero
184     if ( mensaje == "Movimiento válido." )
185     {
186         MostrarMensaje( "Movimiento válido, por favor espere.\r\n" );
187         cuadroActual.Marca = miMarca;
188         PintarCuadros();
189     } // fin de if
190     else if ( mensaje == "Movimiento inválido, intente de nuevo." )
191     {
192         // si el movimiento es inválido, muestra ese mensaje y ahora
193         // es turno de este jugador otra vez
194         MostrarMensaje( mensaje + "\r\n" );
195         miTurno = true;
196     } // fin de else if
197     else if ( mensaje == "El oponente movió." )
198     {
199         // si el oponente movió, busca la ubicación de su movimiento
200         int ubicacion = lector.ReadInt32();
201
202         // establece ese cuadro para que tenga la marca del oponente y
203         // vuelve a pintar el tablero
204         tablero[ ubicacion / 3, ubicacion % 3 ].Marca =
205             ( miMarca == 'X' ? 'O' : 'X' );
206         PintarCuadros();
207
208         MostrarMensaje( "El oponente movió. Es su turno.\r\n" );
209
210         // ahora es el turno de este jugador
211         miTurno = true;
212     } // fin de else if
213     else
214         MostrarMensaje( mensaje + "\r\n" ); // muestra el mensaje
215 } // fin del método ProcesarMensaje
216
217 // envía al servidor el número del cuadro en el que se hizo clic
218 public void EnviarCuadradoClic( int ubicacion )
219 {
220     // si es el movimiento del jugador actual justo ahora
221     if ( miTurno )
222     {
223         // envía la ubicación del movimiento al servidor
224         escritor.Write( ubicacion );
225
226         // si es ahora el turno del otro jugador
227         miTurno = false;
228     } // fin de if
229 } // fin del método EnviarCuadradoClic
230
231 // propiedad de sólo escritura para el cuadro actual
232 public Cuadro CuadroActual
233 {
234     set
235     {
236         cuadroActual = value;

```

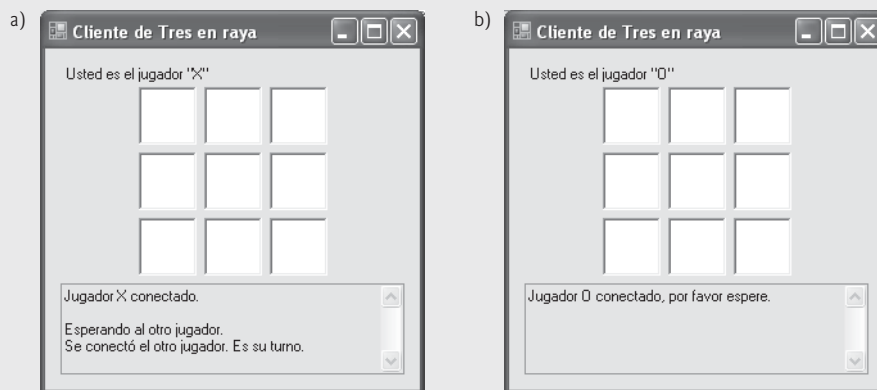
Figura 23.6 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 4 de 6).

```

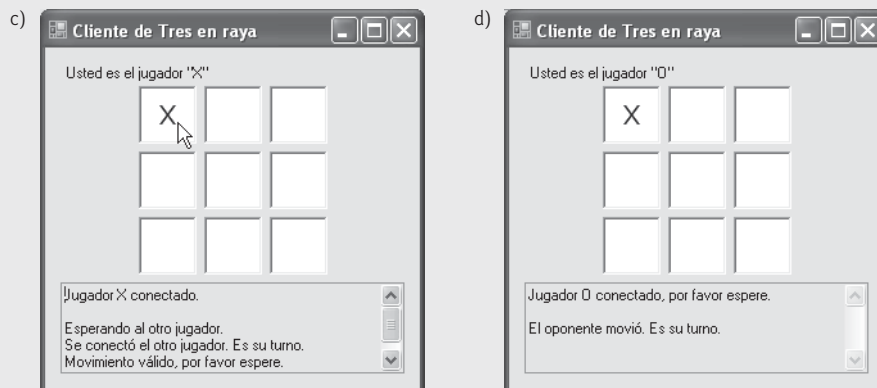
237     } // fin de set
238     } // fin de la propiedad CuadroActual
239 } // fin de la clase ClienteTresEnRayaForm

```

Al principio del juego.



Después que el jugador X hace el primer movimiento.



Después que el jugador O hace el segundo movimiento.

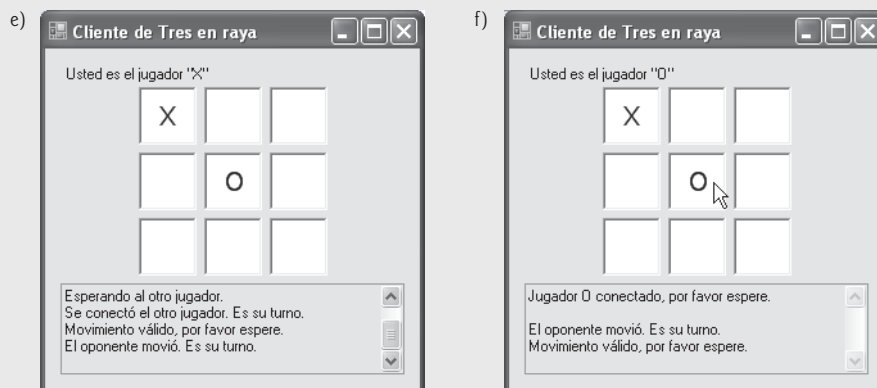


Figura 23.6 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 5 de 6).

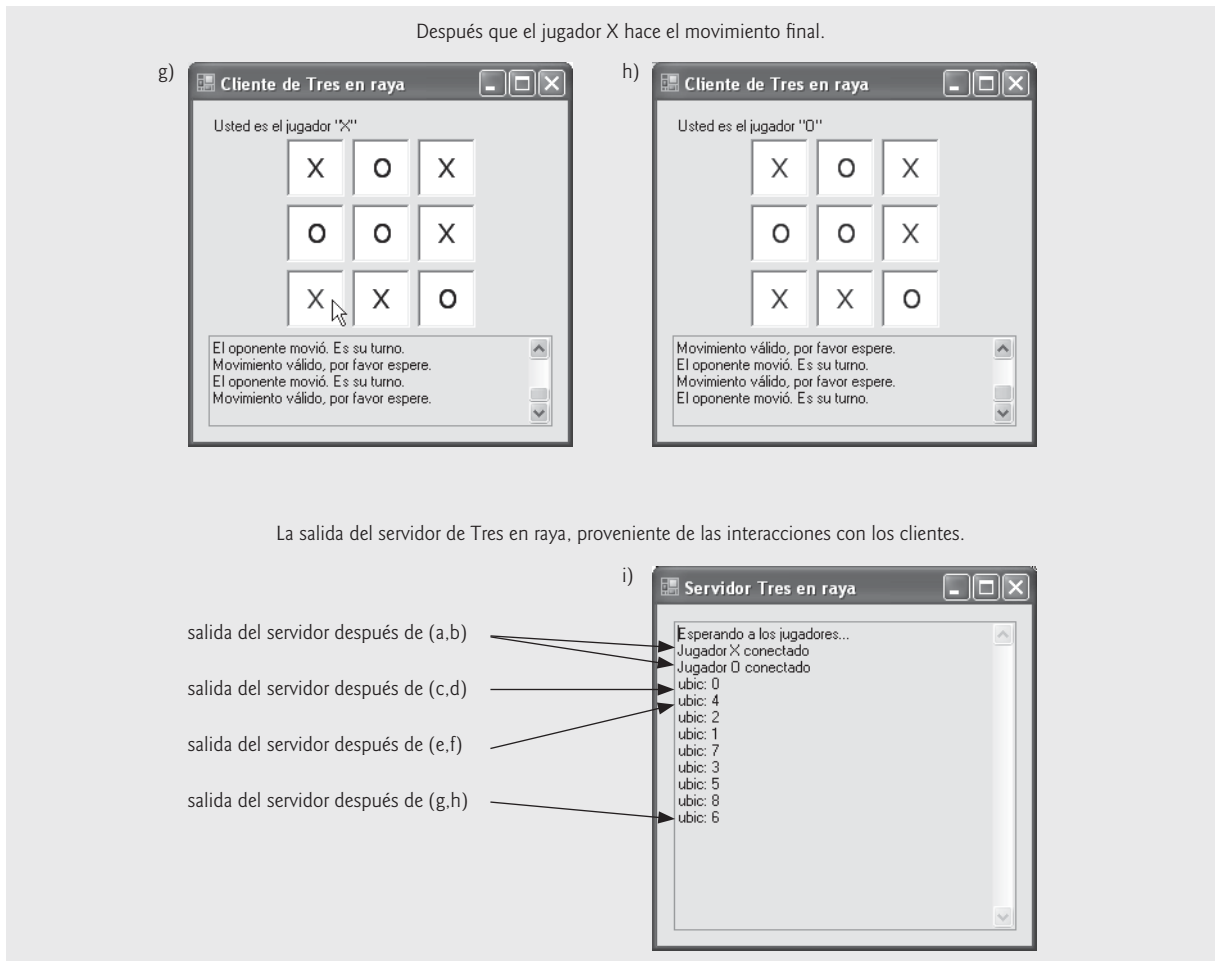


Figura 23.6 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 6 de 6).

```

1  // Fig. 23.7: Cuadro.cs
2  // Un Cuadro en el tablero de Tres en raya.
3  using System.Windows.Forms;
4
5  // la representación de un cuadro en una cuadrícula de tres en raya
6  public class Cuadro
7  {
8      private Panel panel; // panel de la GUI que representa este Cuadro
9      private char marca; // la marca del jugador en este Cuadro (si la hay)
10     private int ubicacion; // ubicacion en el tablero de este Cuadro
11
12     // constructor
13     public Cuadro( Panel nuevoPanel, char nuevaMarca, int nuevaUbicacion )
14     {
15         panel = nuevoPanel;
16         marca = nuevaMarca;
17         ubicacion = nuevaUbicacion;
18     } // fin del constructor
19

```

Figura 23.7 | La clase Cuadro. (Parte 1 de 2).

```

20 // propiedad PanelCuadros; el panel que representa el cuadro
21 public Panel PanelCuadros
22 {
23     get
24     {
25         return panel;
26     } // fin de get
27 } // fin de la propiedad PanelCuadros
28
29 // propiedad Marca; la marca en el cuadro
30 public char Marca
31 {
32     get
33     {
34         return marca;
35     } // fin de get
36     set
37     {
38         marca = value;
39     } // fin de set
40 } // fin de la propiedad Marca
41
42 // propiedad Ubicacion; la ubicación del cuadro en el tablero
43 public int Ubicacion
44 {
45     get
46     {
47         return ubicacion;
48     } // fin de get
49 } // fin de la propiedad Ubicacion
50 } // fin de la clase Cuadro

```

Figura 23.7 | La clase Cuadro. (Parte 2 de 2).

la posición en el tablero en la que se debe colocar la marca. El manejador de eventos Load de `ClienteTresEnRayaForm` (líneas 30-58) abre una conexión con el servidor (línea 50) y obtiene una referencia al objeto `NetworkStream` asociado de la conexión, de `TcpClient` (línea 51). Las líneas 56-57 inician un subproceso para leer los mensajes enviados del servidor al cliente. El servidor pasa los mensajes (por ejemplo, si cada movimiento es válido) al método `ProcesarMensaje` (líneas 179-215). Si el mensaje indica que un movimiento es válido (línea 184), el cliente establece su `Marca` en el cuadro actual (el cuadro en el que el usuario hizo clic) y vuelve a pintar el tablero. Si el mensaje indica que un movimiento es inválido (línea 190), el cliente notifica al usuario que haga clic en un cuadro distinto. Si el mensaje indica que el oponente hizo un movimiento (línea 197), la línea 200 lee un `int` del servidor, especificando en qué parte del tablero debe el cliente colocar la `Marca` del oponente. `ClienteTresEnRayaForm` incluye un par `delegate/método` para permitir que los subprocesos modifiquen la propiedad `Text` de `idLabel` (líneas 97-113), así como los métodos `DisplayDelegate` y `MostrarMensaje` para modificar la propiedad `Text` de `mostrarTextBox` (líneas 77-93).

23.9 Control WebBrowser

Con la FCL 2.0, Microsoft introdujo el control `WebBrowser`, que permite a las aplicaciones incorporar herramientas de exploración Web. El control cuenta con métodos para navegar en páginas Web y mantiene su propio historial de los sitios Web visitados. También genera eventos, a medida que el usuario interactúa con el contenido que se muestra en el control, por lo que su aplicación puede responder a eventos como cuando el usuario hace clic en los vínculos que aparecen en el contenido.

La figura 23.8 demuestra las capacidades del control `WebBrowser`. La clase `ExploradorForm` proporciona la funcionalidad básica de un explorador Web, ya que permite al usuario navegar hacia un URL, desplazarse hacia atrás y hacia delante por el historial de sitios visitados y volver a cargar la página Web actual.

```

1  // Fig. 23.8: Explorador.cs
2  // Ejemplo del control WebBrowser.
3  using System;
4  using System.Windows.Forms;
5
6  public partial class ExploradorForm : Form
7  {
8      public ExploradorForm()
9      {
10         InitializeComponent();
11     } // fin del constructor
12
13     // navega una página hacia atrás
14     private void atrasButton_Click( object sender, EventArgs e )
15     {
16         webBrowser.GoBack();
17     } // fin del método atrasButton_Click
18
19     // navega una página hacia delante
20     private void adelanteButton_Click( object sender, EventArgs e )
21     {
22         webBrowser.GoForward();
23     } // fin del método adelanteButton_Click
24
25     // deja de cargar la página actual
26     private void detenerButton_Click( object sender, EventArgs e )
27     {
28         webBrowser.Stop();
29     } // fin del método detenerButton_Click
30
31     // vuelve a cargar la página actual
32     private void actualizarButton_Click( object sender, EventArgs e )
33     {
34         webBrowser.Refresh();
35     } // fin del método actualizarButton_Click
36
37     // navega a la página inicial del usuario
38     private void inicioButton_Click( object sender, EventArgs e )
39     {
40         webBrowser.GoHome();
41     } // fin del método inicioButton_Click
42
43     // si el usuario oprimió Intro, navega al URL especificado
44     private void navegacionTextBox_KeyDown( object sender,
45         KeyEventArgs e )
46     {
47         if ( e.KeyCode == Keys.Enter )
48             webBrowser.Navigate( navegacionTextBox.Text );
49     } // fin del método navegacionTextBox_KeyDown
50
51     // habilita detenerButton mientras la página actual se está cargando
52     private void webBrowser_Navigating( object sender,
53         WebBrowserNavigatingEventArgs e )
54     {
55         detenerButton.Enabled = true;
56     } // fin del método webBrowser_Navigating
57
58     // actualiza el texto de estado
59     private void webBrowser_StatusTextChanged( object sender,

```

Figura 23.8 | Ejemplo del control WebBrowser. (Parte I de 2).


```

60     EventArgs e )
61 {
62     estadoTextBox.Text = webBrowser.StatusText;
63 } // fin del método webBrowser_StatusTextChanged
64
65 // actualiza el control ProgressBar con base en el porcentaje descargado de la página
66 private void webBrowser_ProgressChanged( object sender,
67     WebBrowserProgressChangedEventArgs e )
68 {
69     paginaProgressBar.Value =
70         ( int ) ( ( 100 * e.CurrentProgress ) / e.MaximumProgress );
71 } // fin del método webBrowser_ProgressChanged
72
73 // actualiza los controles del explorador web en forma apropiada
74 private void webBrowser_DocumentCompleted( object sender,
75     WebBrowserDocumentCompletedEventArgs e )
76 {
77     // establece el texto en navegacionTextBox al URL de la página actual
78     navegacionTextBox.Text = webBrowser.Url.ToString();
79
80     // habilita o deshabilita los controles atrasButton y adelanteButton
81     atrasButton.Enabled = webBrowser.CanGoBack;
82     adelanteButton.Enabled = webBrowser.CanGoForward;
83
84     // deshabilita detenerButton
85     detenerButton.Enabled = false;
86
87     // borra el control paginaProgressBar
88     paginaProgressBar.Value = 0;
89 } // fin del método webBrowser_DocumentCompleted
90
91 // actualiza el título del Explorador
92 private void webBrowser_DocumentTitleChanged( object sender,
93     EventArgs e )
94 {
95     this.Text = webBrowser.DocumentTitle + " - Explorador";
96 } // fin del método webBrowser_DocumentTitleChanged
97 } // fin de la clase ExploradorForm

```

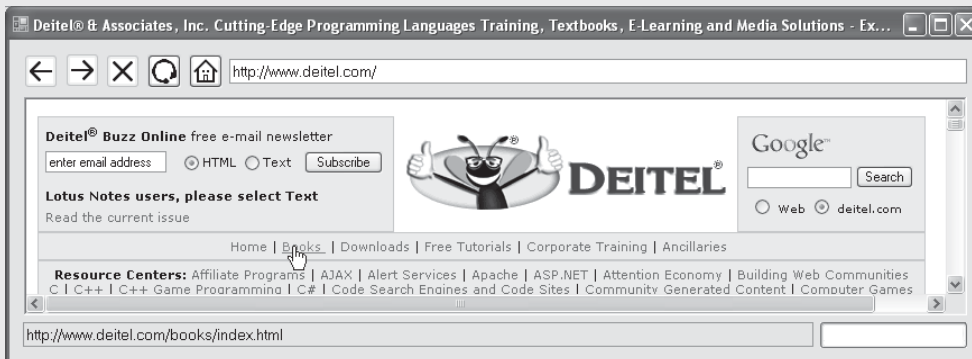


Figura 23.8 | Ejemplo del control WebBrowser. (Parte 2 de 2).

Las líneas 14-41 definen cinco manejadores de eventos Click, uno para cada uno de los cinco controles Button de navegación que aparecen en la parte superior del formulario. Cada manejador de eventos llama a su método correspondiente de WebBrowser. El método **GoBack** de WebBrowser (línea 16) hace que el control navegue de vuelta a la página anterior en el historial de navegación. El método **GoForward** (línea 22) hace que

el control navegue hacia delante a la siguiente página en el historial de navegación. El método **Stop** (línea 28) hace que el control deje de cargar la página actual. El método **Refresh** (línea 34) hace que el control vuelva a cargar la página actual. El método **GoHome** (línea 40) hace que el control navegue a la página inicial del usuario, según esté definida en las opciones de Internet Explorer (bajo **Herramientas > Opciones de Internet...** en la sección **Página de inicio**).

El control `TextBox` a la derecha de los botones de navegación permite al usuario escribir el URL de un sitio Web para explorarlo. Cuando el usuario escribe cada pulsación de tecla en el control `TextBox`, se ejecuta el manejador de eventos en las líneas 44-49. Si la tecla oprimida es *Intro*, la línea 48 llama al método **Navigate** de `WebBrowser` para recuperar el documento en el URL especificado.

El control `WebBrowser` genera un evento **Navigating** cuando empieza a cargar una nueva página. Cuando esto ocurre, se ejecuta el manejador de eventos en las líneas 52-56 y la línea 55 habilita el control `detenerBoton`, para que el usuario pueda cancelar la carga de la página Web.

Por lo general, un usuario puede ver el estado del proceso de carga de una página Web en la parte inferior de la ventana del explorador. Por esta razón, incluimos un control `TextBox` (llamado `estadoTextBox`) y un control `ProgressBar` (llamado `paginaProgressBar`) en la parte inferior de nuestro formulario. El control `WebBrowser` genera un evento **StatusTextChanged** cuando cambia la propiedad **StatusText** de `WebBrowser`. El manejador de eventos para este evento (líneas 59-63) asigna el nuevo contenido de la propiedad `StatusText` del control a la propiedad `Text` de `estadoTextBox` (línea 62), para que el usuario pueda supervisar los mensajes de estado de `WebBrowser`. El control genera un evento **ProgressChanged** cuando se actualiza el progreso de carga de la página en el control `WebBrowser`. El manejador de eventos `ProgressChanged` (líneas 66-71) actualiza la propiedad `Value` de `paginaProgressBar` (líneas 69-70), para reflejar cuánto porcentaje del documento actual se ha cargado.

Cuando el control `WebBrowser` termina de cargar un documento, genera un evento **DocumentCompleted**. Esto ejecuta el manejador de eventos en las líneas 74-89. La línea 78 actualiza el contenido de `navegacionTextBox`, de manera que muestre el URL de la página actual que está cargada (la propiedad `Url` de `WebBrowser`). Esto es muy importante si el usuario navega a otra página Web haciendo clic en un vínculo de la página existente. Las líneas 81-82 utilizan las propiedades **CanGoBack** y **CanGoForward** para determinar si los botones atrás y adelante deben habilitarse o deshabilitarse. Como ahora el documento está cargado, la línea 85 deshabilita el control `detenerButton`. La línea 88 establece la propiedad `Value` de `paginaProgressBar` a 0, para indicar que no se está cargando ningún contenido en ese momento.

Las líneas 92-96 definen un manejador de eventos para el evento **DocumentTitleChanged**, el cual se produce cuando se carga un nuevo documento en el control `WebBrowser`. La línea 95 establece la propiedad `Text` de `ExploradorForm` (que se muestra en la barra de título del formulario) con base en el título del documento actual (**DocumentTitle**) de `WebBrowser`.

23.10 .NET Remoting

El .NET Framework proporciona una tecnología de computación distribuida llamada *.NET Remoting*, la cual permite a un programa acceder a los objetos en otra máquina, a través de una red. El término .NET Remoting es similar en concepto a RMI (invocación de métodos remotos) en Java y RPC (llamada a procedimientos remotos) en los lenguajes de programación por procedimientos. La tecnología .NET Remoting es también similar a los servicios Web (capítulo 22), con unas cuantas diferencias clave. Con los servicios Web, una aplicación cliente se comunica con un servicio Web hospedado por un servidor Web. El cliente y el servicio Web pueden estar escritos en cualquier lenguaje, siempre y cuando puedan transmitir mensajes en SOAP. Con .NET Remoting, una aplicación cliente se comunica con una aplicación servidor, y tanto el cliente como el servidor deben estar escritos en lenguajes .NET. Mediante el uso de .NET Remoting, un cliente y un servidor pueden comunicarse a través de llamadas a métodos y pueden transmitirse objetos entre las aplicaciones; a este proceso se le conoce como *cálculo de referencias (marshaling)* de los objetos.

Canales

El cliente y el servidor pueden comunicarse entre sí a través de *canales*. Por lo general, los canales utilizan el protocolo HTTP o TCP para transmitir mensajes. La ventaja de un *canal HTTP* es que comúnmente los firewalls permiten el uso de conexiones HTTP de manera predeterminada, mientras que generalmente bloquean las conexiones TCP desconocidas. La ventaja de un *canal TCP* es que tiene un mejor rendimiento que un canal HTTP. En una

aplicación de .NET Remoting, el cliente y el servidor crean un canal cada uno, y ambos canales deben usar el mismo protocolo para comunicarse entre sí. En nuestro ejemplo, utilizamos canales HTTP.

Cálculo de referencias (Marshaling)

Existen dos maneras de calcular las referencias de un objeto: por valor y por referencia. El *cálculo de referencias por valor* requiere que el objeto sea *serializable*; esto es, que sea capaz de representarse como un mensaje con formato, que pueda enviarse de una aplicación a otra a través de un canal. El extremo receptor del canal *deserializa* el objeto para obtener una copia del objeto original. Para permitir la serialización y deserialización de un objeto, su clase debe declararse con el atributo [`Serializable`] o debe implementar la interfaz `ISerializable`.

El *cálculo de referencias por referencia* requiere que la clase del objeto extienda a la clase `MarshalByRefObject` del espacio de nombres `System`. Un objeto cuyas referencias se calculan por referencia se conoce como *objeto remoto*, y su clase como *clase remota*. Cuando se calculan las referencias de un objeto por referencia, el objeto en sí no se transmite. En vez de ello, se crean dos objetos *proxy*: un *proxy transparente* y un *proxy real*. El proxy transparente proporciona todos los servicios públicos de un objeto remoto. Por lo general, un cliente llama a los métodos y las propiedades del proxy transparente como si fuera el objeto remoto. Después, el proxy transparente llama al método `Invoke` del proxy real. Esto envía el mensaje apropiado del canal del cliente al canal del servidor. El servidor recibe este mensaje y lleva a cabo la llamada al método especificado, o accede a la propiedad especificada en el objeto actual, que reside en el servidor. En nuestro ejemplo, calculamos las referencias de un objeto remoto por referencia.

Aplicación de información del clima mediante el uso de .NET Remoting

Ahora presentaremos un ejemplo de .NET Remoting que descarga la información sobre el clima *Traveler's Forecast* del sitio Web del Servicio meteorológico nacional:

`http://iwin.nws.noaa.gov/iwin/us/traveler.html`

[Nota: Cuando desarrollamos este ejemplo, el Servicio meteorológico nacional indicó que la información que se proporciona en la página Web *Traveler's Forecast* se proporcionará mediante un servicio Web en un futuro cercano. La información que utilizamos en este ejemplo depende directamente del formato de la página Web *Traveler's Forecast*. Si tiene problemas al ejecutar este ejemplo, consulte la página de FAQs en `www.deitel.com/faq.html`. Este problema potencial demuestra un beneficio en cuanto al uso de los servicios Web o .NET Remoting para implementar aplicaciones de computación distribuidas que pueden cambiar en un futuro. La acción de separar la parte del servidor de la aplicación, que depende del formato de un origen de datos externo, de la parte del cliente de la aplicación, permite actualizar la implementación del servidor sin requerir cambios en el cliente.]

Nuestra aplicación de .NET Remoting consiste en cinco componentes:

1. La clase serializable `ClimaCiudad`, que representa el reporte del clima para una ciudad.
2. La interfaz `Reporte`, que declara una propiedad `Reportes` que el objeto cuyas referencias se calcularon proporciona a una aplicación cliente, para obtener una colección de objetos `ClimaCiudad`.
3. La clase remota `InfoReporte`, que extiende a la clase `MarshalByRefObject`, implementa a la interfaz `Reporte` y se instanciará sólo en el servidor.
4. Una aplicación `ServidorClima` que establece un canal servidor y hace que la clase `InfoReporte` esté disponible en un URI (Identificador uniforme de recursos) específico.
5. Una aplicación `ClienteClima` que establece un canal cliente y solicita un objeto `InfoReporte` del `ServidorClima` para obtener el reporte del clima del día.

Clase ClimaCiudad

La clase `ClimaCiudad` (figura 23.9) contiene información meteorológica para una ciudad. La clase `ClimaCiudad` se declara en el espacio de nombres `Clima` (línea 5) para que pueda reutilizarse, y se publicará en el archivo de biblioteca de clases `Clima.dll`, al cual deben hacer referencia las aplicaciones cliente y servidor. Por esta razón, es conveniente colocar esta clase (y la interfaz `Reporte` de la figura 23.10) en un proyecto de biblioteca de clases. La clase `ClimaCiudad` se declara con el atributo `Serializable` (línea 7), el cual indica que se pueden calcular las

referencias de un objeto de la clase `ClimaCiudad` por valor. Esto es necesario, ya que la propiedad `Reportes` del objeto `InfoReportes` devolverá los objetos `ClimaCiudad`, y deben calcularse las referencias por valor de los mismos valores de retorno de los métodos y las propiedades declarados por una clase remota, del servidor al cliente. (Las referencias a los valores de los argumentos en las llamadas a los métodos también se calculan por valor del cliente al servidor.) Así, cuando el cliente llama a un descriptor de acceso `get` que devuelve objetos `ClimaCiudad`, el canal servidor serializará los objetos `ClimaCiudad` en un mensaje que el canal cliente pueda deserializar, para crear copias de los objetos `ClimaCiudad` originales. La clase `ClimaCiudad` también implementa la interfaz `IComparable` (línea 8), de manera que pueda ordenarse alfabéticamente un arreglo `ArrayList` de objetos `ClimaCiudad`.

```

1  // Fig. 23.9: ClimaCiudad.cs
2  // Clase que representa la información del clima para una ciudad.
3  using System;
4
5  namespace Clima
6  {
7      [ Serializable ]
8      public class ClimaCiudad : IComparable
9      {
10         private string nombreCiudad;
11         private string descripcion;
12         private string temperatura;
13
14         public ClimaCiudad( string ciudad, string informacion,
15                             string grados )
16         {
17             nombreCiudad = ciudad;
18             descripcion = informacion;
19             temperatura = grados;
20         } // fin del constructor
21
22         // propiedad de sólo lectura que obtiene el nombre de la ciudad
23         public string NombreCiudad
24         {
25             get
26             {
27                 return nombreCiudad;
28             } // fin de get
29         } // fin de la propiedad NombreCiudad
30
31         // propiedad de sólo lectura que obtiene la descripción del clima de la ciudad
32         public string Descripcion
33         {
34             get
35             {
36                 return descripcion;
37             } // fin de get
38         } // fin de la propiedad Descripcion
39
40         // propiedad de sólo lectura que obtiene la temperatura de la ciudad
41         public string Temperatura
42         {
43             get
44             {
45                 return temperatura;
46             } // fin de get

```

Figura 23.9 | La clase `ClimaCiudad`. (Parte I de 2).

```

47     } // fin de la propiedad Temperatura
48
49     // implementación del método CompareTo para alfabeticar
50     public int CompareTo( object other )
51     {
52         return string.Compare(
53             NombreCiudad, ( ( ClimaCiudad ) other ).NombreCiudad );
54     } // fin del método Compare
55
56     // devuelve representación string de este objeto ClimaCiudad
57     // (utilizado para mostrar el reporte del clima en la consola del servidor)
58     public override string ToString()
59     {
60         return nombreCiudad + " | " + temperatura + " | " + descripcion;
61     } // fin del método ToString
62 } // fin de la clase ClimaCiudad
63 } // fin del espacio de nombres Clima

```

Figura 23.9 | La clase ClimaCiudad. (Parte 2 de 2).

ClimaCiudad contiene tres variables de instancia (líneas 10-12) para almacenar el nombre de la ciudad, las temperaturas máxima/mínima y la descripción de la condición del tiempo. El constructor de ClimaCiudad (líneas 14-20) inicializa las tres variables de instancia. Las líneas 23-47 declaran tres propiedades de sólo lectura, que permiten obtener los valores de las tres variables de instancia. ClimaCiudad implementa a IComparable, por lo que debe declarar un método llamado CompareTo que recibe una referencia object y devuelve un int (líneas 50-54). Además, queremos alfabeticar los objetos ClimaCiudad con base en los nombres de sus ciudades (nombreCiudad), por lo que CompareTo llama al método string Compare con los nombres de las ciudades de los dos objetos ClimaCiudad. La clase ClimaCiudad también redefine el método ToString para mostrar información para esta ciudad (líneas 58-61). La aplicación servidor utiliza el método ToString para mostrar la información del clima obtenida de la página Web *Traveler's Forecast* en la consola.

Interfaz Reporte

La figura 23.10 muestra el código para la interfaz Reporte. Esta interfaz también se declara en el espacio de nombres Clima (línea 7) y se incluye con la clase ClimaCiudad en el archivo de biblioteca de clases Clima.dll, por lo que puede usarse tanto en la aplicación cliente como en la aplicación servidor. Reporte declara una propiedad de sólo lectura (líneas 11-14), con un descriptor de acceso get que devuelve un arreglo ArrayList de objetos

```

1  // Fig. 23.10: Reporte.cs
2  // Interfaz que define una propiedad para obtener
3  // la información en un reporte meteorológico.
4  using System;
5  using System.Collections;
6
7  namespace Clima
8  {
9      public interface Reporte
10     {
11         ArrayList Reportes
12         {
13             get;
14         } // fin de la propiedad Reportes
15     } // fin de la interfaz Reporte
16 } // fin del espacio de nombres Clima

```

Figura 23.10 | La interfaz Reporte en el espacio de nombres Clima.

ClimaCiudad. La aplicación cliente utiliza esta propiedad para obtener la información en el reporte del clima: el nombre de cada ciudad, la temperatura máxima/mínima y la condición climatológica.

Clase InfoReporte

La clase remota InfoReporte (figura 23.11) implementa a la interfaz Reporte (línea 10) del espacio de nombres Clima (especificado por la directiva using en la línea 8). InfoReporte también extiende a la clase base MarshalByRefObject. La clase InfoReporte es parte de la aplicación remota ServidorClima y no está directamente disponible para la aplicación cliente.

```

1  // Fig. 23.11: InfoReporte.cs
2  // Clase que implementa a la interfaz Reporte, obtiene
3  // y devuelve datos acerca del clima
4  using System;
5  using System.Collections;
6  using System.IO;
7  using System.Net;
8  using Clima;
9
10 public class InfoReporte : MarshalByRefObject, Reporte
11 {
12     private ArrayList listaCiudades; // ciudades, temperaturas, descripciones
13
14     public InfoReporte()
15     {
16         listaCiudades = new ArrayList();
17
18         // crea cliente Web para obtener acceso a la página Web
19         WebClient miCliente = new WebClient();
20
21         // obtiene StreamReader de respuesta, para poder leer la página
22         StreamReader entrada = new StreamReader( miCliente.OpenRead(
23             "http://iwin.nws.noaa.gov/iwin/us/traveler.html" ) );
24
25         string separador1 = "TAV12"; // indica el primer lote de ciudades
26         string separador2 = "TAV13"; // indica el segundo lote de ciudades
27
28         // localiza separador1 en la página Web
29         while ( !entrada.ReadLine().StartsWith( separador1 ) ); // no hace nada
30         LeerCiudades( entrada ); // lee el primer lote de ciudades
31
32         // localiza separador2 en la página Web
33         while ( !entrada.ReadLine().StartsWith( separador2 ) ); // no hace nada
34         LeerCiudades( entrada ); // lee el segundo lote de ciudades
35
36         listaCiudades.Sort(); // ordena alfabéticamente la lista de ciudades
37         entrada.Close(); // cierra StreamReader al servidor NWS
38
39         // muestra los datos en el lado servidor
40         Console.WriteLine( "Datos del sitio Web NWS:" );
41
42         foreach ( ClimaCiudad ciudad in listaCiudades )
43         {
44             Console.WriteLine( ciudad );
45         } // fin de foreach
46     } // fin del constructor

```

Figura 23.11 | El cálculo de las referencias en la clase InfoReporte, que implementa a la interfaz Reporte, es por referencia. (Parte 1 de 2).

```

47
48 // método utilitario que lee un lote de ciudades
49 private void LeerCiudades( StreamReader entrada )
50 {
51     // formato de día y formato de noche
52     string formatoDia =
53         "CITY          WEA      MAX/MIN   WEA      MAX/MIN";
54     string formatoNoche =
55         "CITY          WEA      MIN/MAX   WEA      MIN/MAX";
56     string lineaEntrada = "";
57
58     // localiza el encabezado en donde empieza la información meteorológica
59     do
60     {
61         lineaEntrada = entrada.ReadLine();
62     } while ( !lineaEntrada.Equals( formatoDia ) &&
63             !lineaEntrada.Equals( formatoNoche ) );
64
65     lineaEntrada = entrada.ReadLine(); // obtiene los datos de la primera ciudad
66
67     // mientras haya más ciudades que leer
68     while ( lineaEntrada.Length > 28 )
69     {
70         // crea objeto ClimaCiudad para la ciudad
71         ClimaCiudad clima = new ClimaCiudad(
72             lineaEntrada.Substring( 0, 16 ),
73             lineaEntrada.Substring( 16, 7 ),
74             lineaEntrada.Substring( 23, 7 ) );
75
76         listaCiudades.Add( clima ); // lo agrega a ArrayList
77         lineaEntrada = entrada.ReadLine(); // obtiene los datos de la siguiente ciudad
78     } // fin de while
79 } // fin del método LeerCiudades
80
81 // propiedad para obtener los reportes meteorológicos de las ciudades
82 public ArrayList Reportes
83 {
84     get
85     {
86         return listaCiudades;
87     } // fin de get
88 } // fin de la propiedad Reportes
89 } // fin de la clase InfoReporte

```

Figura 23.11 | El cálculo de las referencias en la clase InfoReporte, que implementa a la interfaz Reporte, es por referencia. (Parte 2 de 2).

Las líneas 14-46 declaran el constructor InfoReporte. La línea 19 crea un objeto **WebClient** (espacio de nombres System.Net) para interactuar con un origen de datos especificado por un URL; en este caso, el URL para la página *Traveler's Forecast* del NWS (<http://iwin.nws.noaa.gov/iwin/us/traveler.html>). Las líneas 22-23 llaman al método OpenRead de WebClient, el cual devuelve un objeto Stream que el programa puede usar para leer datos que contengan la información climatológica del URL especificado. Este objeto Stream se utiliza para crear un objeto StreamReader, por lo que el programa puede leer el marcado HTML de la página Web, línea por línea.

La sección de la página Web en la que estamos interesados consiste en dos lotes de ciudades: de Albany hasta Reno, y de Salt Lake City hasta Washington, D.C. El primer lote ocurre en una sección que empieza con la cadena "TAV12", mientras que el segundo lote ocurre en una sección que empieza con la cadena "TAV13". Declaramos las variables `separador1` y `separador2` para que almacenen estas cadenas. La línea 29 lee el marcado de

HTML una línea a la vez, hasta encontrar "TAV12". Una vez que se llega a "TAV12", el programa llama al método utilitario LeerCiudades para leer un lote de ciudades en el objeto ArrayList listaCiudades. A continuación, la línea 33 lee el marcado de HTML una línea a la vez, hasta encontrar "TAV13", y la línea 34 hace otra llamada al método LeerCiudades para leer el segundo lote de ciudades. La línea 36 llama al método **Sort** de la clase ArrayList para ordenar los objetos ClimaCiudad en forma alfabética, por nombre de ciudad. La línea 37 cierra la conexión de StreamReader con el sitio Web. Las líneas 42-45 imprimen en pantalla la información del clima para cada ciudad, en la pantalla de la consola de la aplicación servidor.

Las líneas 49-79 declaran el método utilitario LeerCiudades, el cual recibe un objeto StreamReader y lee la información para cada ciudad, crea un objeto ClimaCiudad para esto y coloca el objeto ClimaCiudad en listaCiudades. La instrucción do...while (líneas 59-63) continúa leyendo la página, una línea a la vez, hasta encontrar la línea del encabezado que empieza la tabla de pronósticos del clima. Esta línea empieza con formato-Dia (líneas 52-53), indicando el encabezado para la información en horas diurnas, o con formatoNoche (líneas 54-55), indicando el encabezado para la información en horas nocturnas. Debido a que la línea podría estar en cualquier formato con base en la hora del día, la condición de continuación de ciclo comprueba ambas opciones. La línea 65 lee la siguiente línea de la página Web, que es la primera línea que contiene información sobre la temperatura.

La instrucción while (líneas 68-78) crea un nuevo objeto ClimaCiudad para representar a la ciudad actual. Analiza el objeto string que contiene los datos actuales del clima, separando el nombre de la ciudad, la condición climatológica y la temperatura. El objeto ClimaCiudad se agrega a listaCiudades. Después se lee la siguiente línea de la página y se almacena en lineaEntrada para la siguiente iteración. Este proceso continúa mientras la longitud del objeto string que se lee de la página Web sea mayor de 28 (las líneas que contienen los datos del clima son mayores de 28 caracteres). La primera línea menor que esta cantidad indica el final de la sección de pronósticos en la página Web.

La propiedad de sólo lectura Reportes (líneas 82-88) implementa la propiedad Reportes de la interfaz Reporte para devolver listaCiudades. La aplicación cliente llamará en forma remota a esta propiedad para obtener el reporte climatológico del día.

Clase ServidorClima

La figura 23.12 contiene el código para la aplicación servidor. Las directivas using en las líneas 5-7 especifican los espacios de nombres de .NET Remoting **System.Runtime.Remoting**, **System.Runtime.Remoting.Channels** y **System.Runtime.Remoting.Channels.Http**. Los primeros dos espacios de nombres son requeridos para .NET Remoting, y el tercero es requerido para los canales HTTP. El espacio de nombres System.Runtime.Remoting.Channels.Http requiere que el proyecto haga referencia al ensamblado System.Runtime.Remoting, el cual puede encontrarse bajo la ficha .NET en el menú **Agregar referencias**. La directiva using en la línea 8 especifica el espacio de nombres Clima, que contiene la interfaz Reporte. Recuerde agregar una referencia a Clima.dll en este proyecto.

```

1  // Fig. 23.12: ServidorClima.cs
2  // Aplicación servidor que utiliza .NET Remoting para enviar
3  // la información de un reporte meteorológico a un cliente
4  using System;
5  using System.Runtime.Remoting;
6  using System.Runtime.Remoting.Channels;
7  using System.Runtime.Remoting.Channels.Http;
8  using Clima;
9
10 class ServidorClima
11 {
12     static void Main( string[] args )
13     {
14         // establece canal HTTP
15         HttpChannel canal = new HttpChannel( 50000 );

```

Figura 23.12 | La clase ServidorClima expone a la clase remota InfoReporte. (Parte 1 de 2).


```

16     ChannelServices.RegisterChannel( canal, false );
17
18     // registra la clase InfoReporte
19     RemotingConfiguration.RegisterWellKnownServiceType(
20         typeof( InfoReporte ), "Reporte",
21         WellKnownObjectMode.Singleton );
22
23     Console.WriteLine( "Oprima Intro para cerrar el servidor." );
24     Console.ReadLine();
25 } // fin de Main
26 } // fin de la clase ServidorClima

```

Figura 23.12 | La clase ServidorClima expone a la clase remota InfoReporte. (Parte 2 de 2).

Las líneas 15-16 en Main registran un canal HTTP en el equipo actual, en el puerto 50000. Éste es el número de puerto que los clientes utilizarán para conectarse al ServidorClima en forma remota. El argumento *false* en la línea 16 indica que no deseamos habilitar la seguridad, lo cual está más allá del alcance de esta introducción. Las líneas 19-21 registran el tipo de la clase InfoReporte en el URI “Reporte” como una clase remota Singleton. Si una clase remota se registra como **Singleton**, se creará un objeto remoto cuando el primer cliente solicite esa clase remota, y ese objeto remoto dará servicio a todos los clientes. El modo alternativo es **SingleCall**, en donde se crea un objeto remoto para cada llamada individual al método remoto en la clase remota. [Nota: Un objeto remoto Singleton no tiene un tiempo de vida infinito; será candidato a la recolección de basura después de estar inactivo durante 5 minutos. Se creará un nuevo objeto remoto Singleton si otro cliente solicita uno posteriormente.] La clase remota InfoReporte está ahora disponible para los clientes en el URI “http://DireccionIP: 50000/Reporte”, en donde *DireccionIP* es la dirección IP de la computadora en la que se está ejecutando el servidor. El canal permanece abierto mientras que la aplicación siga ejecutándose, por lo que la línea 24 espera a que el usuario que ejecuta la aplicación servidor oprima Intro antes de terminar la aplicación.

Clase ClienteClimaForm

La clase ClienteClimaForm (figura 23.13) es una aplicación Windows que utiliza .NET Remoting para obtener información climatológica de ServidorClima, y muestra la información en forma gráfica, fácil de leer. La GUI contiene 43 controles Label; uno para cada ciudad en la página Web *Traveler's Forecast*. Todos los controles Label están colocados en un Panel con una barra de desplazamiento vertical. Cada control Label muestra la información del clima para una ciudad. Las líneas 7-9 son directivas using para los espacios de nombres requeridos para utilizar .NET Remoting. Para este proyecto, debe agregar referencias al ensamblado System.Runtime.Remoting y al archivo Clima.dll que creamos antes.

```

1  // Fig. 23.13: ClienteClima.cs
2  // Cliente que utiliza .NET Remoting para obtener un reporte meteorológico.
3  using System;
4  using System.Collections;
5  using System.Drawing;
6  using System.Windows.Forms;
7  using System.Runtime.Remoting;
8  using System.Runtime.Remoting.Channels;
9  using System.Runtime.Remoting.Channels.Http;
10 using Clima;
11
12 public partial class ClienteClimaForm : Form
13 {
14     public ClienteClimaForm()

```

Figura 23.13 | La aplicación ClienteClima accede a un objeto InfoReporte en forma remota y muestra el reporte climatológico. (Parte 1 de 3).

```

15     {
16         InitializeComponent();
17     } // fin del constructor
18
19     // obtiene los datos del clima
20     private void ClienteClimaForm_Load( object sender, EventArgs e )
21     {
22         // establece canal HTTP, no necesita proporcionar un número de puerto
23         HttpChannel canal = new HttpChannel();
24         ChannelServices.RegisterChannel( canal, false );
25
26         // obtiene un proxy para un objeto que implementa a la interfaz Reporte
27         Reporte info = ( Reporte ) RemotingServices.Connect(
28             typeof( Reporte ), "http://localhost:50000/Reporte" );
29
30         // obtiene un ArrayList de objetos ClimaCiudad
31         ArrayList ciudades = info.Reportes;
32
33         // crea arreglo y lo llena con todos los controles Label
34         Label[] etiquetasCiudades = new Label[ 43 ];
35         int contadorEtiquetas = 0;
36
37         foreach ( Control control in mostrarPanel.Controls )
38         {
39             if ( control is Label )
40             {
41                 etiquetasCiudades[ contadorEtiquetas ] = ( Label ) control;
42                 ++contadorEtiquetas; // incrementa contador de etiquetas
43             } // fin de if
44         } // fin de foreach
45
46         // crea Hashtable y la llena con todas las condiciones del clima
47         Hashtable clima = new Hashtable();
48         clima.Add( "SUNNY", "sunny" );
49         clima.Add( "PTCLDY", "pcloudy" );
50         clima.Add( "CLOUDY", "mcloudy" );
51         clima.Add( "MOCLDY", "mcloudy" );
52         clima.Add( "TSTRMS", "rain" );
53         clima.Add( "RAIN", "rain" );
54         clima.Add( "SNOW", "snow" );
55         clima.Add( "VRYHOT", "vryhot" );
56         clima.Add( "FAIR", "fair" );
57         clima.Add( "RNSNOW", "rnsnow" );
58         clima.Add( "SHWRS", "showers" );
59         clima.Add( "WINDY", "windy" );
60         clima.Add( "NOINFO", "noinfo" );
61         clima.Add( "MISG", "noinfo" );
62         clima.Add( "DRZL", "rain" );
63         clima.Add( "HAZE", "noinfo" );
64         clima.Add( "SMOKE", "mcloudy" );
65         clima.Add( "SNOWSHWRS", "snow" );
66         clima.Add( "FLRRYS", "snow" );
67         clima.Add( "FOG", "noinfo" );
68
69         // crea la fuente para la salida de texto
70         Font fuente = new Font( "Courier New", 8, FontStyle.Bold );
71
72         // para cada ciudad

```

Figura 23.13 | La aplicación ClienteClima accede a un objeto InfoReporte en forma remota y muestra el reporte climatológico. (Parte 2 de 3).

```

73     for ( int i = 0; i < ciudades.Count; i++ )
74     {
75         // usa el arreglo etiquetasCiudades para buscar la siguiente etiqueta
76         Label ciudadActual = etiquetasCiudades[ i ];
77
78         // usa el objeto ArrayList ciudades para buscar el siguiente objeto ClimaCiudad
79         ClimaCiudad ciudad = ( ClimaCiudad ) ciudades[ i ];
80
81         // establece la imagen de la etiqueta actual a la imagen
82         // que corresponde a la condición climatológica de la ciudad
83         // busca el nombre de imagen correcto en Hashtable clima
84         ciudadActual.Image = new Bitmap( @"images\" +
85             clima[ ciudad.Descripcion.Trim() ] + ".png" );
86         ciudadActual.Font = fuente; // establece la fuente de la etiqueta
87         ciudadActual.ForeColor = Color.White; // establece color de texto de la etiqueta
88
89         // establece texto de etiqueta al nombre de ciudad y temperatura
90         ciudadActual.Text = "\r\n " + ciudad.NombreCiudad + ciudad.Temperatura;
91     } // fin de for
92 } // fin del método ClienteClimaForm_Load
93 } // fin de la clase ClienteClimaForm

```

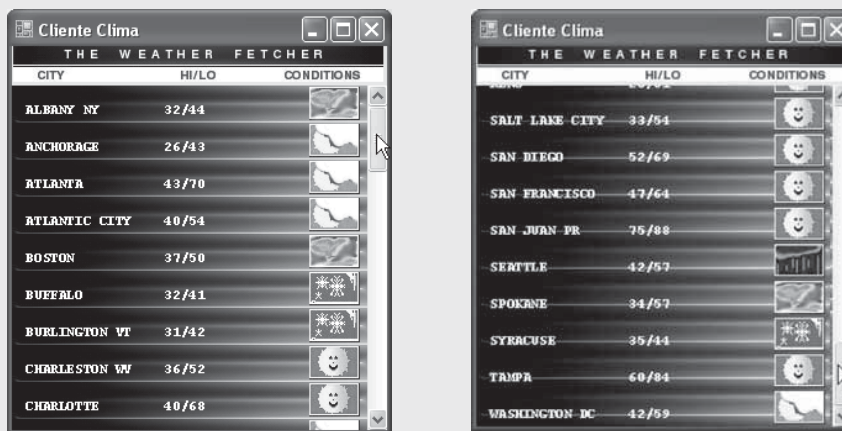


Figura 23.13 | La aplicación *ClienteClima* accede a un objeto *InfoReporte* en forma remota y muestra el reporte climatológico. (Parte 3 de 3).

El método *ClienteClimaForm_Load* (líneas 20-92) obtiene la información del clima cuando se carga esta aplicación Windows. La línea 23 crea un canal HTTP sin especificar un número de puerto. Esto hace que el constructor de *HttpChannel* seleccione cualquier número de puerto disponible en la computadora cliente. No es necesario un número de puerto específico, ya que esta aplicación no tiene sus propios clientes que necesitan conocer el número de puerto de antemano. La línea 24 registra el canal en la computadora cliente. Esto permite que el servidor envíe información de vuelta al cliente. Las líneas 27-28 declaran una variable *Reporte* y le asignan un proxy para un objeto *Reporte* instanciado por el servidor. Este proxy permite que el cliente llame en forma remota a las propiedades de *InfoReporte*, redirigiendo al servidor las llamadas a los métodos. El método **Connect** de **RemotingServices** se conecta al servidor y devuelve una referencia al proxy para el objeto *Reporte*. Tenga en cuenta que estamos ejecutando el cliente y el servidor en la misma computadora, por lo que usamos *localhost* en el URL que representa a la aplicación servidor. Para conectarse a un *ServidorClima* en una computadora distinta, debe modificar este URL de manera acorde. La línea 31 obtiene el arreglo *ArrayList* de los objetos *ClimaCiudad* generados por el constructor de *InfoReporte* (líneas 14-46 de la figura 23.11). La variable *ciudades* ahora hace

referencia a un arreglo `ArrayList` de objetos `ClimaCiudad` que contienen la información obtenida de la página Web *Traveler's Forecast*.

Debido a que la aplicación presenta los datos climatológicos para demasiadas ciudades, debemos establecer una manera de organizar la información en los controles `Label` y asegurar que cada descripción del clima esté acompañada por una imagen apropiada. El programa usa un arreglo para almacenar todos los controles `Label` y un objeto `Hashtable` (del cual hablaremos más en el capítulo 26, Colecciones) para almacenar las descripciones climatológicas y los nombres de sus imágenes correspondientes. Un objeto ***Hashtable*** almacena pares clave-valor, en los que tanto la clave como el valor pueden ser cualquier tipo de objeto. El método ***Add*** agrega pares clave-valor a un objeto `Hashtable`. La clase también proporciona un indexador para regresar el valor para una clave particular en el objeto `Hashtable`. La línea 34 crea un arreglo de referencias `Label`, y las líneas 35-44 colocan en el arreglo los controles `Label` que creamos en el diseñador, para que pueda accederse a ellos mediante la programación, para mostrar información climatológica para ciudades individuales. La línea 47 crea el objeto `Hashtable` `clima` para almacenar pares de condiciones climatológicas y los nombres para las imágenes asociadas con esas condiciones. Observe que el nombre de una descripción climatológica dada no necesariamente corresponde al nombre del archivo PNG que contiene la imagen correcta. Por ejemplo, las condiciones “TSTRMS” y “RAIN” utilizan el archivo de imagen `rain.png`.

Las líneas 73-91 establecen cada control `Label`, de manera que contenga el nombre de una ciudad, la temperatura actual y una imagen correspondiente a las condiciones climatológicas para esa ciudad. La línea 76 obtiene el control `Label` que mostrará la información climatológica para la siguiente ciudad. La línea 79 utiliza el arreglo `ArrayList` llamado `ciudades` para obtener el objeto `ClimaCiudad` que contiene la información climatológica para la ciudad. Las líneas 84-85 establecen la imagen del control `Label` con base en la imagen PNG que corresponde a las condiciones climatológicas de la ciudad. Esto se hace eliminando cualquier espacio en la cadena de descripción, mediante una llamada al método `string Trim` y se obtiene el nombre de la imagen PNG del objeto `Hashtable` `clima`. Las líneas 86-87 establecen las propiedades de `Label` para lograr el efecto visual que se puede ver en las pantallas de resultados de ejemplo. La línea 90 establece la propiedad `Text` para mostrar el nombre de la ciudad y las temperaturas máxima/mínima. [Nota: Para preservar la distribución de la ventana de la aplicación cliente, establecemos las propiedades `MaximumSize` y `MinimumSize` del formulario Windows Forms al mismo valor, para que el usuario no pueda cambiar el tamaño de la ventana.]

Recursos Web para .NET Remoting

En esta sección vimos una introducción básica a la tecnología .NET Remoting. Hay mucho más que ver en cuanto a esta poderosa herramienta del .NET Framework. Los siguientes sitios Web proporcionan información adicional para los lectores que deseen investigar más sobre estas herramientas. Además, si busca la frase “.NET Remoting” en la mayoría de los motores de búsqueda, podrá obtener muchos recursos adicionales.

msdn.microsoft.com/library/en-us/cpguide/html/cpconaccessingobjectsinothertapplication-domainusingnetremoting.asp

La *Guía del desarrollador de .NET Framework* en el sitio Web de MSDN proporciona información detallada acerca de .NET Remoting, incluyendo artículos que hablan sobre la elección entre ASP.NET y .NET Remoting, las generalidades acerca de .NET Remoting, técnicas avanzadas de .NET Remoting y ejemplos.

msdn.microsoft.com/library/en-us/dndotnet/html/introremoting.asp

Este sitio ofrece una visión general sobre las herramientas de .NET Remoting.

search.microsoft.com/search/results.aspx?qu=.net+remoting

Esta búsqueda en `microsoft.com` proporciona vínculos a muchos artículos y recursos sobre .NET Remoting.

23.11 Conclusión

En este capítulo presentamos las técnicas de redes orientadas a las conexiones y sin conexión. Aprendió que Internet es una red “desconfiable” que simplemente transmite paquetes de datos. Hablamos sobre dos protocolos para transmitir paquetes a través de Internet: el Protocolo de control de transmisión (TCP) y el Protocolo de datagramas de usuario (UDP). Aprendió que TCP es un protocolo de comunicación orientado a la conexión, el cual garantiza que los paquetes enviados lleguen al receptor destinado sin daños y en la secuencia correcta. También aprendió que UDP se utiliza por lo general en aplicaciones orientadas al rendimiento, ya que incurre en una mínima sobrecarga para la comunicación entre aplicaciones. Presentamos algunas de las herramientas de C# para implementar comunicaciones

con TCP y UDP. Le mostramos cómo crear una aplicación de chat cliente/servidor simple, mediante el uso de sockets de flujo. Después le mostramos cómo enviar datagramas entre un cliente y un servidor. También vio un servidor de Tres en raya con subprocesamiento múltiple, el cual permite que dos clientes se conecten en forma simultánea al servidor para jugar Tres en raya, uno contra el otro. Le presentamos el nuevo control `WebBrowser`, que le permite agregar herramientas de exploración Web a sus aplicaciones Windows. Por último, demostramos la tecnología .NET Remoting, la cual permite que una aplicación cliente acceda en forma remota a las propiedades y métodos de un objeto instanciado por una aplicación servidor. En el capítulo 24, Estructuras de datos, aprenderá acerca de las estructuras dinámicas de datos que pueden crecer o reducirse en tiempo de ejecución.

24

Estructuras de datos

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- A formar estructuras de datos enlazadas utilizando referencias, clases autorreferenciadas y recursividad.
- Cómo las conversiones boxing y unboxing permiten usar valores de tipos simples en donde se esperan objetos `object` en un programa.
- A crear y manipular estructuras dinámicas de datos como listas enlazadas, colas, pilas y árboles binarios.
- Comprenderá varias aplicaciones importantes de las estructuras de datos enlazadas.
- A crear estructuras de datos reutilizables con clases, herencia y composición.

*De muchas cosas
a las que estoy atado,
no he podido liberarme;
y muchas
de las que me he liberado,
han vuelto a mí.*

—Lee Wilson Dodd

*‘¿Podría caminar un poco
más rápido?’ dijo una merluza
a un caracol.*

*‘Hay una marsopa acercándose
mucho a nosotros y está
pisándome la cola.’*

—Lewis Carroll

Siempre hay lugar en la cima.

—Daniel Webster

Empujen; sigan moviéndose.

—Thomas Morton

*Creo que nunca veré
a un poema tan encantador
como un árbol.*

—Joyce Kilmer

- 24.1 Introducción
- 24.2 Tipos `struct` simples, boxing y unboxing
- 24.3 Clases autorreferenciadas
- 24.4 Listas enlazadas
- 24.5 Pilas
- 24.6 Colas
- 24.7 Árboles
 - 24.7.1 Árbol binario de búsqueda de valores enteros
 - 24.7.2 Árbol binario de búsqueda de objetos `IComparable`
- 24.8 Conclusión

24.1 Introducción

Éste es el primero de tres capítulos que hablarán sobre las *estructuras de datos*; las que hemos estudiado hasta ahora son de tamaño fijo, como los arreglos unidimensionales y bidimensionales. En este capítulo presentaremos las *estructuras de datos dinámicas*, que crecen y se reducen en tiempo de ejecución. Las listas enlazadas son colecciones de elementos de datos “alineados en una fila”; los usuarios pueden insertar y eliminar elementos en cualquier parte de una lista enlazada. Las pilas son importantes en los compiladores y sistemas operativos; pueden insertarse y eliminarse elementos sólo en un extremo de una pila: su *parte superior*. Las colas representan líneas de espera; se insertan elementos en la parte final (conocida como *rabo*) de una cola y se eliminan elementos de su parte inicial (conocida como *cabeza*). Los *árboles binarios* facilitan la búsqueda y ordenamiento de los datos de alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos y la compilación de expresiones en lenguaje máquina. Estas estructuras de datos también tienen muchas otras aplicaciones interesantes.

Hablaremos sobre cada uno de estos tipos de estructuras de datos e implementaremos programas para crearlas y manipularlas. Utilizaremos clases, herencia y composición para crear y empaquetar estas estructuras de datos, para reutilizarlas y darles mantenimiento. En el capítulo 25 presentaremos los genéricos, que le permiten declarar estructuras de datos que pueden adaptarse en forma automática para contener datos de cualquier tipo. En el capítulo 26, Colecciones, hablaremos sobre las clases predefinidas de C# que implementan varias estructuras de datos.

Los ejemplos que presentamos en este capítulo son programas prácticos que pueden utilizarse en cursos más avanzados y en aplicaciones industriales. Los programas se enfocan en la manipulación de referencias. Los ejercicios ofrecen una vasta colección de aplicaciones útiles.

24.2 Tipos `struct` simples, boxing y unboxing

Las estructuras de datos que veremos en este capítulo almacenan referencias `object`. Sin embargo, como veremos en breve, podemos almacenar valores tanto de tipo simple como de tipo por referencia en estas estructuras de datos. En esta sección veremos el mecanismo que permite manipular los valores de tipos simples como si fueran objetos.

Tipos `struct` simples

Cada tipo simple (véase el apéndice L, Los tipos simples) tiene un tipo **`struct`** correspondiente en el espacio de nombres `System`, que declara al tipo simple. Estos tipos `struct` se llaman `Boolean`, `Byte`, `SByte`, `Char`, `Decimal`, `Double`, `Single`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int16` y `UInt16`. Los tipos que se declaran con la palabra clave `struct` son implícitamente tipos por valor.

En realidad, los tipos simples son alias para sus tipos `struct` correspondientes, por lo que una variable de un tipo simple puede declararse usando ya sea la palabra clave para ese tipo simple, o el nombre del tipo `struct`; por ejemplo, `int` e `Int32` son indistintos. Los métodos relacionados con un tipo simple se encuentran en el tipo `struct` correspondiente (por ejemplo, el método `Parse`, que convierte un objeto `string` en un valor `int`, se encuentra en el tipo `struct` `Int32`). En la documentación podrá consultar el tipo `struct` correspondiente y ver los métodos disponibles para manipular valores de ese tipo.

Conversiones boxing y unboxing

Todos los tipos `struct` simples heredan de la clase **ValueType** en el espacio de nombres `System`. La clase `ValueType` hereda de la clase `object`. Por ende, cualquier valor de tipo simple puede asignarse a una variable `object`; a esto se le conoce como *conversión boxing*. En una conversión boxing, el valor de tipo simple se copia en un objeto, para que pueda manipularse como un `object`. Las conversiones boxing pueden realizarse en forma explícita o implícita, como se muestra en las siguientes instrucciones:

```
int i = 5; // crea un valor int
object objeto1 = ( object ) i; // conversión boxing explícita del valor int
object objeto2 = i; // conversión boxing implícita del valor int
```

Después de ejecutar el código anterior, tanto `objeto1` como `objeto2` se refieren a dos objetos distintos que contienen una copia del valor entero en la variable `int i`.

Una *conversión unboxing* puede usarse para convertir en forma explícita una referencia `object` a un valor simple, como se muestra en la siguiente instrucción:

```
int int1 = ( int ) objeto1; // conversión unboxing explícita del valor int
```

Si se trata de realizar una conversión unboxing explícita de una referencia `object` que no se refiera al tipo de valor simple correcto, se produce una excepción **InvalidCastException**.

En el capítulo 25, Genéricos, y en el capítulo 26, Colecciones, hablaremos sobre los genéricos y las colecciones genéricas en C#. Como veremos pronto, los genéricos eliminan la sobrecarga de las conversiones boxing y unboxing, al permitirnos crear y utilizar colecciones de tipos de valores específicos.

24.3 Clases autorreferenciadas

Una *clase autorreferenciada* contiene un miembro de referencia que hace referencia a un objeto del mismo tipo de clase. Por ejemplo, la declaración de la clase en la figura 24.1 define la estructura de una clase autorreferenciada, llamada `Nodo`. Este tipo tiene dos variables de instancia `private`: `datos` de tipo entero y la referencia `Nodo` llamada `siguiente`. El miembro `siguiente` hace referencia a un objeto de tipo `Nodo`, un objeto del mismo tipo que se está declarando aquí; es por ello que se utiliza el término “clase autorreferenciada”. El miembro `siguiente` es un *enlace* (es decir, `siguiente` puede usarse para “enlazar” un objeto de tipo `Nodo` con otro objeto del mismo tipo). La clase `Nodo` también tiene dos propiedades: una para la variable de instancia `datos` (llamada `Datos`) y otra para la variable de instancia `siguiente` (llamada `Siguiente`).

Los objetos autorreferenciados pueden enlazarse entre sí para formar estructuras de datos útiles, como listas, colas, pilas y árboles. La figura 24.2 ilustra dos objetos autorreferenciados, enlazados entre sí para formar una lista enlazada. Una barra diagonal inversa (que representa una referencia `null`) se coloca en el miembro de enlace del segundo objeto autorreferenciado para indicar que el enlace no hace referencia a otro objeto. La barra

```
1 // Fig. 24.1: Fig24_01.cs
2 // Una clase autorreferenciada.
3 class Nodo
4 {
5     private int datos; // almacena datos enteros
6     private Nodo siguiente; // almacena referencia al siguiente Nodo
7
8     public Node( int valorDatos )
9     {
10         // cuerpo del constructor
11     } // fin del constructor
12
13     public int Datos
14     {
15         get
16     }
```

Figura 24.1 | Declaración de la clase `Nodo` autorreferenciada. (Parte 1 de 2).


```

17         // cuerpo de get
18     } // fin de get
19     set
20     {
21         // cuerpo de set
22     } // fin de set
23 } // fin de la propiedad Datos
24
25 public Nodo Siguiente
26 {
27     get
28     {
29         // cuerpo de get
30     } // fin de get
31     set
32     {
33         // cuerpo de set
34     } // fin de set
35 } // fin de la propiedad Siguiente
36 } // fin de la clase Nodo

```

Figura 24.1 | Declaración de la clase `Nodo` autorreferenciada. (Parte 2 de 2).

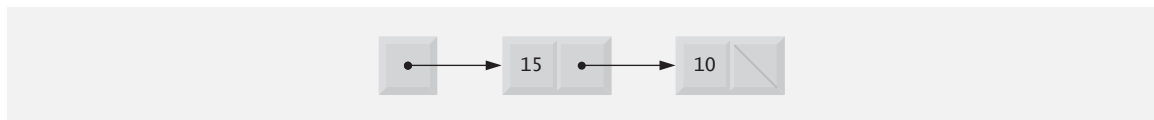


Figura 24.2 | Objetos de una clase autorreferenciada, enlazados entre sí.

diagonal inversa es para fines de ilustración; no corresponde al carácter barra diagonal inversa en C#. Por lo general, una referencia `null` indica el final de una estructura de datos.



Error común de programación 24.1

Si no se establece el enlace en el último nodo de una lista a `null`, se produce un error lógico.

Para crear y mantener estructuras dinámicas de datos se requiere la **asignación dinámica de memoria**: la habilidad para que un programa obtenga más espacio de memoria en tiempo de ejecución, para almacenar nuevos nodos y para liberar el espacio que ya no se necesita. En la sección 9.9 vimos que los programas en C# no liberan explícitamente la memoria asignada en forma dinámica. En vez de ello, C# realiza la recolección automática de basura.

El operador `new` es esencial para la asignación dinámica de memoria. Este operador recibe como operando el tipo del objeto que se asignará en forma dinámica y devuelve una referencia a un objeto de ese tipo. Por ejemplo, la instrucción

```
Nodo nodoParaAgregar = new Nodo( 10 );
```

asigna la cantidad apropiada de memoria para almacenar un `Nodo`, y almacena una referencia a este objeto en `nodoParaAgregar`. Si no hay memoria disponible, `new` lanza una excepción `OutOfMemoryException`. El argumento 10 del constructor especifica los datos del objeto `Nodo`.

Las siguientes secciones hablan sobre listas, pilas, colas y árboles. Estas estructuras de datos se crean y se mantienen con la asignación dinámica de memoria y las clases autorreferenciadas.



Buena práctica de programación 24.1

Al crear un gran número de objetos, considere evaluar la probabilidad de que se lance una excepción `OutOfMemoryException`. Realice un procesamiento de errores apropiado si no se asigna la memoria solicitada.

24.4 Listas enlazadas

Una *lista enlazada* es una colección lineal (es decir, una secuencia) de objetos de una clase autorreferenciada, conocidos como *nodos*, que están conectados por enlaces de referencia; es por ello que se utiliza el término lista “enlazada”. Un programa accede a una lista enlazada a través de una referencia al primer nodo en la lista. El programa accede a cada nodo subsiguiente a través del miembro de referencia de enlace almacenado en el nodo anterior. Por convención, la referencia de enlace en el último nodo de una lista se establece en `null` para indicar el final de la lista. Los datos se almacenan en forma dinámica en una lista enlazada; el programa crea cada nodo según sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo referencias a objetos de otras clases. Las pilas y las colas son también estructuras de datos lineales; de hecho, son versiones restringidas de las listas enlazadas. Los árboles son estructuras de datos no lineales.

Pueden almacenarse listas de datos en los arreglos, pero las listas enlazadas ofrecen varias ventajas. Una lista enlazada es apropiada cuando el número de elementos de datos que se van a representar en la estructura de datos es impredecible. A diferencia de una lista enlazada, el tamaño de un arreglo convencional en C# no puede alterarse, ya que el tamaño del arreglo se fija al momento de crearlo. Los arreglos convencionales pueden llenarse, pero las listas enlazadas se llenan sólo cuando el sistema no tiene la memoria suficiente para satisfacer las solicitudes de asignación dinámica de memoria.



Tip de rendimiento 24.1

Un arreglo puede declararse de manera que contenga más elementos que el número de elementos esperados, pero esto desperdicia memoria. Las listas enlazadas proporcionan una mejor utilización de memoria en estas situaciones, ya que pueden crecer y reducirse en tiempo de ejecución.



Tip de rendimiento 24.2

Después de localizar el punto de inserción para un nuevo elemento en una lista enlazada ordenada, la inserción de un elemento es rápida; sólo hay que modificar dos referencias. Todos los nodos existentes permanecen en sus posiciones actuales en memoria.

Los programadores pueden mantener listas enlazadas en orden con sólo insertar cada nuevo elemento en el punto apropiado de la lista (claro que lleva tiempo localizar el punto de inserción apropiado). Los elementos existentes en la lista no necesitan moverse.



Tip de rendimiento 24.3

Los elementos de un arreglo se almacenan en posiciones contiguas de memoria, para permitir el acceso inmediato a cualquiera de sus elementos; la dirección de cualquier elemento puede calcularse directamente con base en su índice. Las listas enlazadas no permiten dicho acceso inmediato a sus elementos; para acceder a uno de ellos se tiene que recorrer la lista desde su parte inicial.

Por lo general, las listas enlazadas no se almacenan en posiciones contiguas en memoria. En vez de ello, los nodos son adyacentes en forma lógica. La figura 24.3 ilustra una lista enlazada con varios nodos.

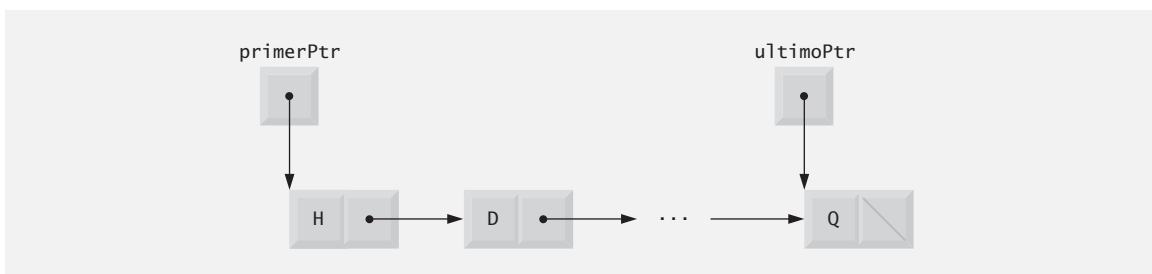


Figura 24.3 | Representación gráfica de una lista enlazada.



Tip de rendimiento 24.4

El uso de estructuras de datos enlazadas y la asignación dinámica de memoria (en vez de arreglos) para las estructuras de datos que crecen y se reducen en tiempo de ejecución puede ahorrar memoria. Sin embargo, tenga en cuenta que los enlaces de referencia ocupan espacio, y la asignación dinámica de memoria incurre en la sobrecarga relacionada con las llamadas a métodos.

Implementación de la lista enlazada

El programa de las figuras 24.4 y 24.5 utiliza un objeto de la clase `Lista` para manipular una lista de tipos de objetos varios. El método `Main` de la clase `PruebaLista` (figura 24.5) crea una lista de objetos, inserta objetos al principio de la lista mediante el método `InsertarAlFrente` de `Lista`, inserta objetos al final de la lista mediante el método `InsertarAlFinal` de `Lista`, elimina objetos de la parte frontal de la lista usando el método `EliminarDelFrente` de `Lista` y elimina objetos de la parte final de la lista usando el método `EliminarDelFinal` de `Lista`. Después de cada operación de inserción y eliminación, el programa invoca al método `Imprimir` de `Lista` para mostrar su contenido actual. Si se trata de eliminar un elemento de una lista vacía, ocurre una excepción `ExcepcionListaVacía`. A continuación presentamos una discusión detallada del programa.



Tip de rendimiento 24.5

Las operaciones de inserción y eliminación en un arreglo almacenado pueden llevar tiempo; todos los elementos que van después del elemento insertado o eliminado deben desplazarse en forma apropiada.

El programa consiste en cuatro clases: `NodoLista` (figura 24.4, líneas 8-49), `Lista` (líneas 52-165), `ExcepcionListaVacía` (líneas 168-174) y `PruebaLista` (figura 24.5). Las clases en la figura 24.4 crean una biblioteca de listas enlazadas (definida en el espacio de nombres `BibliotecaListasEnlazadas`) que puede reutilizarse durante todo el capítulo. Debe colocar el código de la figura 24.4 en su propio proyecto de biblioteca de clases, como se describió en la sección 9.14.

En cada objeto `Lista` hay encapsulada una lista enlazada de objetos `NodoLista`. La clase `NodoLista` (figura 24.4, líneas 8-49) contiene dos variables de instancia: `datos` y `siguiente`. Los datos miembro pueden hacer referencia a cualquier objeto. [Nota: Por lo general, una estructura de datos contendrá datos de un tipo solamente, o datos de cualquier tipo derivado de un tipo base. En este ejemplo, utilizamos datos de varios tipos derivados de `object`, para demostrar que nuestra clase `Lista` puede almacenar datos de cualquier tipo. El miembro `siguiente` almacena una referencia al siguiente objeto `NodoLista` en la lista enlazada. Los constructores de `NodoLista` (líneas 15-18 y 22-26) nos permiten inicializar un objeto `NodoLista` que se colocará al final de una `Lista`, o antes de un `NodoLista` específico en una `Lista`, respectivamente. Una `Lista` accede a las variables miembro de `NodoLista` a través de las propiedades `Siguiente` (líneas 29-39) y `Datos` (líneas 42-48), respectivamente.

```

1 // Fig. 24.4: BibliotecaListasEnlazadas.cs
2 // Declaraciones de las clases NodoLista y Lista.
3 using System;
4
5 namespace BibliotecaListasEnlazadas
6 {
7     // clase para representar un nodo en una lista
8     class NodoLista
9     {
10         private object datos; // almacena los datos para este nodo
11         private NodoLista siguiente; // almacena una referencia al siguiente nodo
12
13         // constructor para crear un NodoLista que haga referencia a valorDatos
14         // y sea el último nodo en la lista
15         public NodoLista( object valorDatos )
16             : this( valorDatos, null )
17         {
18             } // fin del constructor predeterminado

```

Figura 24.4 | Las clases `NodoLista`, `Lista` y `ExcepcionListaVacía`. (Parte I de 4).

```

19
20 // constructor para crear un NodoLista que haga referencia a valorDatos
21 // y se refiera al siguiente NodoLista en la Lista
22 public NodoLista( object valorDatos, NodoLista siguienteNode )
23 {
24     datos = valorDatos;
25     siguiente = siguienteNode;
26 } // fin del constructor
27
28 // propiedad Siguiente
29 public NodoLista Siguiente
30 {
31     get
32     {
33         return siguiente;
34     } // fin de get
35     set
36     {
37         siguiente = value;
38     } // fin de set
39 } // fin de la propiedad Siguiente
40
41 // propiedad Datos
42 public object Datos
43 {
44     get
45     {
46         return datos;
47     } // fin de get
48 } // fin de la propiedad Datos
49 } // fin de la clase NodoLista
50
51 // declaración de la clase Lista
52 public class Lista
53 {
54     private NodoLista primerNode;
55     private NodoLista ultimoNode;
56     private string nombre; // cadena a mostrar, tal como "lista"
57
58     // construye Lista vacía con el nombre especificado
59     public Lista( string nombreLista )
60     {
61         nombre = nombreLista;
62         primerNode = ultimoNode = null;
63     } // fin del constructor
64
65     // construye Lista vacía con "lista" como su nombre
66     public Lista()
67         : this( "lista" )
68     {
69     } // fin del constructor predeterminado
70
71     // Inserta objeto al frente de la Lista. Si está vacía,
72     // primerNode y ultimoNode harán referencia al mismo objeto.
73     // En caso contrario, primerNode hace referencia al nuevo nodo.
74     public void InsertarAlFrente( object insertarElemento )
75     {
76         if ( EstaVacía() )
77             primerNode = ultimoNode = new NodoLista( insertarElemento );

```

Figura 24.4 | Las clases NodoLista, Lista y ExcepcionListaVacía. (Parte 2 de 4).

```

78         else
79             primerNodo = new NodoLista( insertarElemento, primerNodo );
80     } // fin del método InsertarAlFrente
81
82     // Inserta objeto al final de la Lista. Si está vacía,
83     // primerNodo y ultimoNodo harán referencia al mismo objeto.
84     // En caso contrario, la propiedad Siguiente de ultimoNodo hace referencia al
85     // nuevo nodo.
86     public void InsertarAlFinal( object insertarElemento )
87     {
88         if ( EstaVacía() )
89             primerNodo = ultimoNodo = new NodoLista( insertarElemento );
90         else
91             ultimoNodo = ultimoNodo.Siguiente = new NodoLista( insertarElemento );
92     } // fin del método InsertarAlFinal
93
94     // elimina el primer nodo de la Lista
95     public object EliminarDelFrente()
96     {
97         if ( EstaVacía() )
98             throw new ExcepcionListaVacía( nombre );
99
100         object eliminarElemento = primerNodo.Datos; // recupera los datos
101
102         // restablece las referencias primerNodo y ultimoNodo
103         if ( primerNodo == ultimoNodo )
104             primerNodo = ultimoNodo = null;
105         else
106             primerNodo = primerNodo.Siguiente;
107
108         return eliminarElemento; // devuelve los datos eliminados
109     } // fin del método EliminarDelFrente
110
111     // elimina el último nodo de la Lista
112     public object EliminarDelFinal()
113     {
114         if ( EstaVacía() )
115             throw new ExcepcionListaVacía( nombre );
116
117         object eliminarElemento = ultimoNodo.Datos; // obtiene los datos
118
119         // restablece las referencias primerNodo y ultimoNodos
120         if ( primerNodo == ultimoNodo )
121             primerNodo = ultimoNodo = null;
122         else
123         {
124             NodoLista actual = primerNodo;
125
126             // itera mientras el nodo actual no sea el ultimoNodo
127             while ( actual.Siguiente != ultimoNodo )
128                 actual = actual.Siguiente; // avanza al siguiente nodo
129
130             // actual es el nuevo ultimoNodo
131             ultimoNodo = actual;
132             actual.Siguiente = null;
133         } // fin de else
134
135         return eliminarElemento; // devuelve los datos eliminados
136     } // fin del método EliminarDelFinal

```

Figura 24.4 | Las clases NodoLista, Lista y ExcepcionListaVacía. (Parte 3 de 4).

```

136
137 // devuelve verdadero si la Lista está vacía
138 public bool EstaVacía()
139 {
140     return primerNodo == null;
141 } // fin del método EstaVacía
142
143 // imprime en pantalla el contenido de la Lista
144 public void Imprimir()
145 {
146     if ( EstaVacía() )
147     {
148         Console.WriteLine( nombre + " Vacía" );
149         return;
150     } // fin de if
151
152     Console.Write( "La " + nombre + " es: " );
153
154     NodoLista actual = primerNodo;
155
156     // imprime los datos del nodo actual mientras no esté al final de la lista
157     while ( actual != null )
158     {
159         Console.Write( actual.Datos + " " );
160         actual = actual.Siguiente;
161     } // fin de while
162
163     Console.WriteLine( "\n" );
164 } // fin del método Imprimir
165 } // fin de la clase Lista
166
167 // declaración de la clase ExcepcionListaVacía
168 public class ExcepcionListaVacía : ApplicationException
169 {
170     public ExcepcionListaVacía( string nombre )
171         : base( "La " + nombre + " está vacía" )
172     {
173     } // fin del constructor
174 } // fin de la clase ExcepcionListaVacía
175 } // fin del espacio de nombres BibliotecaListasEnlazadas

```

Figura 24.4 | Las clases NodoLista, Lista y ExcepcionListaVacía. (Parte 4 de 4).

La clase Lista (líneas 52-165) contiene las variables de instancia `private primerNodo` (una referencia al primer `NodoLista` en una Lista) y `ultimoNodo` (una referencia al último `NodoLista` en una Lista). Los constructores (líneas 59-63 y 66-69) inicializan ambas referencias a `null` y nos permiten especificar el nombre de la Lista, para fines de visualizar los resultados en pantalla. `InsertarAlFrente` (líneas 74-80), `InsertarAlFinal` (líneas 85-91), `EliminarDelFrente` (líneas 94-108) y `EliminarDelFinal` (líneas 111-135) son los métodos principales de la clase Lista. El método `EstaVacía` (líneas 138-141) es un *método predicado*, el cual determina si la lista está vacía (es decir, la referencia al primer nodo de la lista es `null`). Por lo general, los métodos predicados evalúan una condición y no modifican el objeto desde el que se llaman. Si la lista está vacía, el método `EstaVacía` devuelve `true`; en caso contrario, devuelve `false`. El método `Imprimir` (líneas 144-164) muestra el contenido de la lista. Después de la figura 24.5 hablaremos con detalle sobre los métodos de la clase Lista.

La clase `ExcepcionListaVacía` (líneas 168-174) define una clase de excepción que utilizamos para indicar operaciones ilegales en una Lista vacía.

La clase `PruebaLista` (figura 24.5) utiliza la biblioteca de listas enlazadas para crear y manipular una lista enlazada. [Nota: En el proyecto que contiene la figura 24.5, debe agregar una referencia a la biblioteca de clases que

contiene las clases de la figura 24.4. Si utiliza nuestro ejemplo existente, tal vez necesite actualizar esta referencia.] La línea 11 crea un nuevo objeto `Lista` y lo asigna a la variable `lista`. Las líneas 14-17 crean datos para agregarlos a la lista. Las líneas 20-27 utilizan los métodos de inserción de `Lista` para insertar estos valores y utilizar el método `Imprimir` de `Lista` para mostrar en pantalla el contenido de `lista`, después de cada inserción. Observe que se llevan a cabo conversiones boxing en los valores de las variables de tipos simples de las líneas 20, 22 y 24, en donde se esperan referencias `object`. El código dentro del bloque `try` (líneas 33-50) elimina objetos a través de los métodos de eliminación de `Lista`, imprime en pantalla cada objeto eliminado e imprime en pantalla el contenido de `lista` después de cada eliminación. Si hay un intento de eliminar un objeto de una lista vacía, la instrucción `catch` en las líneas 51-54 atrapa la `ExcepcionListaVacía` y muestra un mensaje de error.

```

1  // Fig. 24.5: PruebaLista.cs
2  // Prueba de la clase Lista.
3  using System;
4  using BibliotecaListasEnlazadas;
5
6  // clase para probar la funcionalidad de la clase Lista
7  class PruebaLista
8  {
9      static void Main( string[] args )
10     {
11         Lista lista = new Lista(); // crea el contenedor Lista
12
13         // crea los datos para almacenar en la Lista
14         bool unBooleano = true;
15         char unCaracter = '$';
16         int unEntero = 34567;
17         string unaCadena = "hola";
18
19         // usa los métodos de inserción de Lista
20         lista.InsertarAlFrente( unBooleano );
21         lista.Imprimir();
22         lista.InsertarAlFrente( unCaracter );
23         lista.Imprimir();
24         lista.InsertarAlFinal( unEntero );
25         lista.Imprimir();
26         lista.InsertarAlFinal( unaCadena );
27         lista.Imprimir();
28
29         // usa los métodos de eliminación de Lista
30         object objetoEliminado;
31
32         // elimina los datos de la lista e imprime después de cada eliminación
33         try
34         {
35             objetoEliminado = lista.EliminarDelFrente();
36             Console.WriteLine( objetoEliminado + " eliminado" );
37             lista.Imprimir();
38
39             objetoEliminado = lista.EliminarDelFrente();
40             Console.WriteLine( objetoEliminado + " eliminado" );
41             lista.Imprimir();
42
43             objetoEliminado = lista.EliminarDelFinal();
44             Console.WriteLine( objetoEliminado + " eliminado" );
45             lista.Imprimir();

```

Figura 24.5 | Demostración de una lista enlazada. (Parte I de 2).

```

46
47     objetoEliminado = lista.EliminarDelFinal();
48     Console.WriteLine( objetoEliminado + " eliminado" );
49     lista.Imprimir();
50 } // fin de try
51 catch ( ExcepcionListaVacia excepcionListaVacia )
52 {
53     Console.Error.WriteLine( "\n" + excepcionListaVacia );
54 } // fin de catch
55 } // fin del método Main
56 } // fin de la clase PruebaLista

```

```

La lista es: True

La lista es: $ True

La lista es: $ True 34567

La lista es: $ True 34567 hola

$ eliminado
La lista es: True 34567 hola

True eliminado
La lista es: 34567 hola

hola eliminado
La lista es: 34567

34567 eliminado
lista Vacía

```

Figura 24.5 | Demostración de una lista enlazada. (Parte 2 de 2).

Método InsertarAlFrente

En las siguientes páginas hablaremos sobre cada uno de los métodos de la clase `Lista` con detalle. El método `InsertarAlFrente` (figura 24.4, líneas 74-80) coloca un nuevo nodo en la parte frontal de la lista. El método consiste en tres pasos:

1. Llama a `EstaVacio` para determinar si la lista está vacía (línea 76).
2. Si la lista está vacía, establece `primerNodo` y `ultimoNodo` para que hagan referencia a un nuevo `NodoLista`, inicializado con `insertarElemento` (línea 77). El constructor `NodoLista` en las líneas 15-18 de la figura 24.4 llama al constructor de `NodoLista` en las líneas 22-26, el cual establece la variable de instancia `datos` para que haga referencia al objeto `object` que se pasa como primer argumento, y establece la referencia `siguiente` en `null`.
3. Si la lista no está vacía, el nuevo nodo se “enlaza” a la lista, estableciendo `primerNodo` para que haga referencia a un nuevo objeto `NodoLista`, inicializado con `insertarElemento` y `primerNodo` (línea 79). Cuando se ejecuta el constructor de `NodoLista` (líneas 22-26), establece la variable de instancia `datos` para que haga referencia al objeto `object` que se pasa como primer argumento, y realiza la inserción estableciendo la referencia `siguiente` al `NodoLista` que se pasa como segundo argumento.

En la figura 24.6, la parte (a) muestra una lista y un nuevo nodo durante la operación `InsertarAlFrente` y antes de que se enlace el nuevo nodo a la lista. Las líneas punteadas y las flechas en la parte (b) ilustran el *paso 3* de la operación `InsertarAlFrente`, la cual permite que el nodo que contiene 12 se convierta en el nuevo frente de la lista.

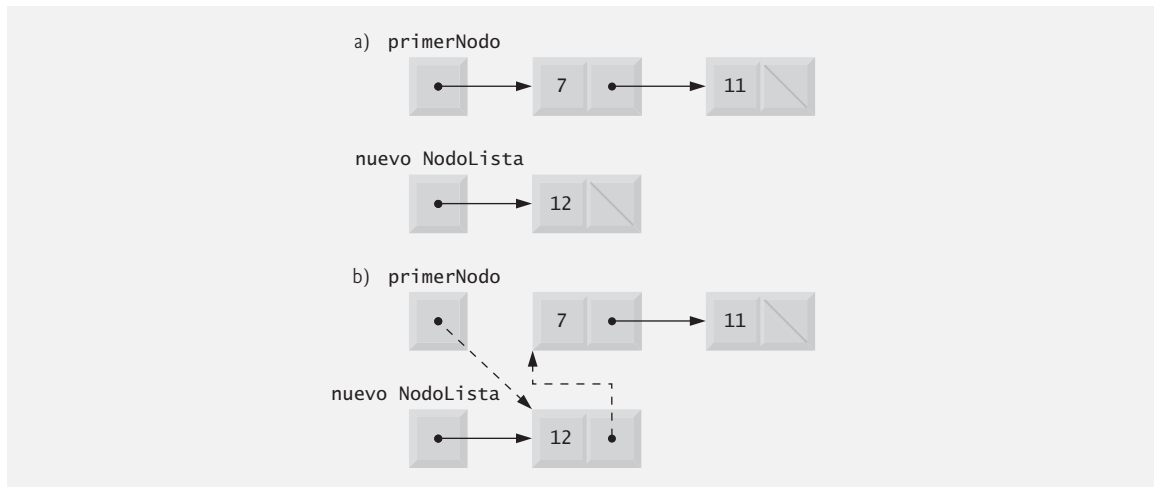


Figura 24.6 | La operación InsertarAlFrente.

Método InsertarAlFinal

El método `InsertarAlFinal` (figura 24.4, líneas 85-91) coloca un nuevo nodo en la parte final de la lista. El método consiste en tres pasos:

1. Llama a `EstaVacía` para determinar si la lista está vacía (línea 87).
2. Si la lista está vacía, establece `primerNodo` y `ultimoNodo` para que hagan referencia a un nuevo `NodoLista` inicializado con `insertarElemento` (línea 88). El constructor de `NodoLista` en las líneas 15-18 llama al constructor de `NodoLista` en las líneas 22-26, el cual establece la variable de instancia `datos` para que haga referencia al objeto `object` que se pasa como primer argumento, y establece la referencia siguiente a `null`.
3. Si la lista no está vacía, enlaza el nuevo nodo a la lista, estableciendo `ultimoNodo` y `ultimoNodo.siguiente` para que haga referencia a un nuevo objeto `NodoLista`, inicializado con `insertarElemento` (línea 90). Cuando el constructor de `NodoLista` (líneas 15-18) se ejecuta, hace una llamada al constructor en las líneas 22-26, el cual establece la variable de instancia `datos` para que haga referencia al objeto `object` que se pasa como argumento, y establece la referencia siguiente a `null`.

En la figura 24.7, la parte (a) muestra una lista y un nuevo nodo durante la operación `InsertarAlFinal`, antes de que se haya enlazado el nuevo nodo a la lista. Las líneas punteadas y las flechas en la parte (b) ilustran el *paso 3* del método `InsertarAlFinal`, el cual permite que se agregue un nuevo nodo al final de una lista que no esté vacía.

Método EliminarDelFrente

El método `EliminarDelFrente` (figura 24.4, líneas 94-108) elimina el nodo frontal de la lista y devuelve una referencia a los datos eliminados. El método lanza una `ExcepcionListaVacía` (línea 97) si el programador trata de eliminar un nodo de una lista vacía. En caso contrario, el método devuelve una referencia a los datos eliminados. Después de determinar si una `Lista` no está vacía, el método consiste en cuatro pasos para eliminar el primer nodo:

1. Asigna `primerNodo.Datos` (los datos que se van a eliminar de la lista) a la variable `eliminarElemento` (línea 99).
2. Si los objetos a los que `primerNodo` y `ultimoNodo` hacen referencia son el mismo objeto, la lista sólo tiene un elemento, por lo que el método establece `primerNodo` y `ultimoNodo` a `null` (línea 103) para eliminar el nodo de la lista (dejando la lista vacía).
3. Si la lista tiene más de un nodo, el método deja la referencia `ultimoNodo` como está, y asigna `primerNodo.siguiente` a `primerNodo` (línea 105). Así, `primerNodo` hace referencia al nodo que anteriormente era el segundo nodo en la lista.

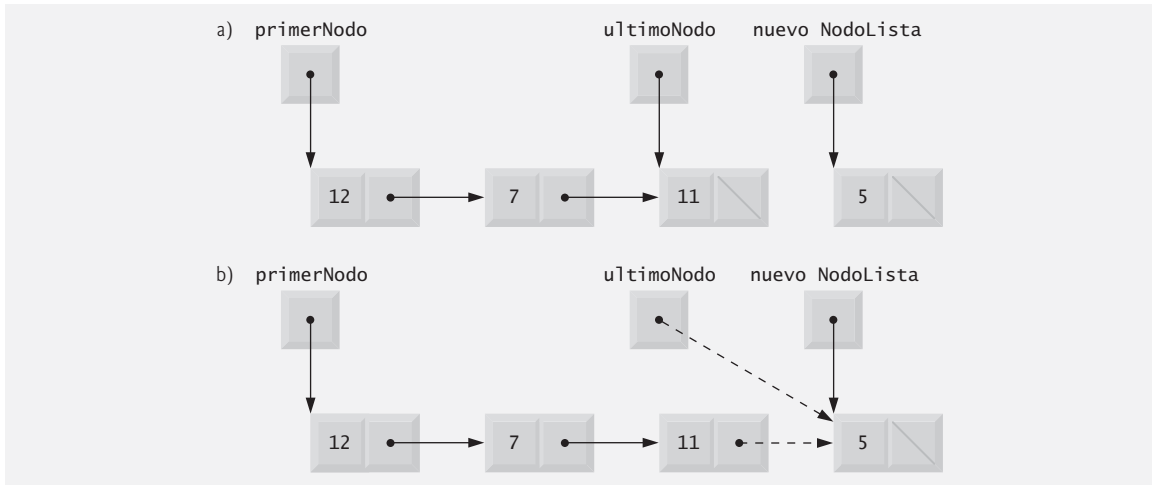


Figura 24.7 | La operación `InsertarAlFinal`.

4. Devuelve la referencia `eliminarElemento` (línea 107).

En la figura 24.8, la parte (a) ilustra una lista antes de una operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de las referencias.

Método `EliminarDelFinal`

El método `EliminarDelFinal` (figura 24.4, líneas 111-135) elimina el último nodo de una lista y devuelve una referencia a los datos eliminados. El método lanza una `ExcepcionListaVacía` (línea 114) si el programa intenta eliminar un nodo de una lista vacía. El método consiste en varios pasos:

1. Asigna `ultimoNodo.Datos` (los datos que se van a eliminar de la lista) a la variable `eliminarElemento` (línea 116).
2. Si `primerNodo` y `ultimoNodo` hacen referencia al mismo objeto (línea 119), la lista sólo tiene un elemento, por lo que el método establece `primerNodo` y `ultimoNodo` a `null` (línea 120) para eliminar ese nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo, crea la variable `NodoLista actual` y le asigna `primerNodo` (línea 123).

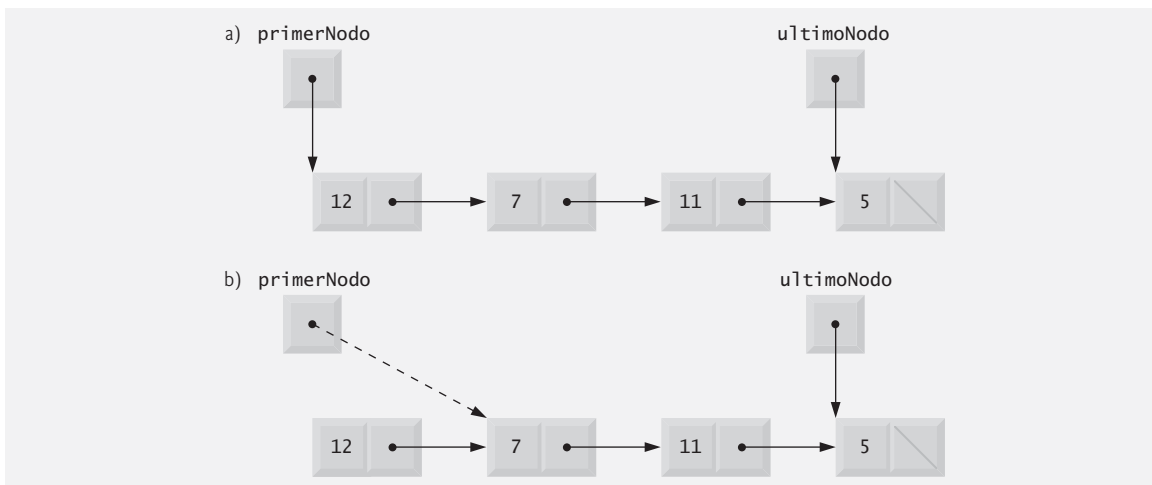


Figura 24.8 | La operación `EliminarDelFrente`.

4. Ahora recorre la lista con `actual`, hasta que hace referencia al nodo que está antes del último nodo. El ciclo `while` (líneas 126-127) asigna `actual.Siguiente` a `actual.Siguiente`, siempre y cuando `actual.Siguiente` no sea igual a `ultimoNodo`.
5. Después de localizar el antepenúltimo nodo, asigna `actual` a `ultimoNodo` (línea 130) para actualizar el nodo que esté al último en la lista.
6. Establece `actual.Siguiente` a `null` (línea 131) para eliminar el último nodo de la lista y terminarla en el nodo actual.
7. Devuelve la referencia `eliminarElemento` (línea 134).

En la figura 24.9, la parte (a) ilustra una lista antes de una operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de las referencias.

Método Imprimir

El método `Imprimir` (figura 24.4, líneas 144-164) determina primero si la lista está vacía (línea 146). Si es así, `Imprimir` muestra un objeto `string` que consiste en el nombre de la lista y la cadena "vacía ", y después devuelve el control al método que hizo la llamada. En caso contrario, `Imprimir` muestra en pantalla los datos en la lista. El método imprime un objeto `string` que consiste en la cadena "La ", el nombre de la lista y la cadena " es: ". Después, la línea 154 crea la variable `NodoLista actual` y la inicializa con `primerNodo`. Mientras que `actual` no sea `null`, significa que hay más elementos en la lista. Por lo tanto, el método muestra `actual.Datos` (línea 159) y después asigna `actual.Siguiente` a `actual` (línea 160), para avanzar al siguiente nodo en la lista.

Listas lineales y circulares, de enlace simple y doblemente enlazadas

El tipo de lista enlazada que hemos visto es una *lista de enlace simple*: la lista comienza con una referencia al primer nodo, y cada nodo contiene una referencia al siguiente nodo "en secuencia". Esta lista termina con un nodo cuyo miembro de referencia tiene el valor `null`. Una lista de enlace simple puede recorrerse sólo en una dirección.

Una *lista circular de enlace simple* (figura 24.10) empieza con una referencia al primer nodo, y cada nodo contiene una referencia al siguiente nodo. El "último nodo" no contiene una referencia `null`; en vez de ello, la referencia en el último nodo apunta de vuelta al primer nodo, con lo cual se cierra el "círculo".

Una *lista doblemente enlazada* (figura 24.11) permite realizar recorridos tanto hacia delante como hacia atrás. A menudo, dicha lista se implementa con dos "referencias iniciales": una que hace referencia al primer elemento de la lista, para permitir un recorrido desde la parte inicial hasta la parte final, y una que hace referencia al último elemento, para permitir un recorrido desde la parte final hasta la parte inicial. Cada nodo tiene tanto

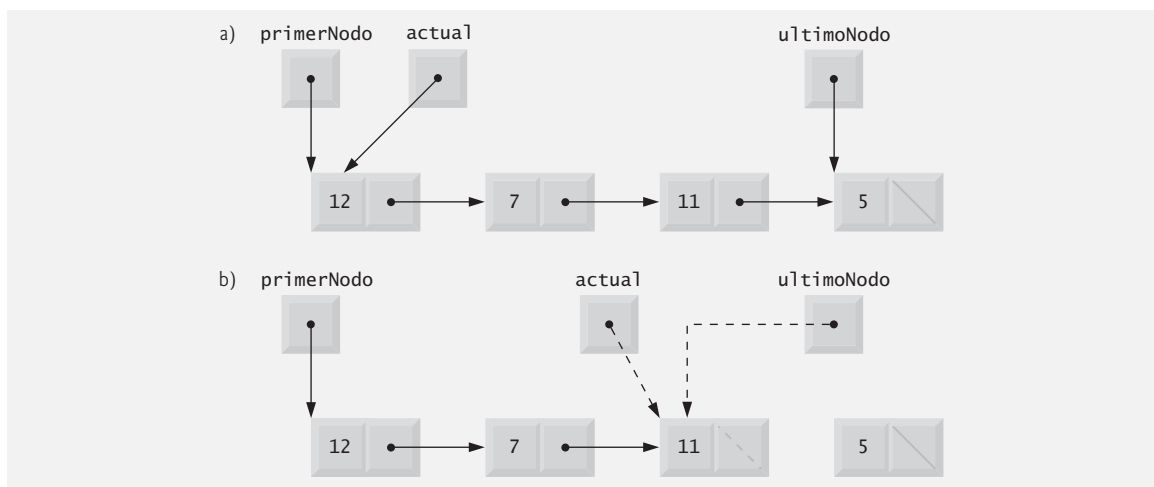


Figura 24.9 | La operación `EliminarDelFinal`.

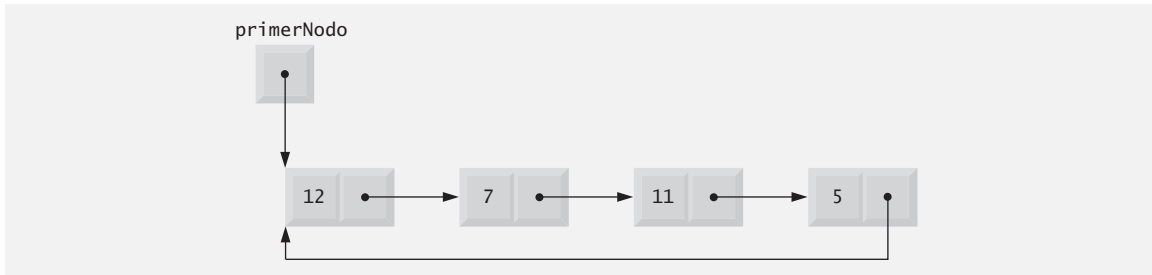


Figura 24.10 | Lista circular de enlace simple.

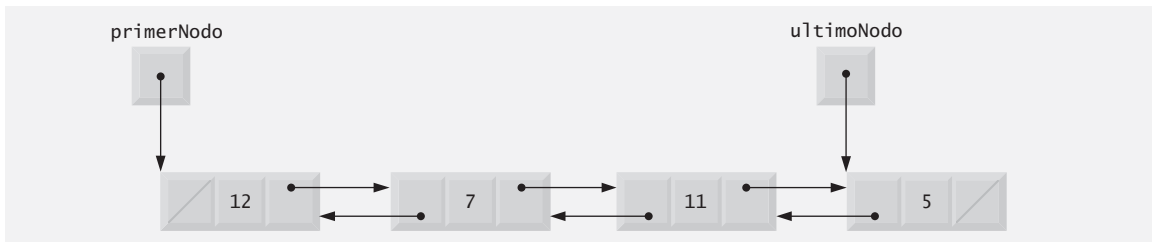


Figura 24.11 | Lista doblemente enlazada.

una referencia hacia delante al siguiente nodo en la lista, como una referencia hacia atrás al nodo anterior en la lista. Por ejemplo, si su lista contiene un directorio telefónico alfabetizado, la búsqueda de alguien cuyo nombre empiece con una letra cerca de la parte inicial del alfabeto podría empezar desde la parte inicial de la lista. La búsqueda de alguien cuyo nombre empiece con una letra cerca del final del alfabeto podría empezar desde la parte final de la lista.

En una *lista circular doblemente enlazada* (figura 24.12), la referencia hacia delante del último nodo se refiere al primer nodo, y la referencia hacia atrás del primer nodo se refiere al último nodo, con lo cual se cierra el “círculo”.

24.5 Pilas

Una *pila* es una versión restringida de una lista enlazada; una pila recibe nuevos nodos y libera nodos sólo desde su parte superior. Por esta razón, a una pila se le conoce como estructura de datos *UEPS* (*último en entrar, primero en salir*).

Las operaciones principales para manipular una pila son *push* (*meter*) y *pop* (*sacar*). La operación meter agrega un nuevo nodo a la parte superior de la pila. La operación sacar elimina un nodo de la parte superior de la pila, y devuelve el elemento de datos del nodo que se extrajo.



Figura 24.12 | Lista circular, doblemente enlazada.

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, cuando un programa llama a un método, éste debe saber cómo regresar al programa que lo llamó, por lo que la dirección de retorno se mete en la pila de llamadas a métodos. Si ocurre una serie de llamadas a métodos, los valores de retorno sucesivos se meten en la pila, en el orden “último en entrar, primero en salir”, de manera que cada método pueda regresar al objeto que lo llamó. Las pilas soportan las llamadas recursivas a métodos, de la misma forma que soportan las llamadas convencionales no recursivas a los métodos.

El espacio de nombres `System.Collections` contiene la clase `Stack` para implementar y manipular pilas que puedan crecer y reducirse durante la ejecución de un programa. En los capítulos 25 y 26 hablaremos sobre la clase `Stack`.

En nuestro siguiente ejemplo, aprovecharemos la estrecha relación entre las listas y las pilas para implementar una clase de pila, mediante la reutilización de una clase de lista. Mostraremos dos formas de reutilización. Primero implementaremos la clase de pila; para ello heredaremos de la clase `Lista` de la figura 24.4. Después implementaremos una clase de pila que funciona en forma idéntica a través de la composición, incluyendo un objeto `Lista` como miembro `private` de una clase de pila.

Clase de pila que hereda de Lista

El programa de las figuras 24.13 y 24.14 crea una clase de pila; para ello hereda de la clase `Lista` de la figura 24.4 (línea 9). Queremos que la pila tenga los métodos `Push`, `Pop`, `EstaVacía` e `Imprimir`. En esencia, éstos son los métodos `InsertarAlFrente`, `EliminarDelFrente`, `EstaVacía` e `Imprimir` de la clase `Lista`. Desde luego que la clase `Lista` contiene otros métodos (como `InsertarAlFinal` y `EliminarDelFinal`) que preferimos no hacer accesibles a través de la interfaz `public` de la pila. Es importante recordar que todos los métodos en la interfaz `public` de la clase `Lista` son también métodos `public` de la clase derivada `HerenciaPila` (figura 24.13).

```

1  // Fig. 24.13: BibliotecaHerenciaPila.cs
2  // Implementación de una pila mediante la herencia de la clase Lista.
3  using System;
4  using BibliotecaListasEnlazadas;
5
6  namespace BibliotecaHerenciaPila
7  {
8      // la clase HerenciaPila hereda las herramientas de la clase Lista
9      public class HerenciaPila : Lista
10     {
11         // pasa el nombre "pila" al constructor de Lista
12         public HerenciaPila()
13             : base( "pila" )
14         {
15         } // fin del constructor
16
17         // coloca valorDatos en la parte superior de la pila, insertando
18         // valorDatos al frente de la lista enlazada
19         public void Push( object valorDatos )
20         {
21             InsertarAlFrente(valorDatos);
22         } // fin del método Push
23
24         // elimina el elemento de la parte superior de la pila; para ello elimina
25         // el elemento que está al frente de la lista enlazada
26         public object Pop()
27         {
28             return EliminarDelFrente();
29         } // fin del método Pop
30     } // fin de la clase HerenciaPila
31 } // fin del espacio de nombres BibliotecaHerenciaPila

```

Figura 24.13 | `HerenciaPila` extiende a la clase `Lista`.

La implementación de cada método `HerenciaPila` llama al método de `Lista` apropiado; el método `Push` llama a `InsertarAlFrente`, el método `Pop` llama a `EliminarDelFrente`. La clase `HerenciaPila` no define los métodos `EstaVacía` e `Imprimir`, ya que hereda estos métodos de la clase `Lista` en su interfaz `public`. Observe que la clase `HerenciaPila` utiliza el espacio de nombres `BibliotecaListasEnlazadas` (figura 24.4); por ende, la biblioteca de clases que define a `HerenciaPila` debe tener una referencia a la biblioteca de clases `BibliotecaListasEnlazadas`.

El método `Main` de `PruebaHerenciaPila` (figura 24.14) utiliza la clase `HerenciaPila` para crear una pila de objetos `object` llamada `pila` (línea 12). Las líneas 15-18 definen cuatro valores que se meterán en la pila y se sacarán de ella. El programa mete a la pila (líneas 21, 23, 25 y 27) un valor `bool` que contiene `true`, un `char` que contiene '\$', un `int` que contiene 34567 y un `string` que contiene "hola". Un ciclo `while` infinito (líneas 33-38) saca los elementos de la pila. Cuando está vacía, el método `Pop` lanza una `ExcepcionListaVacía` y el programa muestra el rastreo de pila de la excepción, el cual muestra la pila del programa en ejecución al momento en que ocurrió la excepción. El programa utiliza el método `Imprimir` (que `HerenciaPila` hereda de la clase `Lista`) para imprimir en pantalla el contenido de la pila después de cada operación. Observe que la clase `PruebaHerenciaPila` utiliza el espacio de nombres `BibliotecaListasEnlazadas` (figura 24.4) y el espacio de nombres `BibliotecaHerenciaPila` (figura 24.13); por ende, la solución para la clase `PruebaHerenciaPila` debe tener referencias a ambas bibliotecas de clases.

```

1  // Fig. 24.14: PruebaHerenciaPila.cs
2  // Prueba de la clase HerenciaPila.
3  using System;
4  using BibliotecaHerenciaPila;
5  using BibliotecaListasEnlazadas;
6
7  // demuestra la funcionalidad de la clase HerenciaPila
8  class PruebaHerenciaPila
9  {
10     static void Main( string[] args )
11     {
12         HerenciaPila pila = new HerenciaPila();
13
14         // crea objetos para almacenarlos en la pila
15         bool unBooleano = true;
16         char unCaracter = '$';
17         int unEntero = 34567;
18         string unaCadena = "hola";
19
20         // usa el método Push para agregar elementos a la pila
21         pila.Push( unBooleano );
22         pila.Imprimir();
23         pila.Push( unCaracter );
24         pila.Imprimir();
25         pila.Push( unEntero );
26         pila.Imprimir();
27         pila.Push( unaCadena );
28         pila.Imprimir();
29
30         // elimina elementos de la pila
31         try
32         {
33             while ( true )
34             {
35                 object objetoEliminado = pila.Pop();
36                 Console.WriteLine( objetoEliminado + " se sacó" );
37                 pila.Imprimir();

```

Figura 24.14 | Uso de la clase `HerenciaPila`. (Parte I de 2).

```

38         } // fin de while
39     } // fin de try
40     catch ( ExcepcionListaVacía excepcionListaVacía )
41     {
42         // si ocurre una excepción, imprime el rastreo de pila
43         Console.Error.WriteLine( excepcionListaVacía.StackTrace );
44     } // fin de catch
45 } // fin de Main
46 } // fin de la clase PruebaHerenciaPila

```

```

La pila es: True

La pila es: $ True

La pila es: 34567 $ True

La pila es: hola 34567 $ True

hola se sacó
La pila es: 34567 $ True

34567 se sacó
La pila es: $ True

$ se sacó
La pila es: True

True se sacó
pila Vacía
  en BibliotecaListasEnlazadas.Lista.EliminarDelFrente()
  en BibliotecaHerenciaPila.HerenciaPila.Pop()
  en PruebaHerenciaPila.Main(String[] args)
  en C:\MisProyectos\Fig24_14\PruebaHerenciaPila\
  PruebaHerenciaPila.cs:línea 35

```

Figura 24.14 | Uso de la clase HerenciaPila. (Parte 2 de 2).

Clase de pila que contiene una referencia a una Lista

Otra manera de implementar una clase de pila es reutilizando una clase de lista a través de la composición. La clase en la figura 24.15 utiliza un objeto `private` de la clase `Lista` (línea 11) en la declaración de la clase `ComposicionPila`. La composición nos permite ocultar los métodos de la clase `Lista` que no deben estar en nuestra interfaz `public` de la pila, proporcionando los métodos de la interfaz `public` sólo a los métodos requeridos de `Lista`. Para implementar cada uno de los métodos de la pila, esta clase delega su trabajo a un método apropiado de `Lista`. Los métodos de `ComposicionPila` llaman a los métodos `InsertarAlFrente`, `EliminarDelFrente`, `EstaVacía` e `Imprimir` de `Lista`. En este ejemplo no mostramos la clase `PruebaComposicionPila`, ya que la única diferencia en este ejemplo es que modificamos el nombre de la clase de pila, de `HerenciaPila` a `ComposicionPila`. Si ejecuta la aplicación desde el código, podrá ver que los resultados son idénticos.

24.6 Colas

La cola es otra estructura de datos de uso común. Una cola es similar a una fila para pagar en un supermercado; el cajero atiende primero a la persona que está al principio de la fila. Otros clientes entran a la fila sólo por su parte final y esperan a que se les atienda. Los nodos de una cola se eliminan sólo desde el principio (o frente) de la misma y se insertan sólo al final de ésta. Por esta razón, una cola es una estructura de datos *PEPS* (*primero en entrar, primero en salir*). Las operaciones para insertar y eliminar se conocen como *enqueue* (agregar a la cola) y *dequeue* (retirar de la cola).

```

1  // Fig. 24.15: BibliotecaComposicionPila.cs
2  // Declaración de ComposicionPila con un objeto Lista compuesto.
3  using System;
4  using BibliotecaListasEnlazadas;
5
6  namespace BibliotecaComposicionPila
7  {
8      // la clase ComposicionPila encapsula las herramientas de Lista
9      public class ComposicionPila
10     {
11         private Lista pila;
12
13         // construye una pila vacía
14         public ComposicionPila()
15         {
16             pila = new Lista( "pila" );
17         } // fin del constructor
18
19         // agrega un objeto a la pila
20         public void Push( object valorDatos )
21         {
22             pila.InsertarAlFrente( valorDatos );
23         } // fin del método Push
24
25         // elimina un objeto de la pila
26         public object Pop()
27         {
28             return pila.EliminarDelFrente();
29         } // fin del método Pop
30
31         // determina si la pila está vacía
32         public bool EstaVacía()
33         {
34             return pila.EstaVacía();
35         } // fin del método EstaVacía
36
37         // imprime el contenido de la pila
38         public void Imprimir()
39         {
40             pila.Imprimir();
41         } // fin del método Imprimir
42     } // fin de la clase ComposicionPila
43 } // fin del espacio de nombres BibliotecaComposicionPila

```

Figura 24.15 | La clase *ComposicionPila* encapsula la funcionalidad de la clase *Lista*.

Las colas tienen muchas aplicaciones en los sistemas computacionales. La mayoría de las computadoras tienen sólo un procesador, por lo que sólo pueden atender una aplicación a la vez. Cada aplicación que requiere tiempo del procesador se coloca en una cola. La aplicación al frente de la cola es la siguiente que recibe atención. Cada aplicación avanza en forma gradual al frente de la cola, a medida que las aplicaciones al frente reciben atención.

Las colas también se utilizan para dar soporte al uso de la *cola de impresión*. Por ejemplo, una sola impresora puede compartirse entre todos los usuarios de la red. Muchos usuarios pueden enviar trabajos a la impresora, incluso cuando ésta ya se encuentre ocupada. Estos trabajos de impresión se colocan en una cola hasta que la impresora esté disponible. Un programa conocido como *spooler* administra la cola para asegurarse que, a medida que se complete cada trabajo de impresión, se envíe el siguiente trabajo a la impresora.

En las redes de computación, los paquetes de información también esperan en colas. Cada vez que un paquete llega a un nodo de la red, debe enrutarse hacia el siguiente nodo de la red a través de la ruta hacia el destino

final del paquete. El nodo enrutador envía un paquete a la vez, por lo que los paquetes adicionales se ponen en una cola hasta que el enrutador pueda enviarlos.

Un servidor de archivos en una red computacional se encarga de las solicitudes de acceso a los archivos de muchos clientes distribuidos en la red. Los servidores tienen una capacidad limitada para dar servicio a las solicitudes de los clientes. Cuando se excede esa capacidad, las solicitudes de los clientes esperan en colas.

Clase de pila que hereda de Lista

El programa de las figuras 24.16 y 24.17 crea una clase de cola, heredando de una clase de lista. Queremos que la clase *HerenciaCola* (figura 24.16) tenga los métodos *Enqueue*, *Dequeue*, *EstaVacia* e *Imprimir*. En esencia, éstos son los métodos *InsertarAlFinal*, *EliminarDelFrente*, *EstaVacia* e *Imprimir* de la clase *Lista*. Desde luego que la clase de lista contiene otros métodos (como *InsertarAlFrente* y *EliminarDelFinal*) que es preferible no hacer accesibles a través de la interfaz *public* de la clase de cola. Recuerde que todos los métodos en la interfaz *public* de la clase *Lista* son también métodos *public* de la clase derivada *HerenciaCola*.

La implementación de cada método *HerenciaCola* llama al método de *Lista* apropiado; el método *Enqueue* llama a *InsertarAlFinal* y el método *Dequeue* llama a *EliminarDelFrente*. Las llamadas a *EstaVacia* y a *Imprimir* invocan las versiones de la clase base que se heredaron de la clase *Lista* a la interfaz *public* de *HerenciaCola*. Observe que la clase *HerenciaCola* usa el espacio de nombres *BibliotecaListasEnlazadas* (figura 24.4); por ende, la biblioteca de clases para *HerenciaCola* debe tener una referencia a la biblioteca de clases *BibliotecaListasEnlazadas*.

El método *Main* de la clase *PruebaHerenciaCola* (figura 24.17) crea un objeto *HerenciaCola* llamado *cola*. Las líneas 15-18 definen cuatro valores que se meten y se sacan de la cola. El programa mete a la cola (líneas 21, 23, 25 y 27) un *bool* que contiene *true*, un *char* que contiene '\$', un *int* que contiene 34567 y un

```

1  // Fig. 24.16: BibliotecaHerenciaCola.cs
2  // Implementación de una cola, heredando de la clase Lista.
3  using System;
4  using BibliotecaListasEnlazadas;
5
6  namespace BibliotecaHerenciaCola
7  {
8      // la clase HerenciaCola hereda las herramientas de Lista
9      public class HerenciaCola : Lista
10     {
11         // pasa el nombre "cola" al constructor de Lista
12         public HerenciaCola()
13             : base( "cola" )
14         {
15         } // fin del constructor
16
17         // coloca valorDatos al final de la cola, insertando
18         // valorDatos al final de la lista enlazada
19         public void Enqueue( object valorDatos )
20         {
21             InsertarAlFinal( valorDatos );
22         } // fin del método Enqueue
23
24         // elimina elemento al principio de la cola, eliminando
25         // el elemento al principio de la lista enlazada
26         public object Dequeue()
27         {
28             return EliminarDelFrente();
29         } // fin del método Dequeue
30     } // fin de la clase HerenciaCola
31 } // fin del espacio de nombres BibliotecaHerenciaCola

```

Figura 24.16 | La clase *HerenciaCola* extiende a la clase *Lista*.

string que contiene "hola". Observe que la clase PruebaHerenciaCola utiliza el espacio de nombres BibliotecaListasEnlazadas y el espacio de nombres BibliotecaHerenciaCola; por ende, la solución para la clase PruebaHerenciaCola debe tener referencias a ambas bibliotecas de clases.

Un ciclo while infinito (líneas 36-41) saca los elementos de la cola en orden PEPS. Cuando no hay más objetos qué sacar, el método Dequeue lanza una ExcepcionListaVacía, y el programa muestra el rastreo de pila de la excepción, que a su vez muestra la pila de ejecución del programa, en el momento en que ocurrió la excepción. El programa utiliza el método Imprimir (heredado de la clase Lista) para imprimir en pantalla el contenido de la cola, después de cada operación. Observe que la clase PruebaHerenciaCola utiliza el espacio de nombres BibliotecaListasEnlazadas (figura 24.4) y el espacio de nombres BibliotecaHerenciaCola (figura 24.16); por ende, la solución para la clase PruebaHerenciaCola debe tener referencias a ambas bibliotecas de clases.

```

1  // Fig. 24.17: PruebaCola.cs
2  // Prueba de la clase HerenciaCola.
3  using System;
4  using BibliotecaHerenciaCola;
5  using BibliotecaListasEnlazadas;
6
7  // demuestra la funcionalidad de la clase HerenciaCola
8  class PruebaCola
9  {
10     static void Main( string[] args )
11     {
12         HerenciaCola cola = new HerenciaCola();
13
14         // crea objetos para almacenarlos en la cola
15         bool unBooleano = true;
16         char unCaracter = '$';
17         int unEntero = 34567;
18         string unaCadena = "hola";
19
20         // usa el método Enqueue para agregar elementos a la cola
21         cola.Enqueue( unBooleano );
22         cola.Imprimir();
23         cola.Enqueue( unCaracter );
24         cola.Imprimir();
25         cola.Enqueue( unEntero );
26         cola.Imprimir();
27         cola.Enqueue( unaCadena );
28         cola.Imprimir();
29
30         // usa el método Dequeue para eliminar elementos de la cola
31         object objetoEliminado = null;
32
33         // elimina elementos de la cola
34         try
35         {
36             while ( true )
37             {
38                 objetoEliminado = cola.Dequeue();
39                 Console.WriteLine( objetoEliminado + " se quitó de la cola" );
40                 cola.Imprimir();
41             } // fin de while
42         } // fin de try
43         catch ( ExcepcionListaVacía excepcionListaVacía )
44         {
45             // si ocurre una excepción, imprime el rastreo de pila

```

Figura 24.17 | Cola creada por herencia. (Parte I de 2).

```

46      Console.Error.WriteLine( excepcionListaVacía.StackTrace );
47      } // fin de catch
48  } // fin del método Main
49  } // fin de la clase PruebaCola

```

```

La cola es: True

La cola es: True $

La cola es: True $ 34567

La cola es: True $ 34567 hola

True se quitó de la cola
La cola es: $ 34567 hola

$ se quitó de la cola
La cola es: 34567 hola

34567 se quitó de la cola
La cola es: hola

hola se quitó de la cola
cola Vacía
    en BibliotecaListasEnlazadas.Lista.EliminarDelFrente()
    en BibliotecaHerenciaCola.HerenciaCola.Dequeue()
    en PruebaCola.Main(String[] args)
    en C:\MisProyectos\Fig24_17\PruebaCola\PruebaCola.cs: línea 38

```

Figura 24.17 | Cola creada por herencia. (Parte 2 de 2).

24.7 Árboles

Las listas enlazadas, las pilas y las colas son *estructuras de datos lineales* (es decir, *secuencias*). Un *árbol* es una estructura de datos bidimensional no lineal, con propiedades especiales. Los nodos de un árbol contienen dos o más enlaces.

Terminología básica

En esta sección hablaremos sobre los *árboles binarios* (figura 24.18): árboles cuyos nodos contienen dos enlaces (de los cuales uno, ambos o ninguno pueden ser `null`). El *nodo raíz* es el primer nodo en un árbol. Cada enlace en el nodo raíz hace referencia a un *hijo*. El *hijo izquierdo* es el primer nodo en el *subárbol izquierdo*, y el *hijo derecho* es el primer nodo en el *subárbol derecho*. Los hijos de un nodo específico se llaman *hermanos*. Un nodo sin hijos se llama *nodo hoja*. Por lo general, los científicos computacionales dibujan los árboles partiendo desde el nodo raíz, hacia abajo; exactamente lo opuesto a la manera en que crecen la mayoría de los árboles reales.



Error común de programación 24.2

Si no se establecen en `null` los enlaces en los nodos hoja de un árbol, se produce un error lógico.

Árbol binario de búsqueda

En nuestro ejemplo del árbol binario, creamos un árbol binario especial, conocido como *árbol binario de búsqueda*. Un árbol binario de búsqueda (sin valores duplicados en los nodos) tiene la característica de que los valores en cualquier subárbol izquierdo son menores que el valor en el *nodo padre* del subárbol, y los valores en cualquier subárbol derecho son mayores que el valor en el nodo padre del subárbol. La figura 24.19 muestra un árbol binario de búsqueda con 9 valores enteros. Hay que tener en cuenta que la figura del árbol binario de búsqueda que corresponde a un conjunto de datos puede depender del orden en el que se insertan los valores en el árbol.

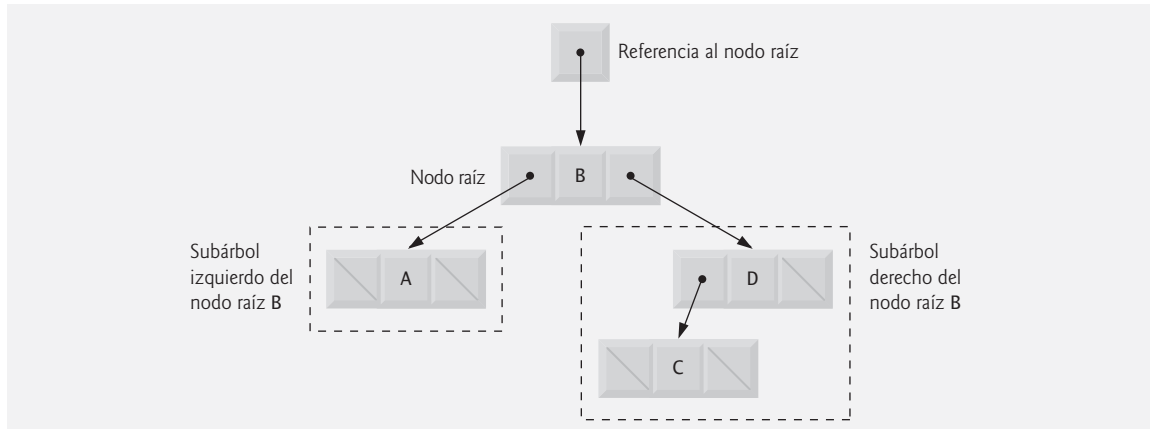


Figura 24.18 | Representación gráfica de un árbol binario.

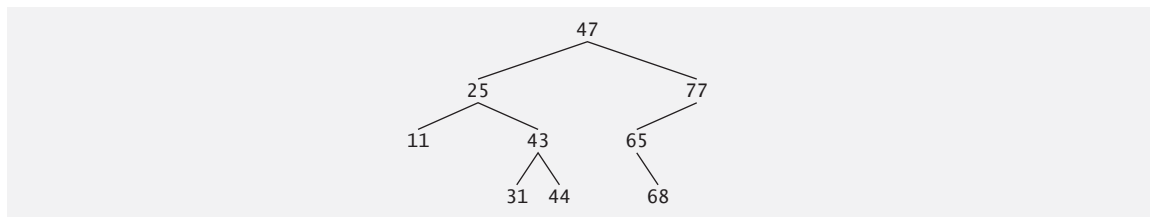


Figura 24.19 | Árbol binario de búsqueda que contiene 9 valores.

24.7.1 Árbol binario de búsqueda de valores enteros

La aplicación de las figuras 24.20 y 24.21 crea un árbol binario de búsqueda de enteros y lo recorre (es decir, avanza por todos sus nodos) de tres formas: mediante los recorridos recursivos *inorden*, *preorden* y *postorden*. El programa genera 10 números aleatorios e inserta cada uno de ellos en el árbol. La figura 24.20 define la clase `Arbol` en el espacio de nombres `BibliotecaArbolesBinarios` para fines de reutilizarla en posteriores ejemplos. La figura 24.21 define la clase `PruebaArbol` para demostrar la funcionalidad de la clase `Arbol`. El método `Main` de la clase `PruebaArbol` crea una instancia de un objeto `Arbol` vacío, después genera al azar 10 enteros e inserta cada valor en el árbol binario mediante una llamada al método `InsertarNodo` de la clase `Arbol`. Después el programa realiza los recorridos preorden, inorden y postorden del árbol. En breve hablaremos sobre estos recorridos.

La clase `NodoArbol` (líneas 8-81 de la figura 24.20) es una clase autorreferenciada que contiene tres miembros de datos `private`: `nodoIzquierdo` y `nodoDerecho` de tipo `NodoArbol`, y datos de tipo `int`. Al principio, todo `NodoArbol` es un nodo hoja, por lo que el constructor (líneas 15-19) inicializa las referencias `nodoIzquierdo` y `nodoDerecho` a `null`. Las propiedades `NodoIzquierdo` (líneas 22-32), `Datos` (líneas 35-45) y `NodoDerecho` (líneas 48-58) proporcionan acceso a los miembros de datos `private` de un `NodoArbol`. En breve hablaremos sobre el método `Insertar` de `NodoArbol` (líneas 62-80).

La clase `Arbol` (líneas 84-170 de la figura 24.20) manipula objetos de la clase `NodoArbol`. La clase `Arbol` tiene una raíz de datos `private` (línea 86): una referencia al nodo raíz del árbol. La clase contiene el método `public InsertarNodo` (líneas 97-103) para insertar un nuevo nodo en el árbol, y los métodos `public RecorridoPreorden` (líneas 106-109), `RecorridoInorden` (líneas 128-131) y `RecorridoPostorden` (líneas 150-153) para iniciar los recorridos del árbol. Cada uno de estos métodos llama a un método utilitario recursivo separado para realizar las operaciones de recorrido en la representación interna del árbol. El constructor de `Arbol` (líneas 89-92) inicializa `raiz` a `null` para indicar que al principio el árbol está vacío.

El método `InsertarNodo` de `Arbol` (líneas 97-103) determina primero si el árbol está vacío. De ser así, la línea 100 asigna un nuevo `NodoArbol`, inicializa el nodo con el entero que se insertará en el árbol y asigna ese nuevo nodo a `raiz`. Si el árbol no está vacío, `InsertarNodo` llama al método `Insertar` de `NodoArbol` (líneas 62-80), el cuál determina en forma recursiva la ubicación para el nuevo nodo en el árbol, e inserta el nodo en esa ubicación. *En un árbol binario, un nodo puede insertarse sólo como un nodo hoja.*

```

1  // Fig. 24.20: BibliotecaArbolesBinarios.cs
2  // Declaración de las clases NodoArbol y Arbol.
3  using System;
4
5  namespace BibliotecaArbolesBinarios
6  {
7      // declaración de la clase NodoArbol
8      class NodoArbol
9      {
10         private NodoArbol nodoIzquierdo; // enlace al hijo izquierdo
11         private int datos; // datos almacenados en el nodo
12         private NodoArbol nodoDerecho; // enlace al hijo derecho
13
14         // inicializa datos del nodo y lo convierte en nodo hoja
15         public NodoArbol( int datosNodo )
16         {
17             datos = datosNodo;
18             nodoIzquierdo = nodoDerecho = null; // el nodo no tiene hijos
19         } // fin del constructor
20
21         // propiedad NodoIzquierdo
22         public NodoArbol NodoIzquierdo
23         {
24             get
25             {
26                 return nodoIzquierdo;
27             } // fin de get
28             set
29             {
30                 nodoIzquierdo = value;
31             } // fin de set
32         } // fin de la propiedad NodoIzquierdo
33
34         // propiedad Datos
35         public int Datos
36         {
37             get
38             {
39                 return datos;
40             } // fin de get
41             set
42             {
43                 datos = value;
44             } // fin de set
45         } // fin de la propiedad Datos
46
47         // propiedad NodoDerecho
48         public NodoArbol NodoDerecho
49         {
50             get
51             {
52                 return nodoDerecho;
53             } // fin de get
54             set
55             {
56                 nodoDerecho = value;
57             } // fin de set
58         } // fin de la propiedad NodoDerecho
59     }

```

Figura 24.20 | Las clases `NodoArbol` y `Arbol` para un árbol binario de búsqueda. (Parte I de 3).

```

60 // inserta NodoArbol en el Arbol que contiene nodos;
61 // ignora los valores duplicados
62 public void Insertar( int valorInsercion )
63 {
64     if ( valorInsercion < datos ) // inserta en el subárbol izquierdo
65     {
66         // inserta nuevo NodoArbol
67         if ( nodoIzquierdo == null )
68             nodoIzquierdo = new NodoArbol( valorInsercion );
69         else // continúa recorriendo el subárbol izquierdo
70             nodoIzquierdo.Insertar( valorInsercion );
71     } // fin de if
72     else if ( valorInsercion > datos ) // inserta en el subárbol derecho
73     {
74         // inserta nuevo NodoArbol
75         if ( nodoDerecho == null )
76             nodoDerecho = new NodoArbol( valorInsercion );
77         else // continúa recorriendo el subárbol derecho
78             nodoDerecho.Insertar( valorInsercion );
79     } // fin de else if
80 } // fin del método Insertar
81 } // fin de la clase NodoArbol
82
83 // declaración de la clase Arbol
84 public class Arbol
85 {
86     private NodoArbol raiz;
87
88     // construye un Arbol de enteros vacío
89     public Arbol()
90     {
91         raiz = null;
92     } // fin del constructor
93
94     // Inserta un nuevo nodo en el árbol binario de búsqueda.
95     // Si el nodo raíz es null, crea el nodo raíz aquí.
96     // En cualquier otro caso, llama al método Insertar de la clase NodoArbol.
97     public void InsertarNodo( int valorInsercion )
98     {
99         if ( raiz == null )
100             raiz = new NodoArbol( valorInsercion );
101         else
102             raiz.Insertar( valorInsercion );
103     } // fin del método InsertarNodo
104
105     // empieza el recorrido preorden
106     public void RecorridoPreorden()
107     {
108         AyudantePreorden( raiz );
109     } // fin del método RecorridoPreorden
110
111     // método recursivo para realizar el recorrido preorden
112     private void AyudantePreorden( NodoArbol nodo )
113     {
114         if ( nodo == null )
115             return;
116
117         // imprime los datos del nodo
118         Console.Write( nodo.Datos + " " );

```

Figura 24.20 | Las clases `NodoArbol` y `Arbol` para un árbol binario de búsqueda. (Parte 2 de 3).

```

119
120     // recorre el subárbol izquierdo
121     AyudantePreorden( nodo.NodoIzquierdo );
122
123     // recorre el subárbol derecho
124     AyudantePreorden( nodo.NodoDerecho );
125 } // fin del método AyudantePreorden
126
127 // empieza recorrido inorden
128 public void RecorridoInorden()
129 {
130     AyudanteInorden( raiz );
131 } // fin del método RecorridoInorden
132
133 // método recursivo para realizar el recorrido inorden
134 private void AyudanteInorden( NodoArbol nodo )
135 {
136     if ( nodo == null )
137         return;
138
139     // recorre el subárbol izquierdo
140     AyudanteInorden( nodo.NodoIzquierdo );
141
142     // imprime datos del nodo
143     Console.Write( nodo.Datos + " " );
144
145     // recorre el subárbol derecho
146     AyudanteInorden( nodo.NodoDerecho );
147 } // fin del método AyudanteInorden
148
149 // empieza recorrido postorden
150 public void RecorridoPostorden()
151 {
152     AyudantePostorden( raiz );
153 } // fin del método RecorridoPostorden
154
155 // método recursivo para realizar el recorrido postorden
156 private void AyudantePostorden( NodoArbol nodo )
157 {
158     if ( nodo == null )
159         return;
160
161     // recorre el subárbol izquierdo
162     AyudantePostorden( nodo.NodoIzquierdo );
163
164     // recorre el subárbol derecho
165     AyudantePostorden( nodo.NodoDerecho );
166
167     // imprime datos del nodo
168     Console.Write( nodo.Datos + " " );
169 } // fin del método AyudantePostorden
170 } // fin de la clase Arbol
171 } // fin del espacio de nombres BibliotecaArbolesBinarios

```

Figura 24.20 | Las clases `NodoArbol` y `Arbol` para un árbol binario de búsqueda. (Parte 3 de 3).

El método `Insertar` de `NodoArbol` compara el valor a insertar con el valor de datos en el nodo raíz. Si el valor a insertar es menor que los datos del nodo raíz, el programa determina si el subárbol izquierdo está vacío (línea 67). De ser así, la línea 68 asigna un nuevo `NodoArbol`, lo inicializa con el entero que se insertará y asigna el nuevo nodo a la referencia `nodoIzquierdo`. En caso contrario, la línea 70 hace una llamada recursiva a `Insertar`

```

1  // Fig. 24.21: PruebaArbol.cs
2  // Este programa prueba la clase Arbol.
3  using System;
4  using BibliotecaArbolesBinarios;
5
6  // declaración de la clase PruebaArbol
7  public class PruebaArbol
8  {
9      // prueba la clase Arbol
10     static void Main( string[] args )
11     {
12         Arbol arbol = new Arbol();
13         int valorInsercion;
14
15         Console.WriteLine( "Insertando valores: " );
16         Random aleatorio = new Random();
17
18         // inserta 10 enteros aleatorios del 0-99 en el árbol
19         for ( int i = 1; i <= 10; i++ )
20         {
21             valorInsercion = aleatorio.Next( 100 );
22             Console.Write( valorInsercion + " " );
23
24             arbol.InsertarNodo( valorInsercion );
25         } // fin de for
26
27         // realiza recorrido preorden del árbol
28         Console.WriteLine( "\n\nRecorrido preorden" );
29         arbol.RecorridoPreorden();
30
31         // realiza recorrido inorden del árbol
32         Console.WriteLine( "\n\nRecorrido inorden" );
33         arbol.RecorridoInorden();
34
35         // realiza recorrido postorden del árbol
36         Console.WriteLine( "\n\nRecorrido postorden" );
37         arbol.RecorridoPostorden();
38     } // fin del método Main
39 } // fin de la clase PruebaArbol

```

```

Insertando valores:
55 14 1 45 71 62 17 23 99 61

Recorrido preorden
55 14 1 45 17 23 71 62 61 99

Recorrido inorden
1 14 17 23 45 55 61 62 71 99

Recorrido postorden
1 23 17 45 14 61 62 99 71 55

```

Figura 24.21 | Creación y recorrido de un árbol binario.

para que el subárbol izquierdo inserte el valor en el subárbol izquierdo. Si el valor a insertar es mayor que los datos del nodo raíz, el programa determina si el subárbol derecho está vacío (línea 75). De ser así, la línea 76 asigna un nuevo `NodoArbol`, lo inicializa con el entero que se insertará y asigna el nuevo nodo a la referencia `nodoDerecho`. En caso contrario, la línea 78 hace una llamada recursiva a `Insertar` para que el subárbol derecho inserte el valor en el subárbol derecho.

Los métodos `RecorridoInorden`, `RecorridoPreorden` y `RecorridoPostorden` llaman a los métodos ayudantes `AyudanteInorden` (líneas 134-147), `AyudantePreorden` (líneas 112-125) y `AyudantePostorden` (líneas 156-169), respectivamente, para recorrer el árbol e imprimir los valores de los nodos. El propósito de los métodos ayudantes en la clase `Arbol` es permitir que el programador inicie un recorrido sin necesidad de obtener una referencia al nodo raíz primero, para después llamar al método recursivo con esa referencia. Los métodos `RecorridoInorden`, `RecorridoPreorden` y `RecorridoPostorden` sólo reciben la variable `private raíz` y la pasan al método ayudante apropiado, para iniciar un recorrido del árbol. Para la siguiente discusión, utilizaremos el árbol binario de búsqueda que se muestra en la figura 24.22.

Algoritmo de recorrido inorden

El método `AyudanteInorden` (líneas 134-147) define los pasos para un recorrido inorden. Estos pasos son:

1. Si el argumento es `null`, regresa de inmediato.
2. Recorre el subárbol izquierdo con una llamada a `AyudanteInorden` (línea 140).
3. Procesa el valor en el nodo (línea 143).
4. Recorre el subárbol derecho con una llamada a `AyudanteInorden` (línea 146).

El recorrido inorden no procesa el valor en un nodo sino hasta que se procesan los valores en el subárbol izquierdo de ese nodo. El recorrido inorden del árbol en la figura 24.22 es

6 13 17 27 33 42 48

Observe que el recorrido inorden de un árbol binario de búsqueda imprime los valores de los nodos en orden ascendente. El proceso de crear un árbol binario de búsqueda en realidad ordena los datos (cuando se acopla con un recorrido inorden); por ende, a este proceso se le conoce como *ordenamiento de un árbol binario*.

Algoritmo de recorrido preorden

El método `AyudantePreorden` (líneas 112-125) define los pasos para un recorrido preorden. Estos pasos son:

1. Si el argumento es `null`, regresa de inmediato.
2. Procesa el valor en el nodo (línea 118).
3. Recorre el subárbol izquierdo con una llamada a `AyudantePreorden` (línea 121).
4. Recorre el subárbol derecho con una llamada a `AyudantePreorden` (línea 124).

El recorrido preorden procesa el valor en cada uno de los nodos, a medida que se van visitando. Después de procesar el valor en un nodo dado, el recorrido preorden procesa los valores en el subárbol izquierdo, y después los valores en el subárbol derecho. El recorrido preorden del árbol en la figura 24.22 es:

27 13 6 17 42 33 48

Algoritmo de recorrido postorden

El método `AyudantePostorden` (líneas 156-169) define los pasos para un recorrido postorden. Estos pasos son:

1. Si el argumento es `null`, regresa de inmediato.
2. Recorre el subárbol izquierdo con una llamada a `AyudantePostorden` (línea 162).

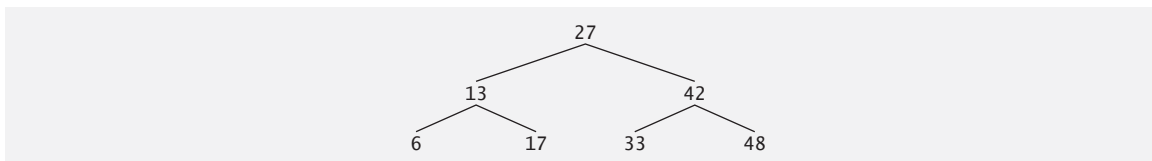


Figura 24.22 | Árbol binario de búsqueda.

3. Recorre el subárbol derecho con una llamada a `AyudantePostorden` (línea 165).
4. Procesa el valor en el nodo (línea 168).

El recorrido postorden procesa el valor en cada nodo, una vez que se han procesado los valores de todos los nodos hijos. El recorrido postorden del árbol en la figura 24.22 es:

6 17 13 33 48 42 27

Eliminación de duplicados

El árbol binario de búsqueda facilita la *eliminación de duplicados*. Al construir un árbol, la operación de inserción reconoce los intentos por insertar un valor duplicado, ya que éste sigue las mismas decisiones de “ir a la izquierda” o “ir a la derecha” en cada comparación, como lo hizo el valor original. Por ende, la operación de inserción compara en un momento dado el valor duplicado con un nodo que contiene el mismo valor. En este punto, la operación de inserción podría simplemente descartar el valor duplicado.

El proceso de buscar en un árbol binario un valor que coincida con un valor clave es rápido, en especial para los árboles binarios *estrechamente empaquetados*. En un árbol binario estrechamente empaquetado, cada nivel contiene una cantidad aproximada de hasta el doble de elementos que el nivel anterior. La figura 24.22 es un árbol binario estrechamente empaquetado. Un árbol binario de búsqueda con n elementos tiene un mínimo de $\log_2 n$ niveles. Por lo tanto, cuando mucho se requieren $\log_2 n$ comparaciones, ya sea para encontrar una coincidencia o para determinar que no existe ninguna. Para buscar en un árbol binario de búsqueda de 1000 elementos (estrechamente empaquetado) se requieren cuando mucho 10 comparaciones, ya que $2^{10} > 1000$. Para buscar en un árbol binario de búsqueda de 1,000,000 elementos (estrechamente empaquetado) se requieren a lo más 20 comparaciones, ya que $2^{20} > 1,000,000$.

Generalidades acerca de los ejercicios con árboles binarios

Los ejercicios del capítulo presentan algoritmos para otras operaciones con árboles binarios, como eliminar un elemento de un árbol binario y realizar un *recorrido en orden de niveles de un árbol binario*. El recorrido en orden de niveles de un árbol binario visita los nodos del árbol fila por fila, empezando en el nivel del nodo raíz. En cada nivel del árbol, un recorrido por orden de nivel visita los nodos de izquierda a derecha.

24.7.2 Árbol binario de búsqueda de objetos `IComparable`

El ejemplo del árbol binario en la sección 24.7.1 funciona bien cuando todos los datos son de tipo `int`. Suponga que desea manipular un árbol binario de valores enteros dobles. Podría reescribir las clases `NodoArbol` y `Arbol` con distintos nombres, y personalizarlas para manipular valores `double`. De manera similar, para cada tipo de datos podría crear versiones personalizadas de las clases `NodoArbol` y `Arbol`. Esto resulta en la proliferación de código, que se volvería difícil de manejar y mantener.

Lo ideal sería definir la funcionalidad de un árbol binario una sola vez, y reutilizar esa funcionalidad para muchos tipos de datos. Los lenguajes como Java™ y C# cuentan con herramientas de polimorfismo, las cuales permiten manipular todos los objetos de manera uniforme. Mediante el uso de dichas herramientas, podemos diseñar una estructura de datos más flexible. La nueva versión de C#, C# 2.0, proporciona estas herramientas mediante los genéricos (capítulo 25).

En nuestro siguiente ejemplo, aprovecharemos las herramientas polimórficas de C#, implementando clases `NodoArbol` y `Arbol` para manipular objetos de cualquier tipo que implemente a la interfaz ***`IComparable`*** (espacio de nombres `System`). Es imperativo poder comparar los objetos almacenados en una búsqueda binaria, para poder determinar la ruta al punto de inserción de un nuevo nodo. Las clases que implementan a `IComparable` definen el método ***`CompareTo`***, que compara el objeto que invocó al método con el objeto que recibe ese método como argumento. El método devuelve un valor `int` menor que cero si el objeto que hizo la llamada es menor que el objeto argumento, cero si los objetos son iguales y un valor positivo si el objeto que hizo la llamada es más grande que el objeto argumento. Además, tanto el objeto que hizo la llamada como los objetos argumentos deben ser del mismo tipo; en caso contrario, el método lanza una excepción `ArgumentException`.

El programa de las figuras 24.23 y 24.24 agrega características al programa de la sección 24.7.1 para manipular objetos `IComparable`. Una restricción en las nuevas versiones de las clases `NodoArbol` y `Arbol` en la figura 24.23 es que cada objeto `Arbol` puede contener objetos de sólo un tipo de datos (por ejemplo, que todos sean

string o que todos sean double). Si un programa trata de insertar varios tipos de datos en el mismo objeto Arbol, se producirán excepciones ArgumentException. Sólo modificamos seis líneas de código en la clase NodoArbol (líneas 12, 16, 36, 63, 65 y 73) y una línea de código en la clase Arbol (línea 98) para habilitar el procesamiento de objetos IComparable. Con la excepción de las líneas 65 y 73, todas las demás modificaciones simplemente sustituyen el tipo int con el tipo IComparable. Las líneas 65 y 73 anteriormente usaban los operadores < y > para comparar el valor a insertar con el valor en un nodo dado. Ahora estas líneas comparan objetos IComparable mediante el método CompareTo de la interfaz, y después evalúan el valor de retorno del método para determinar si es menor que cero (el objeto que hizo la llamada es menor que el objeto argumento) o mayor que cero (el objeto que hizo la llamada es mayor que el objeto argumento). [Nota: si esta clase se hubiera escrito con genéricos, el tipo de datos, int o IComparable podría haberse sustituido en tiempo de compilación por cualquier otro tipo que implemente los operadores y métodos necesarios.]

```

1  // Fig. 24.23: BibliotecaArbolesBinarios2.cs
2  // Declaración de las clases NodoArbol y Arbol para
3  // objetos IComparable.
4  using System;
5
6  namespace BibliotecaArbolesBinarios2
7  {
8      // declaración de la clase NodoArbol
9      class NodoArbol
10     {
11         private NodoArbol nodoIzquierdo; // enlace al hijo izquierdo
12         private IComparable datos; // datos almacenados en el nodo
13         private NodoArbol nodoDerecho; // enlace al subárbol derecho
14
15         // inicializa los datos en este nodo y lo convierte en nodo hoja
16         public NodoArbol( IComparable datosNodo )
17         {
18             datos = datosNodo;
19             nodoIzquierdo = nodoDerecho = null; // el nodo no tiene hijos
20         } // fin del constructor
21
22         // propiedad NodoIzquierdo
23         public NodoArbol NodoIzquierdo
24         {
25             get
26             {
27                 return nodoIzquierdo;
28             } // fin de get
29             set
30             {
31                 nodoIzquierdo = value;
32             } // fin de set
33         } // fin de la propiedad NodoIzquierdo
34
35         // propiedad Datos
36         public IComparable Datos
37         {
38             get
39             {
40                 return datos;
41             } // fin de get
42             set
43             {
44                 datos = value;

```

Figura 24.23 | Las clases NodoArbol y Arbol para manipular objetos IComparable. (Parte I de 4).

```

45     } // fin de set
46 } // fin de la propiedad Datos
47
48 // propiedad NodoDerecho
49 public NodoArbol NodoDerecho
50 {
51     get
52     {
53         return nodoDerecho;
54     } // fin de get
55     set
56     {
57         nodoDerecho = value;
58     } // fin de set
59 } // fin de la propiedad NodoDerecho
60
61 // inserta NodoArbol en Arbol que contiene nodos;
62 // ignora los valores duplicados
63 public void Insertar( IComparable valorInsercion )
64 {
65     if ( valorInsercion.CompareTo( datos ) < 0 )
66     {
67         // inserta en subárbol izquierdo
68         if ( nodoIzquierdo == null )
69             nodoIzquierdo = new NodoArbol( valorInsercion );
70         else // continúa recorriendo el subárbol izquierdo
71             nodoIzquierdo.Insertar( valorInsercion );
72     } // fin de if
73     else if ( valorInsercion.CompareTo( datos ) > 0 )
74     {
75         // inserta en subárbol derecho
76         if ( nodoDerecho == null )
77             nodoDerecho = new NodoArbol( valorInsercion );
78         else // continúa recorriendo el subárbol derecho
79             nodoDerecho.Insertar( valorInsercion );
80     } // fin de else if
81 } // fin del método Insertar
82 } // fin de la clase NodoArbol
83
84 // declaración de la clase Arbol
85 public class Arbol
86 {
87     private NodoArbol raiz;
88
89     // construye un Arbol de enteros vacío
90     public Arbol()
91     {
92         raiz = null;
93     } // fin del constructor
94
95     // Inserta un nuevo nodo en el árbol binario de búsqueda.
96     // Si el nodo raíz es null, lo crea aquí.
97     // En cualquier otro caso, llama al método Insertar de la clase NodoArbol.
98     public void InsertarNodo( IComparable valorInsercion )
99     {
100         if ( raiz == null )
101             raiz = new NodoArbol( valorInsercion );
102         else
103             raiz.Insertar( valorInsercion );

```

Figura 24.23 | Las clases `NodoArbol` y `Arbol` para manipular objetos `IComparable`. (Parte 2 de 4).

```

104     } // fin del método InsertarNodo
105
106     // empieza recorrido preorden
107     public void RecorridoPreorden()
108     {
109         AyudantePreorden( raiz );
110     } // fin del método RecorridoPreorden
111
112     // método recursivo para realizar el recorrido preorden
113     private void AyudantePreorden( NodoArbol nodo )
114     {
115         if ( nodo == null )
116             return;
117
118         // imprime datos del nodo
119         Console.Write( nodo.Datos + " " );
120
121         // recorre subárbol izquierdo
122         AyudantePreorden( nodo.NodoIzquierdo );
123
124         // recorre subárbol derecho
125         AyudantePreorden( nodo.NodoDerecho );
126     } // fin del método AyudantePreorden
127
128     // empieza recorrido inorden
129     public void RecorridoInorden()
130     {
131         AyudanteInorden( raiz );
132     } // fin del método RecorridoInorden
133
134     // método recursivo para realizar recorrido inorden
135     private void AyudanteInorden( NodoArbol nodo )
136     {
137         if ( nodo == null )
138             return;
139
140         // recorre subárbol izquierdo
141         AyudanteInorden( nodo.NodoIzquierdo );
142
143         // imprime datos del nodo
144         Console.Write( nodo.Datos + " " );
145
146         // recorre subárbol derecho
147         AyudanteInorden( nodo.NodoDerecho );
148     } // fin del método AyudanteInorden
149
150     // empieza recorrido postorden
151     public void RecorridoPostorden()
152     {
153         AyudantePostorden( raiz );
154     } // fin del método RecorridoPostorden
155
156     // método recursivo para realizar recorrido postorden
157     private void AyudantePostorden( NodoArbol nodo )
158     {
159         if ( nodo == null )
160             return;
161
162         // recorre subárbol izquierdo

```

Figura 24.23 | Las clases `NodoArbol` y `Arbol` para manipular objetos `IComparable`. (Parte 3 de 4).

```

163         AyudantePostorden( nodo.NodoIzquierdo );
164
165         // recorre subárbol derecho
166         AyudantePostorden( nodo.NodoDerecho );
167
168         // imprime datos del nodo
169         Console.Write( nodo.Datos + " " );
170     } // fin del método AyudantePostorden
171
172 } // fin de la clase Arbol
173 } // fin del espacio de nombres BibliotecaArbolesBinarios2

```

Figura 24.23 | Las clases `NodoArbol` y `Arbol` para manipular objetos `IComparable`. (Parte 4 de 4).

La clase `PruebaArbol` (figura 24.24) crea tres objetos `Arbol` para almacenar valores `int`, `double` y `string`; y el .NET Framework los define como tipos `IComparable`. El programa llena los árboles con los valores en los arreglos `arregloInt` (línea 12), `arregloDouble` (líneas 13-14) y `arregloString` (líneas 15-16), respectivamente.

```

1  // Fig. 24.24: PruebaArbol.cs
2  // Este programa prueba la clase Arbol.
3  using System;
4  using BibliotecaArbolesBinarios2;
5
6  // declaración de la clase PruebaArbol
7  public class PruebaArbol
8  {
9      // prueba la clase Arbol
10     static void Main( string[] args )
11     {
12         int[] arregloInt = { 8, 2, 4, 3, 1, 7, 5, 6 };
13         double[] arregloDouble =
14             { 8.8, 2.2, 4.4, 3.3, 1.1, 7.7, 5.5, 6.6 };
15         string[] arregloString = { "ocho", "dos", "cuatro",
16             "tres", "uno", "siete", "cinco", "seis" };
17
18         // crea Arbol int
19         Arbol arbolInt = new Arbol();
20         llenarArbol( arregloInt, arbolInt, "arbolInt" );
21         recorrerArbol( arbolInt, "arbolInt" );
22
23         // crea Arbol double
24         Arbol arbolDouble = new Arbol();
25         llenarArbol( arregloDouble, arbolDouble, "arbolDouble" );
26         recorrerArbol( arbolDouble, "arbolDouble" );
27
28         // crea Arbol string
29         Arbol arbolString = new Arbol();
30         llenarArbol( arregloString, arbolString, "arbolString" );
31         recorrerArbol( arbolString, "arbolString" );
32     } // fin de Main
33
34     // llena el Arbol con elementos del arreglo
35     static void llenarArbol( Array arreglo, Arbol arbol, string nombre )
36     {
37         Console.WriteLine( "\n\nInsertando en " + nombre + ":" );
38

```

Figura 24.24 | Demostración de la clase `Arbol` con objetos `IComparable`. (Parte 1 de 2).

```

39     foreach ( IComparable datos in arreglo )
40     {
41         Console.Write( datos + " " );
42         arbol.InsertarNodo( datos );
43     } // fin de foreach
44 } // fin del método llenarArbol
45
46 // realiza los recorridos
47 static void recorrerArbol( Arbol arbol, string tipoArbol )
48 {
49     // realiza recorrido preorden del árbol
50     Console.WriteLine( "\n\nRecorrido preorden de " + tipoArbol );
51     arbol.RecorridoPreorden();
52
53     // realiza recorrido inorden del árbol
54     Console.WriteLine( "\n\nRecorrido inorden de " + tipoArbol );
55     arbol.RecorridoInorden();
56
57     // realiza recorrido postorden del árbol
58     Console.WriteLine( "\n\nRecorrido postorden de " + tipoArbol );
59     arbol.RecorridoPostorden();
60 } // fin del método recorrerArbol
61 } // fin de la clase PruebaArbol

```

Insertando en arbolInt:

8 2 4 3 1 7 5 6

Recorrido preorden de arbolInt

8 2 1 4 3 7 5 6

Recorrido inorden de arbolInt

1 2 3 4 5 6 7 8

Recorrido postorden de arbolInt

1 3 6 5 7 4 2 8

Insertando en arbolDouble:

8.8 2.2 4.4 3.3 1.1 7.7 5.5 6.6

Recorrido preorden de arbolDouble

8.8 2.2 1.1 4.4 3.3 7.7 5.5 6.6

Recorrido inorden de arbolDouble

1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8

Recorrido postorden de arbolDouble

1.1 3.3 6.6 5.5 7.7 4.4 2.2 8.8

Insertando en arbolString:

ocho dos cuatro tres uno siete cinco seis

Recorrido preorden de arbolString

ocho dos cuatro cinco tres siete seis uno

Recorrido inorden de arbolString

ocho cinco cuatro uno siete seis tres dos

Recorrido postorden de arbolString

cinco seis siete uno tres cuatro dos ocho

Figura 24.24 | Demostración de la clase Arbol con objetos IComparable. (Parte 2 de 2).

El método `llenarArbol` (líneas 35-44) recibe como argumentos un objeto `Array` que contiene los valores inicializadores para el `Arbol`, un `Arbol` en el que se colocarán los elementos y un `string` que representa el nombre del `Arbol`; después inserta cada elemento del objeto `Array` en el `Arbol`. El método `recorrerArbol` (líneas 47-60) recibe como argumentos un `Arbol` y un `string` que representa el nombre del `Arbol`; después imprime en pantalla los recorridos preorden, inorden y postorden del `Arbol`. Observe que el recorrido inorden de cada `Arbol` imprime en pantalla los datos en orden, sin importar el tipo de datos almacenado en el `Arbol`. Nuestra implementación polimórfica de la clase `Arbol` invoca al método `CompareTo` del tipo de datos apropiado para determinar la ruta hacia el punto de inserción de cada valor, usando las reglas de inserción estándar para árboles binarios de búsqueda. Además, observe que el `Arbol` de objetos `string` aparece en orden alfabético.

24.8 Conclusión

En este capítulo aprendió que los tipos simples son tipos `struct` por valor, pero de todas formas pueden usarse en cualquier parte en la que se esperen objetos `object` en un programa, debido a las conversiones `boxing` y `unboxing`. Aprendió que las listas enlazadas son colecciones de elementos de datos que se “enlazan entre sí en una cadena”. También aprendió que un programa puede realizar inserciones y eliminaciones en cualquier parte de una lista enlazada (aunque nuestra implementación sólo realiza inserciones y eliminaciones en los extremos de la lista). También demostramos que las estructuras de datos tipo pila y cola son versiones restringidas de las listas. Para las pilas, vimos que las inserciones y eliminaciones se realizan sólo en la parte superior; es por ello que las pilas se conocen como estructuras de datos `UEPS` (último en entrar, primero en salir). Para las colas, que representan filas de espera, vimos que las inserciones se realizan en la parte final y las eliminaciones en la parte inicial; es por ello que las colas se conocen como estructuras de datos `PEPS` (primero en entrar, primero en salir). Después presentamos la estructura de datos tipo árbol binario. Vimos un árbol binario de búsqueda que facilita las búsquedas de alta velocidad y la ordenación de los datos, además de una eficiente eliminación de valores duplicados. En el siguiente capítulo presentaremos los genéricos, que le permiten declarar una familia de clases y métodos que implementan la misma funcionalidad sobre cualquier tipo.

25

Genéricos

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Crear métodos genéricos que realicen tareas idénticas con argumentos de distintos tipos.
- Crear una clase `Pila` genérica que pueda usarse para almacenar objetos de cualquier tipo de clase o de interfaz.
- Comprender cómo sobrecargar métodos genéricos con métodos no genéricos, o con otros métodos genéricos.
- Comprender la restricción `new()` de un parámetro de tipo.
- Aplicar múltiples restricciones a un parámetro de tipo.
- Comprender la relación entre los genéricos y la herencia.

*...nuestra individualidad especial,
según se distingue desde nuestra
genérica humanidad.*

—Oliver Wendell Holmes, Sr.

*Todo hombre de genio ve el
mundo desde un ángulo distinto
al de sus semejantes.*

—Havelock Ellis

*Nacido bajo una ley, hacia
otro límite.*

—Lord Brooke

- 25.1 Introducción
- 25.2 Motivación para los métodos genéricos
- 25.3 Implementación de un método genérico
- 25.4 Restricciones de los tipos
- 25.5 Sobrecarga de métodos genéricos
- 25.6 Clases genéricas
- 25.7 Observaciones acerca de los genéricos y la herencia
- 25.8 Conclusión

25.1 Introducción

En el capítulo 24 presentamos estructuras de datos que almacenan y manipulan referencias `object`. Podríamos almacenar cualquier objeto `object` en nuestras estructuras de datos. Un aspecto inconveniente de almacenar referencias `object` se produce cuando se obtienen de una colección. Por lo general, una aplicación necesita procesar tipos específicos de objetos. Como resultado, las referencias `object` que se obtienen de una colección generalmente necesitan de una conversión descendente a un tipo apropiado, para permitir que la aplicación procese los objetos en forma correcta. Además, se deben realizar conversiones `boxing` con los datos de tipos de valores (por ejemplo, `int` y `double`) para manipularlos con referencias `object`, lo cual incrementa la sobrecarga de procesar tales datos. Aparte de eso, si se procesan todos los datos como de tipo `object`, se limita la habilidad del compilador de C# para realizar la comprobación de tipos.

Aunque podemos crear fácilmente estructuras de datos para manipular cualquier tipo de datos como objetos `object` (como hicimos en el capítulo 24), sería bueno si pudiéramos detectar los conflictos de tipo en tiempo de compilación; a esto se le conoce como *seguridad de tipos en tiempo de ejecución*. Por ejemplo, si una `Pila` sólo debe almacenar valores `int`, al tratar de meter un `string` en esa `Pila` se produciría un error en tiempo de compilación. De manera similar, un método `Ordenar` debería ser capaz de comparar elementos en los que se garantice que todos son del mismo tipo. Si creamos versiones de tipo específico de la clase `Pila` y del método `Ordenar`, el compilador C# sin duda podría hacer valer la seguridad de los tipos en tiempo de compilación. Sin embargo, esto requeriría de la creación de muchas copias del mismo código básico.

En este capítulo hablaremos sobre una de las características más recientes de C#: los *genéricos*, que proporcionan los medios para crear los modelos generales antes mencionados. Los *métodos genéricos* nos permiten especificar, mediante la declaración de un solo método, un conjunto de métodos relacionados. Las *clases genéricas* nos permiten especificar, mediante la declaración de una sola clase, un conjunto de clases relacionadas. Asimismo, las *interfaces genéricas* nos permiten especificar, mediante la declaración de una sola interfaz, un conjunto de interfaces relacionadas. Los genéricos proporcionan seguridad de tipos en tiempo de compilación. [Nota: También podemos implementar tipos `struct` y `delegate` genéricos. Para más información, consulte la especificación del lenguaje C#.]

Podemos escribir un método genérico para ordenar un arreglo de objetos y después invocar al método genérico por separado con un arreglo `int`, un arreglo `double`, un arreglo `string` y así sucesivamente, para ordenar cada tipo distinto de arreglo. El compilador realiza la *comprobación de tipos* para asegurar que el arreglo que se pasa al método de ordenamiento contenga sólo elementos del mismo tipo. Podemos escribir una sola clase `Pila` genérica que manipule una pila de objetos, y después crear instancias de objetos `Stack` para una pila de valores `int`, una pila de valores `double`, una pila de valores `string`, y así sucesivamente. El compilador realiza la comprobación de tipos para asegurar que la `Pila` almacene sólo elementos del mismo tipo.

En este capítulo presentaremos ejemplos de clases y métodos genéricos. También consideraremos las relaciones entre los genéricos y otras características de C#, como la sobrecarga y la herencia. El capítulo 26, Colecciones, habla sobre las clases de colecciones genéricas y no genéricas del .NET Framework. Una colección es una estructura de datos que mantiene un grupo de objetos o valores relacionados. Las clases de colecciones del .NET Framework utilizan genéricos para permitirnos especificar los tipos exactos de objetos que debe almacenar una colección específica.

25.2 Motivación para los métodos genéricos

A menudo, los métodos sobrecargados se utilizan para realizar operaciones similares con distintos tipos de datos. Para motivar los métodos genéricos, empecemos con un ejemplo (figura 25.1) que contiene tres métodos `ImprimirArreglo` sobrecargados (líneas 23-29, 32-38 y 41-47). Estos métodos muestran los elementos de un arreglo `int`, un arreglo `double` y un arreglo `char`, respectivamente. Pronto reimplementaremos este programa de una manera más concisa y elegante, mediante el uso de un solo método genérico.

```

1  // Fig. 25.1: MetodosSobrecargados.cs
2  // Uso de métodos sobrecargados para imprimir arreglos de distintos tipos.
3  using System;
4
5  class MetodosSobrecargados
6  {
7      static void Main( string[] args )
8      {
9          // crea arreglos de valores int, double y char
10         int[] arregloInt = { 1, 2, 3, 4, 5, 6 };
11         double[] arregloDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
12         char[] arregloChar = { 'H', 'O', 'L', 'A' };
13
14         Console.WriteLine( "El arreglo arregloInt contiene:" );
15         ImprimirArreglo( arregloInt ); // pasa un arreglo int como argumento
16         Console.WriteLine( "El arreglo arregloDouble contiene:" );
17         ImprimirArreglo( arregloDouble ); // pasa un arreglo double como argumento
18         Console.WriteLine( "El arreglo arregloChar contiene:" );
19         ImprimirArreglo( arregloChar ); // pasa un arreglo char como argumento
20     } // fin de Main
21
22     // imprime en pantalla el arreglo int
23     static void ImprimirArreglo( int[] arregloEntrada )
24     {
25         foreach ( int elemento in arregloEntrada )
26             Console.Write( elemento + " " );
27
28         Console.WriteLine( "\n" );
29     } // fin del método ImprimirArreglo
30
31     // imprime en pantalla el arreglo double
32     static void ImprimirArreglo( double[] arregloEntrada )
33     {
34         foreach ( double elemento in arregloEntrada )
35             Console.Write( elemento + " " );
36
37         Console.WriteLine( "\n" );
38     } // fin del método ImprimirArreglo
39
40     // imprime en pantalla el arreglo char
41     static void ImprimirArreglo( char[] arregloEntrada )
42     {
43         foreach ( char elemento in arregloEntrada )
44             Console.Write( elemento + " " );
45
46         Console.WriteLine( "\n" );
47     } // fin del método ImprimirArreglo
48 } // fin de la clase MetodosSobrecargados

```

Figura 25.1 | Mostrar arreglos de distintos tipos mediante el uso de los métodos sobrecargados. (Parte I de 2).

```

El arreglo arregloInt contiene:
1 2 3 4 5 6

El arreglo arregloDouble contiene:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

El arreglo arregloChar contiene:
H O L A

```

Figura 25.1 | Mostrar arreglos de distintos tipos mediante el uso de los métodos sobrecargados. (Parte 2 de 2).

El programa empieza declarando e inicializando tres arreglos: un arreglo `int` de seis elementos llamado `arregloInt` (línea 10), un arreglo `double` de siete elementos llamado `arregloDouble` (línea 11) y un arreglo `char` de cuatro elementos llamado `arregloChar` (línea 12). Después, las líneas 14-19 imprimen los arreglos en pantalla.

Cuando el compilador encuentra la llamada a un método, trata de localizar la declaración de un método con el mismo nombre y los mismos parámetros que coincidan con los tipos de argumentos en la llamada al método. En este ejemplo, cada llamada a `ImprimirArreglo` coincide exactamente con una de las declaraciones del método `ImprimirArreglo`. Por ejemplo, la línea 15 llama a `ImprimirArreglo` con `arregloInt` como su argumento. En tiempo de compilación, el compilador determina el tipo del argumento `arregloInt` (es decir, `int[]`), trata de localizar un método llamado `ImprimirArreglo` que especifique un solo parámetro `int[]` (lo encuentra en las líneas 23-29) y establece una llamada a ese método. De manera similar, cuando el compilador encuentra la llamada a `ImprimirArreglo` en la línea 17, determina el tipo del argumento `arregloDouble` (es decir, `double[]`), después trata de localizar un método llamado `ImprimirArreglo` que especifique un solo parámetro `double[]` (lo encuentra en las líneas 32-38) y establece una llamada a ese método. Por último, cuando el compilador encuentra la llamada a `ImprimirArreglo` en la línea 19, determina el tipo del argumento `arregloChar` (es decir, `char[]`), después trata de localizar un método llamado `ImprimirArreglo` que especifique un solo parámetro `char[]` (lo encuentra en las líneas 41-47) y establece una llamada a ese método.

Estudie cada uno de los métodos `ImprimirArreglo`. Observe que el tipo de elementos del arreglo (`int`, `double` o `char`) aparece en dos ubicaciones en cada método: el encabezado del mismo (líneas 23, 32 y 41) y el encabezado de la instrucción `foreach` (líneas 25, 34 y 43). Si sustituyéramos los tipos de los elementos en cada método con un nombre genérico (elegimos `E` para representar el tipo “elemento”), entonces los tres métodos se verían como el de la figura 25.2. Parece ser que, si podemos sustituir el tipo del elemento arreglo en cada uno de los tres métodos con un solo “parámetro de tipo genérico”, entonces podemos declarar un método `ImprimirArreglo` que pueda mostrar los elementos de cualquier arreglo. El método en la figura 25.2 no compila, ya que su sintaxis no es correcta. En la figura 25.3 declaramos un método `ImprimirArreglo` genérico con la sintaxis apropiada.

25.3 Implementación de un método genérico

Si las operaciones realizadas por varios métodos sobrecargados son idénticas para cada tipo de argumento, los métodos sobrecargados pueden codificarse en forma más compacta y conveniente, usando un método genérico.

```

1  static void ImprimirArreglo( E[] arregloEntrada )
2  {
3      foreach ( E elemento in arregloEntrada )
4          Console.Write( elemento + " " );
5
6      Console.WriteLine( "\n" );
7  } // fin del método ImprimirArreglo

```

Figura 25.2 | Método `ImprimirArreglo` en el que los nombres de los tipos se sustituyen por convención con el nombre genérico `E`.

Usted puede escribir una sola declaración de un método genérico, que pueda llamarse en distintos momentos, con argumentos de tipos diferentes. Con base en los tipos de los argumentos que se pasan al método genérico, el compilador maneja cada llamada al método de manera apropiada.

La figura 25.3 reimplementa la aplicación de la figura 25.1, usando un método `ImprimirArreglo` genérico (líneas 24-30). Observe que las llamadas al método `ImprimirArreglo` en las líneas 16, 18 y 20 son idénticas a las de la figura 25.1, los resultados de las dos aplicaciones son idénticos y el código en la figura 25.3 tiene 17 líneas menos que el código en la figura 25.1. Como se ilustra en la figura 25.3, los genéricos nos permiten crear y evaluar nuestro código una vez, para después utilizar ese código con muchos tipos de datos distintos. Esto demuestra el poder expresivo de los genéricos.

La línea 24 empieza la declaración del método `ImprimirArreglo`. Todas las declaraciones de métodos genéricos tienen una *lista de parámetros de tipo* delimitada entre los signos `< y >` (`< E >` en este ejemplo) que va después del nombre del método. Cada lista de parámetros de tipo contiene uno o más *parámetros de tipo*, separados por comas. Un parámetro de tipo es un identificador que se utiliza en vez de los nombres reales de los tipos. En la declaración de un método genérico, los parámetros de tipo pueden usarse para declarar el tipo de valor de retorno,

```

1 // Fig. 25.3: MetodoGenerico.cs
2 // Uso de métodos sobrecargados para imprimir arreglos de distintos tipos.
3 using System;
4 using System.Collections.Generic;
5
6 class MetodoGenerico
7 {
8     static void Main( string[] args )
9     {
10         // crea arreglos de tipo int, double and char
11         int[] arregloInt = { 1, 2, 3, 4, 5, 6 };
12         double[] arregloDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
13         char[] arregloChar = { 'H', 'O', 'L', 'A' };
14
15         Console.WriteLine( "El arreglo arregloInt contiene:" );
16         ImprimirArreglo( arregloInt ); // pasa un arreglo int como argumento
17         Console.WriteLine( "El arreglo arregloDouble contiene:" );
18         ImprimirArreglo( arregloDouble ); // pasa un arreglo double como argumento
19         Console.WriteLine( "El arreglo arregloChar contiene:" );
20         ImprimirArreglo( arregloChar ); // pasa un arreglo char como argumento
21     } // fin de Main
22
23     // imprime un pantalla un arreglo de todos los tipos
24     static void ImprimirArreglo< E >( E[] arregloEntrada )
25     {
26         foreach ( E element in arregloEntrada )
27             Console.Write( element + " " );
28
29         Console.WriteLine( "\n" );
30     } // fin del método ImprimirArreglo
31 } // fin de la clase MetodoGenerico

```

```

El arreglo arregloInt contiene:
1 2 3 4 5 6

```

```

El arreglo arregloDouble contiene:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

```

```

El arreglo arregloChar contiene:
H O L A

```

Figura 25.3 | Impresión de los elementos de un arreglo, mediante el uso del método genérico `ImprimirArreglo`.

los tipos de los parámetros y los tipos de las variables locales; los parámetros de tipo actúan como receptáculos para los tipos de los argumentos que se pasan al método genérico. El cuerpo de un método genérico se declara de igual forma que con cualquier otro método. Debemos tener en cuenta que los nombres de los parámetros de tipo esparcidos en la declaración del método deben coincidir con los que se declaran en la lista de parámetros de tipo. Por ejemplo, la línea 26 declara a `elemento` en la instrucción `foreach` con el tipo `E`, que coincide con el parámetro de tipo (`E`) declarado en la línea 24. Además, un parámetro de tipo puede declararse sólo una vez en la lista de parámetros de tipo, pero puede aparecer más de una vez en la lista de parámetros del método. Los nombres de los parámetros de tipo no necesitan ser únicos entre varios métodos genéricos distintos.



Error común de programación 25.1

Si olvida incluir la lista de parámetros de tipo al declarar un método genérico, el compilador no reconocerá los nombres de los parámetros de tipo cuando los encuentre en un método. Esto produce errores de compilación.

La lista de parámetros de tipo de `ImprimirArreglo` (línea 24) declara el parámetro de tipo `E` como receptáculo para el tipo de elemento de arreglo que `ImprimirArreglo` mostrará en pantalla. Observe que `E` aparece en la lista de parámetros como el tipo de elemento de arreglo (línea 24). El encabezado de la instrucción `foreach` (línea 26) también utiliza a `E` como el tipo de `elemento`. Éstas son las mismas dos ubicaciones en donde los métodos `ImprimirArreglo` sobrecargados de la figura 25.1 especificaron `int`, `double` o `char` como el tipo de elemento del arreglo. El resto de `ImprimirArreglo` es idéntico a la versión que se presenta en la figura 25.1.



Buena práctica de programación 25.1

Se recomienda que los parámetros de tipo se especifiquen como letras mayúsculas individuales. Por lo general, un parámetro de tipo que representa el tipo de un elemento en un arreglo (o en otra colección) se llama `E`, por “elemento”, o `T` por “tipo”.

Al igual que en la figura 25.1, el programa de la figura 25.3 empieza por declarar e inicializar el arreglo `int` de seis elementos llamado `arregloInt` (línea 11), el arreglo `double` de siete elementos llamado `arregloDouble` (línea 12) y el arreglo `char` de cuatro elementos llamado `arregloChar` (línea 13). Después se imprime cada arreglo, llamando a `ImprimirArreglo` (líneas 16, 18 y 20); una vez con el argumento `arregloInt`, otra con el argumento `arregloDouble` y la última con el argumento `arregloChar`.

Cuando el compilador encuentra la llamada a un método como en la línea 16, analiza el conjunto de métodos (tanto genéricos como no genéricos) que podrían coincidir con la llamada al método, buscando un método que coincida con la llamada en forma exacta. Si no hay coincidencias exactas, el compilador elige el método que coincida mejor. Si no hay métodos que coincidan, o si hay más de una mejor coincidencia, el compilador genera un error. En la sección 14.5.5.1 de la Especificación del lenguaje C# de Ecma encontrará los detalles completos acerca de la resolución de llamadas a métodos:

www.ecma-international.org/publications/files/ECMA-ST/ECma-334.pdf

o también en la sección 20.9.7 de la Especificación 2.0 del lenguaje C# de Microsoft:

msdn.microsoft.com/vcsharp/programming/language

En el caso de la línea 16, el compilador determina que ocurre una coincidencia exacta si el parámetro de tipo `E` en las líneas 24 y 26 de la declaración del método `ImprimirArreglo` se sustituye con el tipo de los elementos en el argumento `arregloInt` de la llamada al método (es decir, `int`). Después, el compilador establece una llamada a `ImprimirArreglo` con `int` como el *argumento de tipo* para el parámetro de tipo `E`. A esto se le conoce como el proceso de *inferencia de tipos*. El mismo proceso se repite para las llamadas al método `ImprimirArreglo` en las líneas 18 y 20.



Error común de programación 25.2

Si el compilador no puede encontrar una sola declaración de un método no genérico o genérico que sea la mejor coincidencia para la llamada a un método, o si hay varias mejores coincidencias, se produce un error de compilación.

También podemos usar *argumentos de tipo explícito* para indicar el tipo exacto que debe utilizarse para llamar a una función genérica. Por ejemplo, la línea 16 podría escribirse así:

```
ImprimirArreglo< int >( arregloInt ); // pasa un arreglo int como argumento
```

La llamada al método anterior proporciona en forma explícita el argumento de tipo (`int`) que debe utilizarse para sustituir el parámetro de tipo `E` en las líneas 24 y 26 de la declaración del método `ImprimirArreglo`.

El compilador también determina si las operaciones realizadas en los parámetros de tipo del método pueden aplicarse a los elementos del tipo almacenado en el argumento tipo arreglo. La única operación que se realiza con los elementos tipo arreglo en este ejemplo es imprimir en pantalla la representación `string` de esos elementos. La línea 27 realiza una conversión implícita para cada elemento del arreglo de tipo de valor, y una llamada implícita a `ToString` en cada elemento del arreglo de tipo de referencia. Como todos los objetos tienen un método `ToString`, el compilador queda satisfecho de que la línea 27 realice una operación válida para cualquier elemento del arreglo.

Al declarar a `ImprimirArreglo` como método genérico en la figura 25.3, eliminamos la necesidad de los métodos sobrecargados de la figura 25.1, lo cual nos ahorra 17 líneas de código y se crea un método reutilizable que puede imprimir las representaciones de cadena de los elementos en cualquier arreglo, no sólo en arreglos de elementos `int`, `double` o `char`.

25.4 Restricciones de los tipos

En esta sección presentaremos un método genérico llamado `Maximo`, que determina y devuelve el mayor de sus tres argumentos (todos del mismo tipo). El método genérico en este ejemplo utiliza el parámetro de tipo para declarar tanto el tipo de valor de retorno del método, como sus parámetros. Por lo general, al comparar valores para determinar cuál es mayor, se utiliza el operador `>`. No obstante, este operador no está sobrecargado para utilizarse con cualquier tipo integrado en la FCL, o que pueda definirse mediante la extensión de esos tipos. El código genérico se restringe a realizar operaciones cuyo funcionamiento con todos los tipos posibles esté garantizado. Por ende, una expresión como `variable1<variable2` no está permitida, a menos que el compilador pueda asegurar que es posible utilizar el operador `<` para todos los tipos que vayan a emplearse en el código genérico. De manera similar, no se puede llamar a un método en una variable de tipo genérico, a menos que el compilador pueda asegurar que todos los tipos que se van a utilizar en el código genérico cuentan con soporte para ese método.

Interfaz `IComparable< E >`

Es posible comparar dos objetos del mismo tipo, si ese tipo implementa a la interfaz genérica `IComparable< T >` (del espacio de nombres `System`). Un beneficio de implementar esta interfaz es que se pueden utilizar los objetos `IComparable< T >` con los métodos de ordenamiento y búsqueda de las clases en el espacio de nombres `System.Collections.Generic`; en el capítulo 26, Colecciones, hablaremos sobre esos métodos. Todas las estructuras en la FCL que corresponden a los tipos simples implementan esta interfaz. Por ejemplo, la estructura para el tipo simple `double` es `Double`, y la estructura para el tipo simple `int` es `Int32`; tanto `Double` como `Int32` implementan a la interfaz `IComparable`. Los tipos que implementan a `IComparable< T >` deben declarar un método `CompareTo` para comparar objetos. Por ejemplo, si tenemos dos objetos `int` llamados `int1` e `int2`, pueden compararse con la siguiente expresión:

```
int1.CompareTo( int2 )
```

El método `CompareTo` debe devolver 0 si los objetos son iguales, un número negativo si `int1` es menor que `int2`, o un entero positivo si `int1` es mayor que `int2`. Es responsabilidad del programador que declara a un tipo que implementa a la interfaz `IComparable< T >` declarar el método `CompareTo`, de tal forma que compare el contenido de dos objetos de ese tipo y devuelva el resultado apropiado.

Especificación de las restricciones de los tipos

Aun cuando pueden compararse objetos `IComparable`, no pueden usarse con el código genérico de manera pre-determinada, ya que no todos los tipos implementan a la interfaz `IComparable< T >`. No obstante, podemos restringir los tipos que pueden usarse con un método o clase genéricos para asegurar que cumplan con ciertos requerimientos. Esta característica, conocida como *restricción de tipos*, restringe el tipo del argumento que se

proporciona a un parámetro de tipo específico. La figura 25.4 declara el método `Maximo` (líneas 20-33) con una restricción de tipo que requiere que cada uno de los argumentos del método sea de tipo `IComparable< T >`. Esta restricción es importante, ya que no todos los objetos pueden compararse. Sin embargo, se garantiza que todos los objetos `IComparable< T >` tienen un método `CompareTo` que puede utilizarse en el método `Maximo` para determinar el mayor de sus tres argumentos.

El método genérico `Maximo` utiliza el parámetro de tipo `T` como el tipo de valor de retorno del método (línea 20), como el tipo de los parámetros `x`, `y` y `z` del método (línea 20) y como el tipo de la variable local `max` (línea 22). La cláusula **where** del método `Maximo` (después de la lista de parámetros en la línea 20) especifica la restricción de tipos para el parámetro de tipo `T`. En este caso, la cláusula `where T : IComparable< T >` indica que este método requiere que los argumentos de tipo implementen a la interfaz `IComparable< T >`. Si no se especifica una restricción de tipo, la predeterminada es `object`.

C# cuenta con varios tipos de restricciones de tipos. Una *restricción de clase* indica que el argumento de tipo debe ser un objeto de una clase base específica, o de una de sus subclases. Una *restricción de interfaz* indica que la clase del argumento de tipo debe implementar una interfaz específica. La restricción de tipo en la línea 20 es una

```

1  // Fig. 25.4: PruebaMaximo.cs
2  // El método genérico Maximo devuelve el mayor de los tres objetos.
3  using System;
4
5  class PruebaMaximo
6  {
7      static void Main( string[] args )
8      {
9          Console.WriteLine( "El máximo de {0}, {1} y {2} es {3}\n",
10              3, 4, 5, Maximo( 3, 4, 5 ) );
11          Console.WriteLine( "El máximo de {0}, {1} y {2} es {3}\n",
12              6.6, 8.8, 7.7, Maximo( 6.6, 8.8, 7.7 ) );
13          Console.WriteLine( "El máximo de {0}, {1} y {2} es {3}\n",
14              "pera", "manzana", "naranja",
15              Maximo( "pera", "manzana", "naranja" ) );
16      } // fin de Main
17
18      // la función genérica determina el
19      // mayor de los tres objetos IComparable
20      static T Maximo< T >( T x, T y, T z ) where T : IComparable < T >
21      {
22          T max = x; // supone que al principio x es el mayor
23
24          // compara y con max
25          if ( y.CompareTo( max ) > 0 )
26              max = y; // y es el mayor hasta ahora
27
28          // compara z con max
29          if ( z.CompareTo( max ) > 0 )
30              max = z; // z es el mayor
31
32          return max; // devuelve el objeto mayor
33      } // fin del método Maximo
34  } // fin de la clase PruebaMaximo

```

El máximo de 3, 4 y 5 es 5

El máximo de 6.6, 8.8 y 7.7 es 8.8

El máximo de pera, manzana y naranja es pera

Figura 25.4 | El método genérico `Maximo` con una restricción de tipos en su parámetro de tipo.

restricción de interfaz, ya que `IComparable< T >` es una interfaz. Podemos especificar que el argumento de tipo debe ser un tipo de referencia o un tipo de valor, mediante el uso de la *restricción de tipo de referencia* (**class**) o la *restricción de tipo de valor* (**struct**), respectivamente. Por último, podemos especificar una *restricción de constructor* (**new()**) para indicar que el código genérico puede usar el operador `new` para crear nuevos objetos del tipo representado por el parámetro de tipo. Si se especifica un parámetro de tipo con una restricción de constructor, la clase del argumento de tipo debe proporcionar a `public` un constructor sin parámetros o predeterminado, para asegurar que puedan crearse objetos de la clase sin pasar argumentos al constructor; en caso contrario, se produce un error de compilación.

Es posible aplicar *restricciones múltiples* a un parámetro de tipo. Para ello, simplemente se proporciona una lista de restricciones separadas por comas en la cláusula `where`. Si tiene una restricción de clase, de tipo de referencia o de tipo de valor, debe listarse primero; sólo uno de estos tipos de restricciones pueden usarse para cada parámetro de tipo. Las restricciones de interfaz (en caso de haber) se listan después. La restricción de constructor se lista al último (si la hay).

Análisis del código

El método `Maximo` supone que su primer argumento (`x`) es el mayor y lo asigna a la variable local `max` (línea 22). A continuación, la instrucción `if` en las líneas 25-26 determinan si `y` es mayor que `max`. La condición invoca al método `CompareTo` de `y` con la expresión `y.CompareTo( max )`. Si `y` es mayor que `max`, entonces `y` se asigna a la variable `max` (línea 26). De manera similar, la instrucción en las líneas 29-30 determina si `z` es mayor que `max`. Si esto es cierto, la línea 30 asigna `z` a `max`. Después, la línea 32 devuelve `max` al método que hizo la llamada.

En `Main` (líneas 7-16), la línea 10 llama a `Maximo` con los enteros 3, 4 y 5. El método genérico `Maximo` es una coincidencia para esta llamada, pero sus argumentos deben implementar la interfaz `IComparable< T >`, para asegurar que puedan compararse. El tipo `int` es un sinónimo para `struct Int32`, que implementa a la interfaz `IComparable< int >`. Por ende, los valores `int` (y otros tipos simples) son argumentos válidos para el método `Maximo`.

La línea 12 pasa tres argumentos `double` a `Maximo`. De nuevo, esto se permite debido a que `double` es sinónimo del tipo `struct Double`, que implementa a `IComparable< double >`. La línea 15 pasa a `Maximo` tres objetos `string`, que son también objetos `IComparable< string >`. Observe que colocamos de manera intencional el valor más grande en una posición distinta en cada llamada al método (líneas 10, 12 y 15) para mostrar que el método genérico siempre encuentra el valor máximo, sin importar su posición en la lista de argumentos y sin importar el argumento de tipo inferido.

25.5 Sobrecarga de métodos genéricos

Un método genérico puede *sobrecargarse*. Una clase puede proporcionar dos o más métodos genéricos con el mismo nombre, pero con distintos parámetros para los métodos. Por ejemplo, podríamos proporcionar una segunda versión del método genérico `ImprimirArreglo` (figura 25.3) con los parámetros adicionales `indiceBajo` e `indiceAlto`, para especificar la porción del arreglo que se desea imprimir. Un método genérico también puede sobrecargarse mediante otro método genérico con el mismo nombre y con un número distinto de parámetros de tipo, o mediante un método genérico con distintos números de parámetros de tipo y de parámetros del método.

Un método genérico puede sobrecargarse mediante métodos no genéricos que tengan el mismo nombre del método y el mismo número de parámetros. Cuando el compilador encuentra una llamada a un método, busca la declaración del método que coincida con más precisión con el nombre del método y los tipos de los argumentos especificados en la llamada. Por ejemplo, el método genérico `ImprimirArreglo` de la figura 25.3 podría sobrecargarse con una versión específica para objetos `string`, que imprima en pantalla estos objetos en formato tabular ordenado. Si el compilador no puede hacer que coincida la llamada a un método con un método no genérico o genérico, o si hay ambigüedad debido a varias posibles coincidencias, el compilador genera un error. Los métodos genéricos también pueden sobrecargarse mediante métodos no genéricos que tengan el mismo nombre del método, pero un número distinto de parámetros.

25.6 Clases genéricas

El concepto de una estructura de datos (por ejemplo, una pila), que contiene elementos de datos, puede comprenderse sin importar el tipo de elemento que manipule. Una clase genérica proporciona el medio para describir

una clase en forma independiente de su tipo. Así, podemos crear instancias de objetos específicos para el tipo de la clase genérica. Esta capacidad es una oportunidad para la reutilización de software.

Una vez que se tiene una clase genérica, se puede usar una notación simple y concisa para indicar el (los) tipo(s) actual(es) que debe(n) usarse en vez del (los) parámetro(s) de tipo de la clase. En tiempo de compilación, el compilador se encarga de la seguridad de los tipos de nuestro código, y el sistema en tiempo de ejecución sustituye los parámetros de tipo con argumentos reales, para que nuestro código cliente interactúe con la clase genérica.

Por ejemplo, una clase genérica Pila podría ser la base para crear muchas clases de Pila (por ejemplo, “Pila de double”, “Pila de int”, “Pila de char”, “Pila de Empleado”). La figura 25.5 presenta la declaración de una clase Pila genérica. La declaración de una clase genérica es similar a la de una clase no genérica, con la excepción de que el nombre de la clase va seguido de una lista de parámetros de tipo (línea 5) y, en este caso, una restricción en su parámetro de tipo (que veremos en breve). El parámetro de tipo E representa el tipo de elemento que la Pila manipulará. Al igual que con los métodos genéricos, la lista de parámetros de tipo de una clase genérica puede tener uno o más parámetros de tipo, separados por comas. El parámetro de tipo E se utiliza a lo largo de la declaración de la clase Pila (figura 25.5) para representar el tipo de elemento. La clase Pila declara la variable `elementos` como un arreglo de tipo E (línea 8). Este arreglo (creado en la línea 20 o 22) almacena los elementos de la Pila. [Nota: Este ejemplo implementa una Pila como un arreglo. Como vimos en el capítulo 24, Estructuras de datos, es común que los objetos Pila se implementen también como listas enlazadas.]

```

1  // Fig. 25.5: Pila.cs
2  // La clase genérica Pila
3  using System;
4
5  class Pila< E >
6  {
7      private int superior; // ubicación del elemento superior
8      private E[] elementos; // arreglo que almacena elementos de la Pila
9
10     // el constructor sin parámetros crea una Pila del tamaño predeterminado
11     public Pila() : this( 10 ) // tamaño predeterminado de la pila
12     {
13         // constructor vacío; llama al constructor de la línea 18 para realizar
14         // la inicialización
15     } // fin el constructor de Pila
16
17     // el constructor crea una Pila con el número especificado de elementos
18     public Pila( int tamanoPila )
19     {
20         if ( tamanoPila > 0 ) // valida tamanoPila
21             elementos = new E[ tamanoPila ]; // crea elementos tamanoPila
22         else
23             elementos = new E[ 10 ]; // crea 10 elementos
24
25         superior = -1; // al principio, la Pila está vacía
26     } // fin del constructor de Pila
27
28     // mete (push) un elemento en la Pila; si tiene éxito, devuelve true
29     // en caso contrario, lanza ExcepcionPilaLlena
30     public void Push( E meterValor )
31     {
32         if ( superior == elementos.Length - 1 ) // la Pila está llena
33             throw new ExcepcionPilaLlena( String.Format(
34                 "La pila está llena, no se puede meter {0}", meterValor ) );
35
36         superior++; // incrementa elemento superior
37         elementos[ superior ] = meterValor; // coloca meterValor en la Pila

```

Figura 25.5 | Declaración de la clase genérica Pila. (Parte I de 2).

```

37     } // fin del método Push
38
39     // devuelve el elementos superior si no está vacía
40     // en cualquier otro caso lanza ExcepcionPilaVacía
41     public E Pop()
42     {
43         if ( superior == -1 ) // la pila está vacía
44             throw new ExcepcionPilaVacía( "La pila está vacía, no se puede sacar" );
45
46         superior--; // decrementa elemento superior
47         return elementos[ superior + 1 ]; // devuelve valor del elemento superior
48     } // fin del método Pop
49 } // fin de la clase Pila

```

Figura 25.5 | Declaración de la clase genérica Pila. (Parte 2 de 2).

La clase Pila tiene dos constructores. El constructor sin parámetros (líneas 11-14) pasa el tamaño de pila predeterminado (10) al constructor de un argumento, usando la sintaxis `this` (línea 11) para invocar a otro constructor en la misma clase. El constructor de un argumento (líneas 17-25) valida el argumento `tamanoPila` y crea un arreglo con el `tamanoPila` especificado (si es mayor que 0) o un arreglo de 10 elementos, en caso contrario.

El método Push (líneas 29-37) determina primero si se está tratando de meter un elemento en una Pila llena. Si es así, las líneas 32-33 lanzan una `ExcepcionPilaLlena` (declarada en la figura 25.6). Si la Pila no está llena, la línea 35 incrementa el contador `superior` para indicar la nueva posición superior, y la línea 36 coloca el argumento en esa ubicación del arreglo `elementos`.

El método Pop (líneas 41-48) determina primero si se está tratando de sacar un elemento de una Pila vacía. Si es así, la línea 44 lanza una `ExcepcionPilaVacía` (declarada en la figura 25.7). En caso contrario, la línea 46 decrementa el contador `superior` para indicar la nueva posición superior, y la línea 47 devuelve el elemento superior original de la Pila.

Las clases `ExcepcionPilaLlena` (figura 25.6) y `ExcepcionPilaVacía` (figura 25.7) proporcionan cada una un constructor sin parámetros y un constructor con un argumento de las clases de excepciones (como vimos en la sección 12.8). El constructor sin parámetros establece el mensaje de error predeterminado, y el constructor con un argumento establece un mensaje de error personalizado.

Al igual que con los métodos genéricos, cuando se compila una clase genérica el compilador realiza la comprobación de tipos en los parámetros de tipo de la clase, para asegurar que puedan utilizarse con el código en la

```

1  // Fig. 25.6: ExcepcionPilaLlena.cs
2  // Indica que una pila está llena.
3  using System;
4
5  class ExcepcionPilaLlena : ApplicationException
6  {
7      // constructor sin parámetros
8      public ExcepcionPilaLlena() : base( "La pila está llena" )
9      {
10         // constructor vacío
11     } // fin del constructor de ExcepcionPilaLlena
12
13     // constructor con un parámetro
14     public ExcepcionPilaLlena( string exception ) : base( exception )
15     {
16         // constructor vacío
17     } // fin del constructor de ExcepcionPilaLlena
18 } // fin de la clase ExcepcionPilaLlena

```

Figura 25.6 | Declaración de la clase `ExcepcionPilaLlena`.

```

1 // Fig. 25.7: ExcepcionPilaVacía.cs
2 // Indica que una pila está vacía.
3 using System;
4
5 class ExcepcionPilaVacía : ApplicationException
6 {
7     // constructor sin parámetros
8     public ExcepcionPilaVacía() : base( "La pila está vacía" )
9     {
10         // constructor vacío
11     } // fin del constructor de ExcepcionPilaVacía
12
13     // constructor con un parámetro
14     public ExcepcionPilaVacía( string exception ) : base( exception )
15     {
16         // constructor vacío
17     } // fin del constructor de ExcepcionPilaVacía
18 } // fin de la clase ExcepcionPilaVacía

```

Figura 25.7 | Declaración de la clase ExcepcionPilaVacía.

clase genérica. Las restricciones determinan las operaciones que pueden realizarse con los parámetros de tipo. El sistema en tiempo de ejecución sustituye los parámetros de tipo con los verdaderos tipos en tiempo de ejecución. Para la clase Pila (figura 25.5) no se especifica una restricción de tipos, por lo que se utiliza la restricción de tipo predeterminada, `object`. El alcance del parámetro de tipo de una clase genérica es toda la clase.

Ahora consideraremos una aplicación (figura 25.8) que utiliza la clase genérica Pila. Las líneas 13-14 declaran variables de tipo `Pila< double >` (se pronuncia como “Pila de elementos double”) y `Pila< int >` (se pronuncia como “Pila de elementos int”). Los tipos `double` e `int` son los argumentos de tipo de la Pila. El compilador sustituye los parámetros de tipo en la clase genérica, para que el compilador pueda realizar la comprobación de tipos. El método `Main` crea instancias de objetos `pilaDouble` con un tamaño de 5 (línea 18) y `pilaInt` con un tamaño de 10 (línea 19), después llama a los métodos `PruebaPushDouble` (líneas 28-48), `PruebaPopDouble` (líneas 51-73), `PruebaPushInt` (líneas 76-96) y `PruebaPopInt` (líneas 99-121) para manipular los dos objetos Pila en este ejemplo.

```

1 // Fig. 25.8: PruebaPila.cs
2 // Programa de prueba de la clase genérica Pila.
3 using System;
4
5 class PruebaPila
6 {
7     // crea arreglos con elementos double e int
8     static double[] elementosDouble =
9         new double[]{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
10    static int[] elementosInt =
11        new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
12
13    static Pila< double > pilaDouble; // pila que almacena objetos double
14    static Pila< int > pilaInt; // pila que almacena objetos int
15
16    static void Main( string[] args )
17    {
18        pilaDouble = new Pila< double >( 5 ); // Pila de valores double
19        pilaInt = new Pila< int >( 10 ); // Pila de valores int
20

```

Figura 25.8 | Programa de prueba de la clase genérica Pila. (Parte I de 4).

```

21     PruebaPushDouble(); // mete valores double en pilaDouble
22     PruebaPopDouble(); // saca valores double de pilaDouble
23     PruebaPushInt(); // mete valores int en pilaInt
24     PruebaPopInt(); // saca valores int de pilaInt
25 } // fin de Main
26
27 // prueba el método Push con pilaDouble
28 static void PruebaPushDouble()
29 {
30     // mete elementos en la pila
31     try
32     {
33         Console.WriteLine( "\nMetiendo elementos en pilaDouble" );
34
35         // mete elementos en la pila
36         foreach ( double elemento in elementosDouble )
37         {
38             Console.Write( "{0:F1} ", elemento );
39             pilaDouble.Push( elemento ); // mete elemento en pilaDouble
40         } // fin de foreach
41     } // fin de try
42     catch ( ExcepcionPilaLlena exception )
43     {
44         Console.Error.WriteLine();
45         Console.Error.WriteLine( "Mensaje: " + exception.Message );
46         Console.Error.WriteLine( exception.StackTrace );
47     } // fin de catch
48 } // fin del método PruebaPushDouble
49
50 // prueba el método Pop con pilaDouble
51 static void PruebaPopDouble()
52 {
53     // saca elementos de la pila
54     try
55     {
56         Console.WriteLine( "\nSacando elementos de pilaDouble" );
57
58         double sacarValor; // almacena elemento eliminado de la pila
59
60         // elimina todos los elementos de la Pila
61         while ( true )
62         {
63             sacarValor = pilaDouble.Pop(); // saca elemento de pilaDouble
64             Console.Write( "{0:F1} ", sacarValor );
65         } // fin de while
66     } // fin de try
67     catch ( ExcepcionPilaVacía exception )
68     {
69         Console.Error.WriteLine();
70         Console.Error.WriteLine( "Mensaje: " + exception.Message );
71         Console.Error.WriteLine( exception.StackTrace );
72     } // fin de catch
73 } // fin del método PruebaPopDouble
74
75 // prueba el método Push con pilaInt
76 static void PruebaPushInt()
77 {
78     // mete elementos en la pila
79     try

```

Figura 25.8 | Programa de prueba de la clase genérica Pila. (Parte 2 de 4).

```

80     {
81         Console.WriteLine( "\nMetiendo elementos en pilaInt" );
82
83         // mete elementos en la pila
84         foreach ( int elemento in elementosInt )
85         {
86             Console.Write( "{0} ", elemento );
87             pilaInt.Push( elemento ); // mete elemento en pilaInt
88         } // fin de foreach
89     } // fin de try
90     catch ( ExcepcionPilaLlena exception )
91     {
92         Console.Error.WriteLine();
93         Console.Error.WriteLine( "Mensaje: " + exception.Message );
94         Console.Error.WriteLine( exception.StackTrace );
95     } // fin de catch
96 } // fin del método PruebaPushInt
97
98 // prueba el método Pop con pilaInt
99 static void PruebaPopInt()
100 {
101     // saca elementos de la pila
102     try
103     {
104         Console.WriteLine( "\nSacando elementos depilaIntk" );
105
106         int sacarValor; // almacena elemento eliminado de la pila
107
108         // elimina todos los elementos de la Pila
109         while ( true )
110         {
111             sacarValor = pilaInt.Pop(); // saca elemento de pilaInt
112             Console.Write( "{0} ", sacarValor );
113         } // fin de while
114     } // fin de try
115     catch ( ExcepcionPilaVacía exception )
116     {
117         Console.Error.WriteLine();
118         Console.Error.WriteLine( "Mensaje: " + exception.Message );
119         Console.Error.WriteLine( exception.StackTrace );
120     } // fin de catch
121 } // fin del método PruebaPopInt
122 } // fin de la clase PruebaPila

```

Metiendo elementos en pilaDouble

1.1 2.2 3.3 4.4 5.5 6.6

Mensaje: La pila está llena, no se puede meter 6.6

en Pila`1.Push(E meterValor) en

C:\MisProyectos\Fig25_05_08\Pila\Pila.cs:línea 32

en PruebaPila.PruebaPushDouble() en

C:\MisProyectos\Fig25_05_08\Pila\PruebaPila.cs:línea 39

Sacando elementos de pilaDouble

5.5 4.4 3.3 2.2 1.1

Mensaje: La pila está vacía, no se puede sacar

en Pila`1.Pop() en C:\MisProyectos\Fig25_05_08\Pila\Pila.cs:línea 44

en PruebaPila.PruebaPopDouble() en

C:\MisProyectos\Fig25_05_08\Pila\PruebaPila.cs:línea 63

Figura 25.8 | Programa de prueba de la clase genérica Pila. (Parte 3 de 4).


```

Metiendo elementos en pilaInt
1 2 3 4 5 6 7 8 9 10 11
Mensaje: La pila está llena, no se puede meter 11
    en Pila<T>.Push(E meterValor) en
        C:\MisProyectos\Fig25_05_08\Pila\Pila.cs:línea 32
    en PruebaPila.PruebaPushInt() en
        C:\MisProyectos\Fig25_05_08\Pila\PruebaPila.cs:línea 87

Sacando elementos de pilaInt
10 9 8 7 6 5 4 3 2 1
Mensaje: La pila está vacía, no se puede sacar
    en Pila<T>.Pop() en C:\MisProyectos\Fig25_05_08\Pila\Pila.cs:línea 44
    en PruebaPila.PruebaPopInt() en
        C:\MisProyectos\Fig25_05_08\Pila\PruebaPila.cs:línea 111

```

Figura 25.8 | Programa de prueba de la clase genérica Pila. (Parte 4 de 4).

El método `PruebaPushDouble` (líneas 28-48) invoca al método `Push` para colocar en `pilaDouble` los valores `double` 1.1, 2.2, 3.3, 4.4 y 5.5 almacenados en el arreglo `elementosDouble`. La instrucción `for` termina cuando el programa de prueba trata de meter un sexto valor en `pilaDouble` (la cual está llena, ya que `pilaDouble` sólo puede almacenar cinco elementos). En este caso, el método lanza una `ExcepcionPilaLlena` (figura 25.6) para indicar que la Pila está llena. Las líneas 42-47 atrapan esta excepción e imprimen el mensaje y la información de rastreo de pila. Este rastreo de pila indica la excepción que ocurrió y muestra que el método `Push` de `Pila` generó la excepción en las líneas 34-35 del archivo `Pila.cs` (figura 25.5). El rastreo también muestra que el método `PruebaPushDouble` de `PruebaPila` hizo una llamada al método `Push` en la línea 39 de `PruebaPila.cs`. Esta información nos permite determinar los métodos que estaban en la pila de llamadas a métodos en el momento en que ocurrió la excepción. Debido a que este programa atrapa la excepción, el entorno en tiempo de ejecución de C# considera que se ha manejado la excepción y el programa puede continuar su ejecución.

El método `PruebaPopDouble` (líneas 51-73) invoca al método `Pop` de `Pila` en un ciclo `while` infinito para eliminar todos los valores de la pila. Observe en los resultados que los valores se sacan en el orden “último en entrar, primero en salir”; es evidente que ésta es la característica que define a las pilas. El ciclo `while` (líneas 61-65) continúa hasta que la pila está vacía. Al tratar de sacar un elemento de la pila vacía, se produce una `ExcepcionPilaVacía`. Esto hace que el programa proceda al bloque `catch` (líneas 67-72) y maneje la excepción, para que pueda continuar su ejecución. Cuando el programa de prueba trata de sacar un sexto valor, la `pilaDouble` está vacía, por lo que el método `Pop` lanza una `ExcepcionPilaVacía`.

El método `PruebaPushInt` (líneas 76-96) invoca al método `Push` de `Pila` para colocar valores en `pilaInt` hasta que se llene. El método `PruebaPopInt` (líneas 99-121) invoca al método `Pop` de `Pila` para eliminar valores de `pilaInt` hasta que se vacíe. Una vez más, observe que los valores se sacan en el orden “último en entrar, primero en salir”.

Creación de métodos genéricos para probar la clase Pila< E >

Observe que el código en los métodos `PruebaPushDouble` y `PruebaPushInt` es casi idéntico para meter valores en una `Pila< double >` o en una `Pila< int >`, respectivamente. De manera similar, el código en los métodos `PruebaPopDouble` y `PruebaPopInt` es casi idéntico para sacar valores de una `Pila< double >` o de una `Pila< int >`, respectivamente. Esto presenta otra oportunidad para usar los métodos genéricos. La figura 25.9 declara el método genérico `PruebaPush` (líneas 32-53) para realizar las mismas tareas que `PruebaPushDouble` y `PruebaPushInt` en la figura 25.8; esto es, meter los valores en una `Pila< E >`. De manera similar, el método genérico `PruebaPop` (líneas 55-77) realiza las mismas tareas que `PruebaPopDouble` y `PruebaPopInt` en la figura 25.8; esto es, sacar valores de una `Pila< E >`. Observe que los resultados de la figura 25.9 coinciden en forma precisa con los resultados de la figura 25.8.

El método `Main` (líneas 17-30) crea los objetos `Pila< double >` (línea 19) y `Pila< int >` (línea 20). Las líneas 23-29 invocan a los métodos genéricos `PruebaPush` y `PruebaPop` para probar los objetos `Pila`.

El método genérico `PruebaPush` (líneas 32-53) utiliza el parámetro de tipo `E` (que se especifica en la línea 32) para representar el tipo de datos almacenado en la Pila. El método genérico recibe tres argumentos: un `string`


```

1  // Fig. 25.9: PruebaPila.cs
2  // Programa de prueba de la clase genérica Pila.
3  using System;
4  using System.Collections.Generic;
5
6  class PruebaPila
7  {
8      // crea arreglos de elementos double e int
9      static double[] elementosDouble =
10         new double[] { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
11     static int[] elementosInt =
12         new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
13
14     static Pila< double > pilaDouble; // pila que almacena objetos double
15     static Pila< int > pilaInt; // pila que almacena objetos int
16
17     static void Main( string[] args )
18     {
19         pilaDouble = new Pila< double >( 5 ); // Pila de elementos double
20         pilaInt = new Pila< int >( 10 ); // Pila de elementos int
21
22         // mete elementos double en pilaDouble
23         PruebaPush( "pilaDouble", pilaDouble, elementosDouble );
24         // saca elementos de pilaDouble
25         PruebaPop( "pilaDouble", pilaDouble );
26         // mete elementos int en pilaInt
27         PruebaPush( "pilaInt", pilaInt, elementosInt );
28         // saca elementos int de pilaInt
29         PruebaPop( "pilaInt", pilaInt );
30     } // fin de Main
31
32     static void PruebaPush< E >( string nombre, Pila< E > pila,
33         IEnumerable< E > elementos )
34     {
35         // mete elementos en la pila
36         try
37         {
38             Console.WriteLine( "\nMetiendo elementos en " + nombre );
39
40             // mete elementos en la pila
41             foreach ( E elemento in elementos )
42             {
43                 Console.Write( "{0} ", elemento );
44                 pila.Push( elemento ); // mete elemento en la pila
45             } // fin de foreach
46         } // fin de try
47         catch ( ExcepcionPilaLlena excepcion )
48         {
49             Console.Error.WriteLine();
50             Console.Error.WriteLine( "Mensaje: " + excepcion.Message );
51             Console.Error.WriteLine( excepcion.StackTrace );
52         } // fin de catch
53     } // fin del método PruebaPush
54
55     static void PruebaPop< E >( string nombre, Pila< E > pila )
56     {
57         // saca elementos de la pila
58         try
59         {

```

Figura 25.9 | Paso de una Pila de tipo genérico a un método genérico. (Parte I de 2).

```

60         Console.WriteLine( "\nSacando elementos de " + nombre );
61
62         E sacarValor; // almacena elemento eliminado de la pila
63
64         // elimina todos los elementos de la Pila
65         while ( true )
66         {
67             sacarValor = pila.Pop(); // saca elemento de la pila
68             Console.Write( "{0} ", sacarValor );
69         } // fin de while
70     } // fin de try
71     catch ( ExcepcionPilaVacía excepcion )
72     {
73         Console.Error.WriteLine();
74         Console.Error.WriteLine( "Mensaje: " + excepcion.Message );
75         Console.Error.WriteLine( excepcion.StackTrace );
76     } // fin de catch
77 } // fin de PruebaPop
78 } // fin de la clase PruebaPila

```

Metiando elementos en pilaDouble

1.1 2.2 3.3 4.4 5.5 6.6

Mensaje: La pila está llena, no se puede meter 6.6

en Pila`1.Push(E meterValor) en
C:\MisProyectos\Fig25_09\Pila\Pila.cs:línea 32
en PruebaPila.PruebaPush[E](String nombre, Pila`1 pila, IEnumerable`1
elementos) en C:\MisProyectos\Fig25_09\Pila\PruebaPila.cs:línea 44

Sacando elementos de pilaDouble

5.5 4.4 3.3 2.2 1.1

Mensaje: La pila está vacía, no se pueden sacar elementos

en Pila`1.Pop() en C:\MisProyectos\Fig25_09\Pila\Pila.cs:línea 44
en PruebaPila.PruebaPop[E](String nombre, Pila`1 pila) en
C:\MisProyectos\Fig25_09\Pila\PruebaPila.cs:línea 67

Metiando elementos en pilaInt

1 2 3 4 5 6 7 8 9 10 11

Mensaje: La pila está llena, no se puede meter 11

en Pila`1.Push(E meterValor) en
C:\MisProyectos\Fig25_09\Pila\Pila.cs:línea 32
en PruebaPila.PruebaPush[E](String nombre, Pila`1 pila, IEnumerable`1
elementos) en C:\MisProyectos\Fig25_09\Pila\PruebaPila.cs:línea 44

Sacando elementos de pilaInt

10 9 8 7 6 5 4 3 2 1

Mensaje: La pila está vacía, no se pueden sacar elementos

en Pila`1.Pop() en C:\MisProyectos\Fig25_09\Pila\Pila.cs:línea 44
en PruebaPila.PruebaPop[E](String nombre, Pila`1 pila) en
C:\MisProyectos\Fig25_09\Pila\PruebaPila.cs:línea 67

Figura 25.9 | Paso de una Pila de tipo genérico a un método genérico. (Parte 2 de 2).

que representa el nombre del objeto Pila para fines de imprimirlo en pantalla, un objeto de tipo Pila< E > y un objeto IEnumerable< E >; el tipo de elementos que se meterán en Pila< E >. Observe que el compilador hace cumplir la consistencia entre el tipo de la Pila y los elementos que se meterán a la misma cuando se invoque a Push, que es el argumento de tipo de la llamada al método genérico. El método genérico PruebaPop (líneas 55-77) recibe dos argumentos: un string que representa el nombre del objeto Pila para fines de imprimirlo en pantalla, y un objeto de tipo Pila< E >. Observe que el parámetro de tipo E que se utiliza en los métodos

`PruebaPush` y `PruebaPop` tiene una restricción `new`, ya que ambos métodos reciben un objeto de tipo `Pila< E >` y el parámetro de tipo en la declaración genérica de la clase `Pila` tiene una restricción `new` (línea 5, figura 25.5). Esto asegura que los objetos que manipularán los métodos `PruebaPush` y `PruebaPop` puedan usarse con la clase genérica `Pila`, la cual requiere que los objetos almacenados en la `Pila` tengan un constructor `public` predeterminado o sin parámetros.

25.7 Observaciones acerca de los genéricos y la herencia

Los genéricos pueden utilizarse con la herencia de varias formas:

- Una clase genérica puede derivarse de una clase no genérica. Por ejemplo, la clase `object` (que no es una clase genérica) es una clase base directa o indirecta de todas las clases genéricas.
- Una clase genérica puede derivarse de otra clase genérica. En el capítulo 24 vimos que la clase `Pila` no genérica (figura 24.12) hereda de la clase `Lista` no genérica (figura 24.5). También podemos crear una clase `Pila` genérica si heredamos de una clase `Lista` genérica.
- Una clase no genérica puede derivarse de una clase genérica. Por ejemplo, podemos implementar una clase `ListaDirecciones` no genérica que herede de una clase `Lista` genérica, y que sólo almacene objetos `Direccion`.

25.8 Conclusión

En este capítulo se presentaron los genéricos: una de las herramientas más recientes de C#. Hablamos acerca de cómo los genéricos imponen la seguridad de tipos en tiempo de comprobación, mediante la comprobación de la discordancia de tipos en tiempo de comprobación. Aprendió que el compilador permite que el código genérico se compile sólo si todas las operaciones realizadas con los parámetros de tipo en el código genérico están soportadas por todos los tipos que podrían utilizarse con ese código genérico. También aprendió a declarar métodos y clases genéricas, mediante el uso de parámetros de tipo. Demostramos cómo utilizar una restricción de tipo para especificar los requerimientos de un parámetro de tipo; un componente clave de la seguridad de tipos en tiempo de ejecución. Hablamos sobre varios tipos de restricciones de tipos, incluyendo las restricciones de tipos de referencia, restricciones de tipos de valor, restricciones de clase, restricciones de interfaz y restricciones de constructor. Aprendió que una restricción de constructor indica que el argumento de tipo debe proporcionar un constructor `public` sin parámetros o predeterminado, de manera que los objetos de ese tipo puedan crearse con `new`. También hablamos sobre cómo implementar restricciones múltiples de tipos para un parámetro de tipo. Le mostramos cómo los genéricos mejoran la reutilización de código. Por último, mencionamos varias formas de usar los genéricos en la herencia. En el siguiente capítulo demostraremos las clases, interfaces y algoritmos de colecciones de la FCL de .NET. Las clases de colecciones son estructuras de datos preconstruidas que usted puede reutilizar en sus aplicaciones, lo cual le ahorra tiempo. En el siguiente capítulo presentaremos las colecciones genéricas y las colecciones no genéricas, que son más antiguas.

26

Colecciones

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Acerca de las colecciones genéricas y no genéricas que proporciona el .NET Framework.
- A utilizar los métodos `static` de la clase `Array` para manipular arreglos.
- A utilizar enumeradores para “recorrer” una colección.
- A utilizar la instrucción `foreach` con las colecciones de .NET.
- A utilizar las clases de colección no genéricas `ArrayList`, `Stack` y `Hashtable`.
- A utilizar las clases de colección genéricas `SortedDictionary` y `LinkedList`.
- A utilizar envolturas de sincronización para que las colecciones sean seguras en aplicaciones con subprocesamiento múltiple.

*¡Las figuras que puede
contener un contenedor brillante!*

—Theodore Roethke

*La sabiduría se adquiere
no por la edad,
sino por la capacidad.*

—Titus Maccius Plautus

*Es un acertijo envuelto en
un misterio dentro de un enigma.*

—Winston Churchill

*Creo que es la colección más
extraordinaria de talento,
de conocimiento humano,
que haya sido recopilada
en la Casa Blanca; con la
posible excepción de cuando
Thomas Jefferson cenó solo.*

—John F. Kennedy

- 26.1 Introducción
- 26.2 Generalidades acerca de las colecciones
- 26.3 La clase Array y los enumeradores
- 26.4 Colecciones no genéricas
 - 26.4.1 La clase ArrayList
 - 26.4.2 La clase Stack
 - 26.4.3 La clase Hashtable
- 26.5 Colecciones genéricas
 - 26.5.1 La clase genérica SortedDictionary
 - 26.5.2 La clase genérica LinkedList
- 26.6 Colecciones sincronizadas
- 26.7 Conclusión

26.1 Introducción

En el capítulo 24 vimos cómo crear y manipular estructuras de datos. La discusión fue de “bajo nivel”, en cuanto a que creamos de la manera difícil cada elemento de cada estructura de datos en forma dinámica mediante `new`, y modificamos las estructuras de datos manipulando directamente sus elementos y referencias a otros elementos. En este capítulo consideraremos las clases de estructuras de datos pre-empaquetadas que proporciona el .NET Framework. Estas clases se conocen como *clases de colección*, ya que almacenan colecciones de datos. Cada instancia de una de estas clases es una *colección* de elementos. Algunos ejemplos de colecciones son las cartas que utilizamos en un juego de cartas, las canciones almacenadas en su computadora, los registros de bienes raíces en su registro local de escrituras (que asignan números de libro y de página con los propietarios de los terrenos y casas), y los jugadores en su equipo deportivo favorito.

Las clases de colección permiten a los programadores almacenar conjuntos de elementos mediante el uso de estructuras de datos existentes, sin preocuparse por la forma en que se implementan. Éste es un excelente ejemplo de reutilización de código. Los programadores pueden escribir código con más rapidez y esperar un excelente rendimiento, maximizando la velocidad de ejecución y minimizando el consumo de memoria. En este capítulo hablaremos sobre las interfaces de colección que listan las capacidades de cada tipo de colección, las clases de implementación y los *enumeradores* que se utilizan para “recorrer” las colecciones.

El .NET Framework proporciona tres espacios de nombres dedicados a las colecciones. El espacio de nombres `System.Collections` contiene colecciones que almacenan referencias a objetos de la clase `object`. El espacio de nombres `System.Collections.Generic` más reciente contiene clases genéricas para almacenar colecciones de tipos especificados. El espacio de nombres `System.Collections.Specialized` más reciente contiene varias colecciones que soportan tipos específicos, como `string` y `bit`. Si desea aprender más acerca de este espacio de nombres, visite el sitio <http://msdn.microsoft.com/library/spa/default.asp?url=/library/SPA/cpref/html/frlrfssystemcollectionsspecialized.asp>. Las colecciones en estos espacios de nombres proporcionan componentes estandarizados y reutilizables; no necesita escribir sus propias clases de colecciones. Estas colecciones están escritas para reutilizarse ampliamente. Están optimizadas para ejecutarse con rapidez y para un uso eficiente de la memoria. A medida que se desarrollen nuevas estructuras de datos y algoritmos que se adapten a este marco de trabajo, una gran base de programadores ya estarán familiarizados con las interfaces y algoritmos implementados por esas estructuras de datos.

26.2 Generalidades acerca de las colecciones

Todas las clases de colección en el .NET Framework implementan cierta combinación de las interfaces de colección. Estas interfaces declaran las operaciones que se van a realizar de manera genérica en varios tipos de colecciones. La figura 26.1 lista algunas de las interfaces de las colecciones del .NET Framework. Todas las interfaces en la figura 26.1 se declaran en el espacio de nombres `System.Collections` y tienen equivalentes genéricas en el espacio de nombres `System.Collections.Generic`. Las implementaciones de estas interfaces se proporcionan

Interfaz	Descripción
<code>ICollection</code>	La interfaz raíz en la jerarquía de colecciones, a partir de la cual heredan las interfaces <code>ICollection</code> e <code>IDictionary</code> . Contiene una propiedad <code>Count</code> para determinar el tamaño de una colección, y un método <code>CopyTo</code> para copiar el contenido de una colección a un arreglo tradicional.
<code>ICollection</code>	Una colección ordenada que puede manipularse como un arreglo. Proporciona un indexador para acceder a los elementos con un índice <code>int</code> . También tiene métodos para buscar en una colección y modificarla, incluyendo <code>Add</code> , <code>Remove</code> , <code>Contains</code> e <code>IndexOf</code> .
<code>IDictionary</code>	Una colección de valores, indexados por un objeto “clave” arbitrario. Proporciona un indexador para acceder a los elementos con un índice <code>object</code> y métodos para modificar la colección (por ejemplo, <code>Add</code> , <code>Remove</code>). La propiedad <code>Keys</code> de <code>IDictionary</code> contiene los objetos <code>object</code> que se utilizan como índices, y la propiedad <code>Values</code> contiene todos los objetos <code>object</code> almacenados.
<code>IEnumerator</code>	Un objeto que puede enumerarse. Esta interfaz contiene exactamente un método llamado <code>GetEnumerator</code> , el cual devuelve un objeto <code>IEnumerator</code> (que veremos en la sección 26.3). <code>ICollection</code> implementa a <code>IEnumerator</code> , por lo que todas las clases de colección implementan a <code>IEnumerator</code> , ya sea en forma directa o indirecta.

Figura 26.1 | Algunas interfaces de colección comunes.

dentro del marco de trabajo. Los programadores también pueden proporcionar implementaciones específicas a sus propios requerimientos.

En versiones anteriores de C#, el .NET Framework proporcionaba principalmente las clases de colección en los espacios de nombres `System.Collections` y `System.Collections.Specialized`. Estas clases almacenaban y manipulaban referencias `object`. Podíamos almacenar cualquier `object` en una colección. Un aspecto inconveniente en cuanto a ordenar referencias `object` ocurre cuando se obtienen de una colección. Por lo general, una aplicación necesita procesar tipos específicos de objetos. Como resultado, las referencias `object` que se obtienen de una colección necesitan comúnmente de una conversión descendente a un tipo apropiado, para que la aplicación pueda procesar los objetos en forma correcta.

El .NET Framework 2.0 ahora incluye también el espacio de nombres `System.Collections.Generic`, el cual utiliza las capacidades de los genéricos que presentamos en el capítulo 25. Muchas de estas nuevas clases son simplemente contrapartes genéricas de las clases en el espacio de nombres `System.Collections`. Esto significa que puede especificar el tipo exacto que se almacenará en una colección. También obtiene los beneficios de la comprobación de tipos en tiempo de compilación; el compilador asegura que se estén utilizando los tipos apropiados con su colección y, en caso contrario, genera mensajes de error en tiempo de compilación. Además, una vez que se especifica el tipo almacenado en una colección, cualquier elemento que se obtenga de ésta tendrá el tipo correcto. Esto elimina la necesidad de conversiones de tipo explícitas, que pueden lanzar excepciones `InvalidCastException` en tiempo de ejecución si el objeto referenciado no es del tipo apropiado. Esto también elimina la sobrecarga de la conversión explícita, con lo cual se mejora la eficiencia.

En este capítulo mostraremos seis clases de colección: **`Array`**, **`ArrayList`**, **`Stack`**, **`Hashtable`**, **`SortedDictionary`** genérica y **`LinkedList`** genérica, más las herramientas de arreglos integradas. El espacio de nombres `System.Collections` proporciona varias estructuras de datos más, incluyendo **`BitArray`** (una colección de valores verdadero/falso), **`Queue`** y **`SortedList`** (una colección de pares clave/valor que se ordenan por clave y se puede acceder a ellos ya sea por clave, o por índice). La figura 26.2 muestra un resumen de muchas de las clases de colección. También hablaremos sobre la interfaz `IEnumerator`. Las clases de colección pueden crear enumeradores para que los programadores puedan recorrer las colecciones. Aunque estos enumeradores tienen distintas implementaciones, todos implementan la interfaz `IEnumerator`, por lo que pueden procesarse mediante el polimorfismo. Como veremos pronto, la instrucción `foreach` es simplemente una notación conveniente para usar un enumerador. En la siguiente sección, empezaremos nuestra discusión con un análisis de los enumeradores y las herramientas de las colecciones para la manipulación de arreglos.

Clase	Implementa a	Descripción
<i>Espacio de nombres System:</i>		
Array	ICollection	La clase base de todos los arreglos convencionales. Vea la sección 26.3.
<i>Espacio de nombres System.Collections:</i>		
ArrayList	ICollection	Imita los arreglos convencionales, pero crece o se reduce según sea necesario para adaptarse al número de elementos. Vea la sección 26.4.1.
BitArray	ICollection	Un arreglo de elementos bool que hace un uso eficiente de la memoria.
Hashtable	IDictionary	Una colección desordenada de pares clave-valor, a los que se puede acceder por la clave. Vea la sección 26.4.3.
Queue	ICollection	Una colección PEPS (primero en entrar, primero en salir). Vea la sección 24.6, Colas.
SortedList	IDictionary	Una clase Hashtable genérica que ordena los datos por claves, a la cual se puede acceder ya sea por clave o por índice.
Stack	ICollection	Una colección UEPS (último en entrar, primero en salir). Vea la sección 26.4.2.
<i>Espacio de nombres System.Collections.Generic:</i>		
Dictionary< K, E >	IDictionary< K, E >	Una colección genérica desordenada de pares clave-valor, a la cual se puede acceder por clave.
LinkedList< E >	ICollection< E >	Una lista doblemente enlazada. Vea la sección 26.5.2.
List< E >	ICollection< E >	Una clase ArrayList genérica.
Queue< E >	ICollection< E >	Una clase Queue genérica.
SortedDictionary< K, E >	IDictionary< K, E >	Una clase Dictionary que ordena los datos por las claves en un árbol binario. Vea la sección 26.5.1.
SortedList< K, E >	IDictionary< K, E >	Una clase SortedList genérica.
Stack< E >	ICollection< E >	Una clase Stack genérica.
[Nota: Todas las clases de colección implementan de forma directa o indirecta a ICollection y a IEnumerable (o a las interfaces genéricas equivalentes ICollection< E > y IEnumerable< E > para las colecciones genéricas).]		

Figura 26.2 | Algunas clases de colección del .NET Framework.

26.3 La clase Array y los enumeradores

El capítulo 8 presentó las herramientas básicas de procesamiento de arreglos. Todos los arreglos heredan en forma implícita de la clase base abstract Array (espacio de nombres System), el cual define la propiedad Length, que especifica el número de elementos en el arreglo. Además, la clase Array proporciona métodos static, los cuales tienen algoritmos para procesar arreglos. Por lo general, la clase Array sobrecarga estos métodos; por ejemplo, el método Reverse de Array puede invertir el orden de los elementos en todo un arreglo completo, o

puede invertir los elementos en un rango especificado de elementos en un arreglo. Para una lista completa de los métodos `static` de la clase `Array`, visite la página:

[msdn2.microsoft.com/es-es/library/system.array\(VS.80\).aspx](https://msdn2.microsoft.com/es-es/library/system.array(VS.80).aspx)

La figura 26.3 demuestra varios métodos `static` de la clase `Array`.

```

1  // Fig. 26.3: UsoDeArray.cs
2  // Métodos static de la clase Array para manipulaciones de arreglos comunes.
3  using System;
4  using System.Collections;
5
6  // demuestra los algoritmos de la clase Array
7  public class UsoDeArray
8  {
9      private static int[] valoresInt = { 1, 2, 3, 4, 5, 6 };
10     private static double[] valoresDouble = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11     private static int[] copiaValoresInt;
12
13     // el método Main demuestra los métodos de la clase Array
14     public static void Main( string[] args )
15     {
16         copiaValoresInt = new int[ valoresInt.Length ]; // los valores predeterminados
17         // son ceros
18
19         Console.WriteLine( "Valores iniciales del arreglo:\n" );
20         ImprimirArreglos(); // imprime el contenido inicial del arreglo
21
22         // ordena valoresDouble
23         Array.Sort( valoresDouble );
24
25         // copia valoresInt a copiaValoresInt
26         Array.Copy( valoresInt, copiaValoresInt, valoresInt.Length );
27
28         Console.WriteLine( "\nValores del arreglo después de Sort y Copy:\n" );
29         ImprimirArreglos(); // imprime el contenido del arreglo
30         Console.WriteLine();
31
32         // busca el 5 en valoresInt
33         int resultado = Array.BinarySearch( valoresInt, 5 );
34         if ( resultado >= 0 )
35             Console.WriteLine( "Se encontró el 5 en el elemento {0} de valoresInt",
36                               resultado );
37         else
38             Console.WriteLine( "No se encontró el 5 en valoresInt" );
39
40         // busca el 8763 en valoresInt
41         resultado = Array.BinarySearch( valoresInt, 8763 );
42         if ( resultado >= 0 )
43             Console.WriteLine( "Se encontró el 8763 en el elemento {0} de valoresInt",
44                               resultado );
45         else
46             Console.WriteLine( "No se encontró el 8763 en valoresInt" );
47     } // fin del método Main
48
49     // imprime el contenido del arreglo con enumeradores
50     private static void ImprimirArreglos()

```

Figura 26.3 | Clase `Array` que se utiliza para realizar muchas manipulaciones comunes de arreglos. (Parte 1 de 2).


```

50     {
51         Console.Write( "valoresDouble: " );
52
53         // itera a través del arreglo double con un enumerador
54         IEnumerator enumerador = valoresDouble.GetEnumerator();
55
56         while ( enumerador.MoveNext() )
57             Console.Write( enumerador.Current + " " );
58
59         Console.Write( "\nvaloresInt: " );
60
61         // itera a través del arreglo int con un enumerador
62         enumerador = valoresInt.GetEnumerator();
63
64         while ( enumerador.MoveNext() )
65             Console.Write( enumerador.Current + " " );
66
67         Console.Write( "\ncopiaValoresInt: " );
68
69         // itera a través del segundo arreglo int con una instrucción foreach
70         foreach ( int elemento in copiaValoresInt )
71             Console.Write( elemento + " " );
72
73         Console.WriteLine();
74     } // fin del método ImprimirArreglos
75 } // fin de la clase UsoDeArray

```

Valores iniciales del arreglo:

```

valoresDouble: 8.4 9.3 0.2 7.9 3.4
valoresInt: 1 2 3 4 5 6
copiaValoresInt: 0 0 0 0 0 0

```

Valores del arreglo después de Sort y Copy:

```

valoresDouble: 0.2 3.4 7.9 8.4 9.3
valoresInt: 1 2 3 4 5 6
copiaValoresInt: 1 2 3 4 5 6

```

Se encontró el 5 en el elemento 4 de valoresInt
No se encontró el 8763 en valoresInt

Figura 26.3 | Clase Array que se utiliza para realizar muchas manipulaciones comunes de arreglos. (Parte 2 de 2).

Las directivas `using` en las líneas 3-4 incluyen los espacios de nombres `System` (para las clases `Array` y `Console`) y `System.Collections` (para la interfaz `IEnumerator`, que veremos en breve). Las referencias a los ensamblados para estos espacios de nombres se incluyen de manera implícita en cada aplicación, por lo que no es necesario agregar nuevas referencias al archivo de proyecto.

Nuestra clase de prueba declara tres variables tipo arreglo `static` (líneas 9-11). Las primeras dos líneas inicializan `valoresInt` y `valoresDouble` con un arreglo `int` y un `double`, respectivamente. El fin de la variable estática `copiaValoresInt` es demostrar el método `Copy` de `Array`, por lo que se deja con el valor predeterminado `null`; todavía no se refiere a un arreglo.

La línea 16 inicializa `copiaValoresInt` con un arreglo `int` que tiene la misma longitud que el arreglo `valoresInt`. La línea 19 llama al método `ImprimirArreglos` (líneas 49-74) para imprimir en pantalla el contenido inicial de los tres arreglos. En breve hablaremos sobre el método `ImprimirArreglos`. De los resultados de la figura 26.3 podemos ver que cada elemento del arreglo `copiaValoresInt` se inicializa con el valor predeterminado 0.

La línea 22 utiliza el método `static Sort` de `Array` para ordenar el arreglo `valoresDouble`. Cuando este método regresa, el arreglo contiene sus elementos originales, ordenados en forma ascendente.

La línea 25 utiliza el método `static Copy` de `Array` para copiar elementos del arreglo `valoresInt` al arreglo `copiaValoresInt`. El primer argumento es el arreglo a copiar (`valoresInt`), el segundo argumento es el arreglo de destino (`copiaValoresInt`) y el tercer argumento es un `int` que representa el número de elementos a copiar (en este caso, `valoresInt.Length` especifica a todos los elementos).

Las líneas 32 y 40 invocan al método `static BinarySearch` de `Array` para realizar búsquedas binarias en el arreglo `valoresInt`. El método `BinarySearch` recibe el arreglo *ordenado* en el que se realizará la búsqueda, y la clave que debe buscar. El método devuelve el índice en el arreglo en donde encuentra la clave (o un número negativo si no la encontró). Observe que `BinarySearch` asume que recibe un arreglo ordenado. Su comportamiento con un arreglo desordenado es impredecible.



Error común de programación 26.1

Pasar un arreglo desordenado a `BinarySearch` es un error lógico; el valor que se devuelve no está definido.

El método `ImprimirArreglos` (líneas 49-74) utiliza los métodos de la clase `Array` para iterar a través de cada arreglo. En la línea 54, el método `GetEnumerator` obtiene un enumerador para el arreglo `valoresInt`. Recuerde que `Array` implementa a la interfaz `IEnumerable`. Todos los arreglos heredan implícitamente de `Array`, por lo que los tipos de arreglos `int[]` y `double[]` implementan el método `GetEnumerator` de la interfaz `IEnumerable`, el cual devuelve un enumerador que puede iterar a través de la colección. La interfaz `IEnumerator` (que todos los enumeradores implementan) define los métodos `MoveNext` y `Reset`, junto con la propiedad `Current`. `MoveNext` desplaza el enumerador hasta el siguiente elemento en la colección. La primera llamada a `MoveNext` posiciona el enumerador en el primer elemento de la colección. `MoveNext` devuelve `true` si hay por lo menos un elemento más en la colección; en caso contrario, el método devuelve `false`. El método `Reset` posiciona el enumerador antes del primer elemento de la colección. Los métodos `MoveNext` y `Reset` lanzan una excepción `InvalidOperationException` si el contenido de la colección se modifica en cualquier forma, antes de crear el enumerador. La propiedad `Current` devuelve el objeto en la ubicación actual en la colección.



Error común de programación 26.2

Si se modifica una colección después de crear un enumerador para esa colección, el enumerador se vuelve inválido de inmediato; cualquier método que se llame con el enumerador después de este punto lanzará excepciones `InvalidOperationException`. Por esta razón, se dice que los enumeradores son de “falla rápida”.

Cuando el método `GetEnumerator` devuelve un enumerador en la línea 54, al principio se posiciona *antes* del primer elemento en el arreglo `valoresDouble`. Después, cuando la línea 56 llama a `MoveNext` en la primera iteración del ciclo `while`, el enumerador avanza al primer elemento en `valoresDouble`. La instrucción `while` en las líneas 56-57 itera a través de cada elemento, hasta que el enumerador pasa el final de `valoresDouble` y `MoveNext` devuelve `false`. En cada iteración, utilizamos la propiedad `Current` del enumerador para obtener e imprimir en pantalla el elemento actual del arreglo. Las líneas 62-65 iteran a través del arreglo `valoresInt`.

Observe que `ImprimirArreglos` se llama dos veces (líneas 19 y 28), por lo que `GetEnumerator` se llama dos veces en `valoresDouble`. El método `GetEnumerator` (líneas 54 y 62) siempre devuelve un enumerador que se posiciona antes del primer elemento. Observe además que la propiedad `Current` de `IEnumerator` es de sólo lectura. Los enumeradores no pueden utilizarse para modificar el contenido de las colecciones, sólo para obtenerlo.

Las líneas 70-71 utilizan una instrucción `foreach` para iterar a través de los elementos de la colección como un enumerador. De hecho, esta instrucción se comporta de la misma forma que un enumerador. Ambos iteran a través de los elementos de un arreglo, uno por uno, en un orden bien definido. Ninguno le permite modificar los elementos durante la iteración. Ésta no es una coincidencia. La instrucción `foreach` obtiene de manera implícita un enumerador a través del método `GetEnumerator`, y utiliza el método `MoveNext` y la propiedad `Current` del enumerador para recorrer la colección, de la misma forma que se hizo explícitamente en las líneas 54-57. Por esta razón, podemos utilizar la instrucción `foreach` para iterar a través de *cualquier* colección que implemente la interfaz `IEnumerable`, y no sólo arreglos. En la siguiente sección mostraremos esta funcionalidad, cuando hablemos sobre la clase `ArrayList`.

Otros métodos `static` de `Array` incluyen a **`Clear`** (para establecer un rango de elementos a 0 o `null`), **`CreateInstance`** (para crear un nuevo arreglo de un tipo especificado), **`IndexOf`** (para localizar la primera ocurrencia de un objeto en un arreglo, o en una porción del mismo), **`LastIndexOf`** (para localizar la última ocurrencia de un objeto en un arreglo, o en una porción del mismo) y **`Reverse`** (para invertir el contenido de un arreglo, o de una porción del mismo).

26.4 Colecciones no genéricas

El espacio de nombres `System.Collections` en la Biblioteca de clases del .NET Framework es la fuente principal de las colecciones no genéricas. Estas clases proporcionan implementaciones estándar de muchas de las estructuras de datos que vimos en el capítulo 24, con colecciones que almacenan referencias de tipo `object`. En esta sección, mostraremos las clases `ArrayList`, `Stack` y `Hashtable`.

26.4.1 La clase `ArrayList`

En la mayoría de los lenguajes de programación, los arreglos convencionales tienen un tamaño fijo; no pueden cambiarse en forma dinámica para conformarse a los requerimientos de memoria en tiempo de ejecución de una aplicación. En algunas aplicaciones, esta limitación del tamaño fijo presenta un problema para los programadores, ya que deben elegir entre usar arreglos de tamaño fijo que sean lo bastante grandes como para almacenar el máximo número de elementos que pueda requerir la aplicación, o utilizar estructuras de datos dinámicas que puedan aumentar y reducir la cantidad de memoria requerida para almacenar datos, en respuesta a los requerimientos cambiantes de una aplicación en tiempo de ejecución.

La colección **`ArrayList`** del .NET Framework imita la funcionalidad de los arreglos convencionales y proporciona la capacidad de modificar el tamaño de la colección en forma dinámica, a través de sus métodos. En cualquier momento dado, un objeto `ArrayList` contiene un cierto número de elementos menos que, o iguales a, su **capacidad**: el número de elementos reservados en un momento dado para el objeto `ArrayList`. Una aplicación puede manipular la capacidad mediante la propiedad `Capacity` de `ArrayList`. Si un objeto `ArrayList` necesita crecer, duplica su capacidad de manera predeterminada.



Tip de rendimiento 26.1

Al igual que con las listas enlazadas, insertar elementos adicionales en un objeto `ArrayList` cuyo tamaño actual sea menor que su capacidad es una operación rápida.



Tip de rendimiento 26.2

Insertar un elemento en un objeto `ArrayList` que necesite crecer más para acomodarlo es una operación lenta. Un objeto `ArrayList` que se encuentra al límite de su capacidad debe reasignar su memoria y los valores existentes que están copiados en la misma.



Tip de rendimiento 26.3

Si el almacenamiento es de máxima importancia, use el método `TrimToSize` de la clase `ArrayList` para recortar un objeto `ArrayList` a su tamaño exacto. Esto optimizará el uso de la memoria del objeto `ArrayList`. Pero debe tener cuidado; si la aplicación necesita insertar elementos, el proceso será más lento debido a que el objeto `ArrayList` debe crecer en forma dinámica (el recorte no deja espacio para que crezca).



Tip de rendimiento 26.4

Tal vez parezca que el incremento predeterminado de la capacidad de un objeto `ArrayList` (duplicar su tamaño) desperdicia espacio de almacenamiento, pero duplicar el tamaño es una manera eficiente de que un objeto `ArrayList` crezca con rapidez al tamaño más apropiado. Éste es un uso mucho más eficiente de tiempo que aumentar el tamaño del objeto `ArrayList` un elemento a la vez, en respuesta a las operaciones de inserción.

Los objetos `ArrayList` almacenan referencias a objetos de la clase `object`. Todas las clases se derivan de la clase `object`, por lo que un objeto `ArrayList` puede contener objetos de cualquier tipo. La figura 26.4 lista algunos métodos y propiedades útiles de la clase `ArrayList`.

Método o propiedad	Descripción
Add	Agrega un objeto <code>object</code> al objeto <code>ArrayList</code> y devuelve un <code>int</code> que especifica el índice en el que se agregó el objeto <code>object</code> .
Capacity	Propiedad que obtiene y establece el número de elementos para los que se reserva espacio en un momento dado, dentro del objeto <code>ArrayList</code> .
Clear	Elimina todos los elementos del objeto <code>ArrayList</code> .
Contains	Devuelve <code>true</code> si el objeto <code>object</code> especificado está en el objeto <code>ArrayList</code> ; en caso contrario, devuelve <code>false</code> .
Count	Propiedad de sólo lectura que obtiene el número de elementos almacenados en el objeto <code>ArrayList</code> .
IndexOf	Devuelve el índice de la primera ocurrencia del objeto <code>object</code> especificado en el objeto <code>ArrayList</code> .
Insert	Inserta un objeto <code>object</code> en el índice especificado.
Remove	Elimina la primera ocurrencia del objeto <code>object</code> especificado.
RemoveAt	Elimina un objeto en el índice especificado.
RemoveRange	Elimina un número especificado de elementos, empezando en un índice especificado en el objeto <code>ArrayList</code> .
Sort	Ordena el objeto <code>ArrayList</code> .
TrimToSize	Establece la capacidad del objeto <code>ArrayList</code> al número de elementos que contiene el objeto <code>ArrayList</code> en un momento dado (<code>Count</code>).

Figura 26.4 | Algunos métodos y propiedades de la clase `ArrayList`.

La figura 26.5 demuestra el uso de la clase `ArrayList` y varios de sus métodos. La clase `ArrayList` pertenece al espacio de nombres `System.Collections` (línea 4). Las líneas 8-11 declaran dos arreglos de objetos `string` (`colores` y `eliminarColores`), que utilizaremos para llenar dos objetos `ArrayList`. En la sección 9.11 vimos que las constantes deben inicializarse en tiempo de compilación, pero las variables de sólo lectura (`readonly`) pueden inicializarse en tiempo de ejecución. Los arreglos son objetos creados en tiempo de ejecución, por lo que declaramos a `colores` y a `eliminarColores` con `readonly` (no `const`) para que no se puedan modificar. Cuando la aplicación empieza a ejecutarse, creamos un objeto `ArrayList` con una capacidad inicial de un elemento, y lo almacenamos en la variable `lista` (línea 16). La instrucción `foreach` en las líneas 20-21 agrega los cinco elementos del arreglo `colores` a `lista`, a través del método **Add** de `ArrayList`, por lo que `lista` crece para dar cabida a estos nuevos elementos. La línea 25 utiliza el constructor sobrecargado de `ArrayList` para crear un nuevo objeto `ArrayList`, el cual se inicializa con el contenido del arreglo `eliminarColores` y después lo asigna a la variable `eliminarLista`. Este constructor puede inicializar el contenido de un objeto `ArrayList` con los elementos de cualquier objeto `ICollection` que se le pase como parámetro. Muchas de las clases de colección tienen un constructor similar. Observe que la llamada al constructor en la línea 25 realiza la tarea de las líneas 20-21.

La línea 28 llama al método `MostrarInformacion` (líneas 38-55) para imprimir en pantalla el contenido de la `lista`. Este método utiliza una instrucción `foreach` para recorrer los elementos de un objeto `ArrayList`. Como vimos en la sección 26.3, la instrucción `foreach` es una abreviación conveniente para llamar al método `GetEnumerator` de `ArrayList` y utilizar un enumerador para recorrer los elementos de la colección. Además, debemos usar una variable de iteración de tipo `object`, ya que la clase `ArrayList` no es genérica y almacena referencias a objetos `object`.

Utilizamos las propiedades **Count** y **Capacity** en la línea 46 para mostrar el número actual de elementos y el máximo número de elementos que pueden almacenarse sin asignar más memoria al objeto `ArrayList`. Los resultados de la figura 26.5 indican que el objeto `ArrayList` tiene capacidad de 8; recuerde que un objeto `ArrayList` duplica su capacidad cuando necesita más espacio.

```

1  // Fig. 26.5: PruebaArrayList.cs
2  // Uso de la clase ArrayList.
3  using System;
4  using System.Collections;
5
6  public class PruebaArrayList
7  {
8      private static readonly string[] colores =
9          { "MAGENTA", "ROJO", "BLANCO", "AZUL", "CYAN" };
10     private static readonly string[] eliminarColores =
11         { "ROJO", "BLANCO", "AZUL" };
12
13     // crea un objeto ArrayList, le agrega los colores y lo manipula
14     public static void Main( string[] args )
15     {
16         ArrayList lista = new ArrayList( 1 ); // capacidad inicial de 1
17
18         // agrega los elementos del arreglo colores
19         // al objeto ArrayList lista
20         foreach ( string color in colores )
21             lista.Add( color ); // agrega el color al objeto ArrayList lista
22
23         // agrega los elementos en el arreglo eliminarColores al
24         // objeto ArrayList eliminarLista con el constructor de ArrayList
25         ArrayList eliminarLista = new ArrayList( eliminarColores );
26
27         Console.WriteLine( "ArrayList: " );
28         MostrarInformacion( lista ); // imprime la lista en pantalla
29
30         // elimina del objeto ArrayList lista los colores en eliminarLista
31         EliminarColores( lista, eliminarLista );
32
33         Console.WriteLine( "\nArrayList después de llamar a EliminarColores: " );
34         MostrarInformacion( lista ); // imprime en pantalla el contenido de lista
35     } // fin del método Main
36
37     // muestra información sobre el contenido de un arreglo llamado lista
38     private static void MostrarInformacion( ArrayList arregloLista )
39     {
40         // itera a través del arreglo lista con una instrucción foreach
41         foreach ( object elemento in arregloLista )
42             Console.Write( "{0} ", elemento ); // invoca a ToString
43
44         // muestra el tamaño y la capacidad
45         Console.WriteLine( "\nTamaño = {0}; Capacidad = {1}",
46             arregloLista.Count, arregloLista.Capacity );
47
48         int index = arregloLista.IndexOf( "AZUL" );
49
50         if ( index != -1 )
51             Console.WriteLine( "El arreglo lista contiene AZUL en el índice {0}.",
52                 index );
53         else
54             Console.WriteLine( "El arreglo lista no contiene AZUL." );
55     } // fin del método MostrarInformacion
56
57     // elimina de primeraLista los colores especificados en segundaLista
58     private static void EliminarColores( ArrayList primeraLista,
59         ArrayList segundaLista )

```

Figura 26.5 | Uso de la clase ArrayList. (Parte I de 2).

```

60     {
61         // itera a través del segundo objeto ArrayList como si fuera un arreglo
62         for ( int cuenta = 0; cuenta < segundaLista.Count; cuenta++ )
63             primeraLista.Remove( segundaLista[ cuenta ] );
64     } // fin del método EliminarColores
65 } // fin de la clase PruebaArrayList

```

ArrayList:
MAGENTA ROJO BLANCO AZUL CYAN
Tamaño = 5; Capacidad = 8
El arreglo lista contiene AZUL en el índice 3.

ArrayList después de llamar a EliminarColores:
MAGENTA CYAN
Tamaño = 2; Capacidad = 8
El arreglo lista no contiene AZUL.

Figura 26.5 | Uso de la clase ArrayList. (Parte 2 de 2).

En la línea 48, invocamos al método **IndexOf** para determinar la posición del objeto string "AZUL" en arregloLista y almacenar el resultado en la variable local index. El método IndexOf devuelve -1 si no se encuentra el elemento. La instrucción if en las líneas 50-54 comprueba si índice es -1 para determinar si arregloLista contiene "AZUL". Si es así, se imprime su índice en pantalla. La clase ArrayList también cuenta con el método **Contains**, el cual simplemente devuelve true si un objeto se encuentra en el objeto ArrayList, y false en caso contrario. El método Contains es más conveniente si no necesitamos el índice del elemento.



Tip de rendimiento 26.5

Los métodos **IndexOf** y **Contains** de ArrayList realizan una búsqueda lineal, la cual es una operación costosa para objetos ArrayList extensos. Si el objeto ArrayList está ordenado, use el método **BinarySearch** para realizar una búsqueda más eficiente. Este método devuelve el índice del elemento, o un número negativo si no se encontró ese elemento.

Una vez que el método **MostrarInformacion** regresa, llamamos al método **EliminarColores** (líneas 58-64) con los dos objetos ArrayList. La instrucción for en las líneas 62-63 itera a través del arreglo ArrayList segundaLista. La línea 63 utiliza un indexador para acceder a un elemento del objeto ArrayList; para ello, coloca después del nombre de la referencia ArrayList dos corchetes ([]) que contienen el índice deseado del elemento. Si el índice especificado no es mayor que 0 y menor que el número de elementos almacenados actualmente en el objeto ArrayList (que se especifican mediante la propiedad Count de ArrayList), se produce una excepción **ArgumentOutOfRangeException**.

Utilizamos el indexador para obtener cada uno de los elementos de segundaLista y después eliminamos cada uno de primeraLista mediante el método **Remove**. Este método elimina un elemento especificado de un objeto ArrayList, para lo cual realiza una búsqueda lineal y elimina (sólo) la primera ocurrencia del objeto especificado. Todos los elementos subsiguientes se desplazan hacia el inicio del objeto ArrayList, para llenar la posición vacía.

Después de la llamada a **EliminarColores**, la línea 34 vuelve a imprimir en pantalla el contenido de lista, confirmando que se eliminaron los elementos de eliminarLista.

26.4.2 La clase Stack

La clase Stack implementa a una estructura de datos tipo pila, y proporciona gran parte de la funcionalidad que definimos en nuestra propia implementación en la sección 24.3. Puede regresar a esa sección para recordar los conceptos de la estructura de datos tipo pila. En la figura 24.11 creamos una aplicación de prueba para demostrar la estructura de datos **HerenciaPila** que desarrollamos. En la figura 26.6 adaptamos la figura 24.14 para demostrar la clase de colección Stack del .NET Framework.

La directiva `using` en la línea 4 nos permite utilizar la clase `Stack` con su nombre no calificado del espacio de nombres `System.Collections`. La línea 10 crea un objeto `Stack` con la capacidad inicial predeterminada (10 elementos). Como es de esperarse, la clase `Stack` tiene los métodos `Push` y `Pop` para realizar las operaciones básicas de una pila.

El método `Push` recibe un objeto `object` como argumento y lo inserta en la parte superior del objeto `Stack`. Si el número de elementos en el objeto `Stack` (la propiedad `Count`) es igual a la capacidad al momento de la operación `Push`, el objeto `Stack` crece para dar cabida a más objetos `object`. Las líneas 19-26 usan el método `Push` para agregar cuatro elementos (`bool`, `char`, `int` y `string`) a la pila e invocar al método `ImprimirPila` (líneas 50-64) después de cada operación `Push`, para imprimir en pantalla el contenido de la pila. Observe que esta clase `Stack` no genérica sólo puede almacenar referencias a objetos `object`, por lo que aplica una conversión `boxing` a cada uno de los elementos de tipo de valor (`bool`, `char` e `int`) de manera implícita, antes de agregarlos al objeto `Stack`. (El espacio de nombres `System.Collections.Generic` proporciona una clase `Stack` genérica, que tiene muchos de los mismos métodos y propiedades que se utilizan en la figura 26.6.)

El método `ImprimirPila` (líneas 50-64) usa la propiedad `Count` de `Stack` (que se implementó para cumplir con los requisitos de la interfaz `ICollection`) para obtener el número de elementos en pila. Si la pila no está vacía (es decir, si `Count` no es igual a 0), utilizamos una instrucción `foreach` para iterar a través de la pila e imprimir su contenido, invocando en forma implícita al método `ToString` de cada elemento. La instrucción `foreach` invoca en forma implícita al método `GetEnumerator` de `Pila`, que podríamos haber llamado explícitamente para recorrer la pila a través de un enumerador.

El método `Peek` devuelve el valor del elemento superior de la pila, pero no elimina ese elemento de la misma. Utilizamos `Peek` en la línea 30 para obtener el objeto superior del objeto `Stack`, y después imprimimos en pantalla ese objeto, invocando en forma implícita al método `ToString` del mismo. Si el objeto `Stack` está vacío cuando la aplicación llama a `Peek`, se produce una excepción `InvalidOperationException`. (No necesitamos un bloque de manejo de excepciones, ya que sabemos que aquí la pila no está vacía.)

```

1  // Fig. 26.6: PruebaStack.cs
2  // Demostración de la clase Stack.
3  using System;
4  using System.Collections;
5
6  public class PruebaStack
7  {
8      public static void Main( string[] args )
9      {
10         Stack pila = new Stack(); // capacidad predeterminada de 10
11
12         // crea objetos para almacenarlos en la pila
13         bool unBooleano = true;
14         char unCaracter = '$';
15         int unEntero = 34567;
16         string unaCadena = "hola";
17
18         // usa el método Push para agregar elementos a (la parte superior de) la pila
19         pila.Push( unBooleano );
20         ImprimirStack( pila );
21         pila.Push( unCaracter );
22         ImprimirStack( pila );
23         pila.Push( unEntero );
24         ImprimirStack( pila );
25         pila.Push( unaCadena );
26         ImprimirStack( pila );
27
28         // comprueba el elemento superior de la pila
29         Console.WriteLine( "El elemento superior de la pila es {0}\n",

```

Figura 26.6 | Demostración de la clase `Stack`. (Parte 1 de 2).

```

30     pila.Peek() );
31
32     // elimina elementos de la pila
33     try
34     {
35         while ( true )
36         {
37             object objetoEliminado = pila.Pop();
38             Console.WriteLine( "Se sacó " + objetoEliminado );
39             ImprimirStack( pila );
40         } // fin de while
41     } // fin de try
42     catch ( InvalidOperationException excepcion )
43     {
44         // si ocurre una excepción, imprime el rastreo de pila
45         Console.Error.WriteLine( excepcion );
46     } // fin de catch
47 } // fin de Main
48
49 // imprime el contenido de una pila
50 private static void ImprimirStack( Stack pila )
51 {
52     if ( pila.Count == 0 )
53         Console.WriteLine( "La pila está vacía\n" ); // la pila está vacía
54     else
55     {
56         Console.Write( "La pila es: " );
57
58         // itera a través de la pila mediante una instrucción foreach
59         foreach ( object elemento in pila )
60             Console.Write( "{0} ", elemento ); // invoca a ToString
61
62         Console.WriteLine( "\n" );
63     } // fin de else
64 } // fin del método ImprimirStack
65 } // fin de la clase PruebaStack

```

```

La pila es: True
La pila es: $ True
La pila es: 34567 $ True
La pila es: hola 34567 $ True
El elemento superior de la pila es hola
Se sacó hola
La pila es: 34567 $ True
Se sacó 34567
La pila es: $ True
Se sacó $
La pila es: True
Se sacó True
La pila está vacía
System.InvalidOperationException: Pila vacía.
    en System.Collections.Stack.Pop()
    en PruebaStack.Main(String[] args) en C:\MisProyectos\
    fig26_06\PruebaStack\PruebaStack.cs:línea 37

```

Figura 26.6 | Demostración de la clase Stack. (Parte 2 de 2).

El método `Pop` no recibe argumentos; elimina y devuelve el objeto que se encuentra en ese momento en la parte superior del objeto `Stack`. Un ciclo infinito (líneas 35-40) saca objetos de la pila y los imprime en pantalla, hasta que la pila se vacía. Cuando la aplicación llama a `Pop` con la pila vacía, se lanza una excepción `InvalidOperationException`. El bloque `catch` (líneas 42-46) imprime la excepción en forma implícita, invocando al método `ToString` de `InvalidOperationException` para obtener un mensaje de error y el rastreo de pila.



Error común de programación 26.3

Al tratar de utilizar los métodos `Peek` o `Pop` con un objeto `Stack` vacío (una pila cuya propiedad `Count` es 0) se produce una excepción `InvalidOperationException`.

Aunque la figura 26.6 no lo demuestra, la clase `Stack` también tiene el método `Contains`, que devuelve `true` si el objeto `Stack` contiene el objeto especificado, y `false` en caso contrario.

26.4.3 La clase `Hashtable`

Cuando una aplicación crea objetos de tipos nuevos o ya existentes, necesita administrarlos en forma eficiente. Esto incluye los procesos de ordenar y obtener los objetos. Las acciones de ordenar y obtener información con los arreglos son eficientes si algún aspecto de nuestros datos coincide directamente con el valor clave, y si esas claves son únicas y están estrechamente empaquetadas. Si tenemos 100 empleados con números de Seguro Social de nueve dígitos, y queremos almacenar y recuperar los datos de los empleados mediante el uso del número de Seguro Social como clave, por lo general se requeriría un arreglo con 999,999,999, ya que hay 999,999,999 números únicos de nueve dígitos. Si tenemos un arreglo de ese tamaño, podríamos tener un rendimiento muy alto al ordenar y obtener los registros de los empleados con sólo utilizar el número de Seguro Social como índice para el arreglo, pero sería un gran desperdicio de memoria.

Muchas aplicaciones tienen este problema; o las claves son del tipo incorrecto (es decir, no son números no negativos) o son del tipo correcto, pero están muy esparcidas a lo largo de un extenso rango.

Lo que se necesita es un esquema de alta velocidad para convertir claves como los números de Seguro Social y los números de pieza en los inventarios en índices únicos para el arreglo. Así, cuando una aplicación necesite almacenar algo, el esquema podría convertir la clave de la aplicación rápidamente en un índice, y el registro de información podría almacenarse en esa ubicación del arreglo. Para obtener datos se hace lo mismo: una vez que la aplicación tiene una clave para la que desea obtener el registro de datos, simplemente aplica la conversión a la clave, lo cual produce el subíndice del arreglo en donde residen los datos, y recupera esos datos.

El esquema que describimos aquí es la base de una técnica conocida como *hashing*, en la que almacenamos datos en una estructura de datos conocida como *tabla de hash*. ¿Por qué el nombre? Porque cuando convertimos una clave en un subíndice de un arreglo, literalmente revolvemos los bits, haciendo un “picadillo (hash)” del número. En realidad, este número no tiene un significado real más allá de su utilidad para almacenar y obtener este registro de datos específico.

Hay un ligero defecto en este esquema cuando ocurren *conflictos* (es decir, dos claves distintas se asignan a la misma celda, o elemento, en el arreglo). Como no podemos ordenar dos registros de datos distintos en el mismo espacio, necesitamos encontrar un lugar alternativo para todos los registros más allá del primero que se asignen a un subíndice particular en el arreglo. Un esquema para hacer esto es reasignar la transformación de hashing a la clave, para proporcionar una siguiente celda candidata en el arreglo. El proceso de hashing está diseñado para ser aleatorio, por lo que se supone que con sólo unos cuantos valores de hash se encontrará una celda disponible.

Otro de los esquemas utiliza un valor de hash para localizar la primera celda candidata. Si la celda está ocupada, se realiza una búsqueda lineal en las celdas subsiguientes hasta encontrar una celda disponible. Para obtener datos se realiza el mismo procedimiento: se obtiene un valor de hash para la clave, y la celda resultante se comprueba para determinar si contiene los datos deseados. Si es así, la búsqueda termina. Si no, se realiza una búsqueda lineal en las celdas subsiguientes hasta que se encuentran los datos deseados.

La solución más popular para los conflictos de tablas hash es hacer que cada celda de la tabla sea un “recipiente” hash; por lo general, una lista enlazada de todos los pares clave-valor que se asocian a esa celda. Ésta es la solución que implementa la clase `Hashtable` del .NET Framework.

El *factor de carga* afecta al rendimiento de los esquemas de hash. El factor de carga es la proporción del número de objetos almacenados en la tabla hash en relación con el número total de celdas de la tabla hash. A medida que esta proporción aumenta, la probabilidad de conflictos tiende a incrementarse.



Tip de rendimiento 26.6

*El factor de carga en una tabla hash es un ejemplo clásico de una **concesión entre espacio/tiempo**: al incrementar el factor de carga, obtenemos un mejor uso de la memoria, pero la aplicación se ejecuta con más lentitud debido al aumento en los conflictos de hash. Al reducir el factor de carga obtenemos una mayor velocidad en la aplicación, debido a que se reducen los conflictos de hash, pero obtenemos un uso más deficiente de la memoria, ya que una gran parte de la tabla hash permanece vacía.*

Los estudiantes de ciencias computacionales estudian los esquemas de hash en cursos llamados “Estructuras de datos” y “Algoritmos”. Como reconocimiento al valor del uso de hash, el .NET Framework proporciona la clase `Hashtable` para que los programadores puedan emplear con facilidad los valores de hash en sus aplicaciones.

Este concepto es en extremo importante, en relación con nuestro estudio de la programación orientada a objetos. Las clases encapsulan y ocultan la complejidad (es decir, los detalles de implementación) y ofrecen interfaces amigables para los usuarios. El proceso de diseñar clases para que realicen lo anterior con eficiencia es una de las habilidades máspreciadas en el campo de la programación orientada a objetos.

Una **función hash** realiza un cálculo que determina en qué parte de la tabla hash se van a colocar datos. La función hash se aplica a la clave en un par clave-valor de objetos. La clase `Hashtable` puede aceptar cualquier objeto como clave. Por esta razón, la clase `Object` define el método `GetHashCode`, el cual heredan todos los objetos. La mayoría de las clases que son candidatas a utilizarse como claves en una tabla hash, redefinen este método para proporcionar uno que realice cálculos eficientes de código hash para un tipo específico. Por ejemplo, un objeto `string` tiene un cálculo de código hash basado en el contenido del `string`. La figura 26.7 utiliza un objeto `Hashtable` para contar el número de ocurrencias de cada palabra en un objeto `string`.

Las líneas 4-6 contienen directivas `using` para los espacios de nombres `System` (para la clase `Console`), `System.Text.RegularExpressions` (para la clase `Regex`, que vimos en el capítulo 16, Cadenas, caracteres y

```

1  // Fig. 26.7: PruebaHashtable.cs
2  // Aplicación que cuenta el número de ocurrencias de cada palabra
3  // en una cadena y las almacena en una tabla hash.
4  using System;
5  using System.Text.RegularExpressions;
6  using System.Collections;
7
8  public class PruebaHashtable
9  {
10     public static void Main( string[] args )
11     {
12         // crea una tabla hash con base en la entrada del usuario
13         Hashtable tabla = RecolectarPalabras();
14
15         // muestra el contenido de la tabla hash
16         MostrarHashtable( tabla );
17     } // fin del método Main
18
19     // crea una tabla hash a partir de la entrada del usuario
20     private static Hashtable RecolectarPalabras()
21     {
22         Hashtable tabla = new Hashtable(); // crea una nueva tabla hash
23
24         Console.WriteLine( "Escriba una cadena: " ); // pide la entrada al usuario
25         string entrada = Console.ReadLine(); // obtiene la entrada
26
27         // divide el texto de entrada en símbolos (tokens)
28         string[] palabras = Regex.Split( entrada, @"\s+" );

```

Figura 26.7 | Aplicación que cuenta el número de ocurrencias de cada palabra en un `string` y los almacena en una tabla hash. (Parte I de 2).

```

29
30 // procesamiento de las palabras de entrada
31 foreach ( string palabra in palabras )
32 {
33     string clavePalabra = palabra.ToLower(); // obtiene palabra en minúsculas
34
35     // si la tabla hash contiene la palabra
36     if ( tabla.ContainsKey( clavePalabra ) )
37     {
38         tabla[ clavePalabra ] = ( ( int ) tabla[ clavePalabra ] ) + 1;
39     } // fin de if
40     else
41         // agrega nueva palabra con una cuenta de 1 a la tabla hash
42         tabla.Add( clavePalabra, 1 );
43 } // fin de foreach
44
45 return tabla;
46 } // fin del método RecolectarPalabras
47
48 // muestra el contenido de la tabla hash
49 private static void MostrarHashtable( Hashtable tabla )
50 {
51     Console.WriteLine( "\nEl objeto Hashtable contiene:\n{0,-12}{1,-12}",
52         "Clave:", "Valor:" );
53
54     // genera la salida para cada clave en la tabla hash
55     // al iterar a través de la propiedad Keys con una instrucción foreach
56     foreach ( object clave in tabla.Keys )
57         Console.WriteLine( "{0,-12}{1,-12}", clave, tabla[ clave ] );
58
59     Console.WriteLine( "\ntamaño: {0}", tabla.Count );
60 } // fin del método MostrarHashtable
61 } // fin de la clase PruebaHashtable

```

Escriba una cadena:

Tan quieto como un barco pintado en un oceano pintado

El objeto Hashtable contiene:

Clave: Valor:

barco	1
un	2
en	1
como	1
oceano	1
quieto	1
pintado	2
tan	1

tamaño: 8

Figura 26.7 | Aplicación que cuenta el número de ocurrencias de cada palabra en un string y los almacena en una tabla hash. (Parte 2 de 2).

expresiones regulares) y System.Collections (para la clase Hashtable). La clase PruebaHashtable declara tres métodos static. El método RecolectarPalabras (líneas 20-46) recibe como entrada un objeto string y devuelve un objeto Hashtable en el que cada valor almacena el número de veces que aparece esa palabra en el objeto string, y la palabra se utiliza para la clave. El método MostrarHashtable (líneas 49-60) muestra el objeto Hashtable que recibe en formato de columnas. El método Main (líneas 10-17) simplemente invoca a Recolectar-

Palabras (línea 13) y después pasa el objeto `Hashtable` devuelto por `RecolectarPalabras` a `MostrarHashtable` en la línea 16.

El método `RecolectarPalabras` (líneas 20-46) empieza inicializando la variable local `tabla` con un nuevo objeto `Hashtable` (línea 22), el cual tiene una capacidad inicial predeterminada de 0 elementos y un factor de carga máximo predeterminado de 1.0. Cuando el número de elementos en el objeto `Hashtable` se vuelve mayor que el número de celdas multiplicado por el factor de carga, la capacidad se incrementa en forma automática. (Este detalle de implementación es invisible para los clientes de la clase.) Las líneas 24-25 piden al usuario que introduzca una cadena de caracteres. En la línea 28 utilizamos el método `static Split` de la clase `Regex` para dividir el objeto `string`, con base en sus caracteres de espacio en blanco. Esto crea un arreglo de “palabras”, que después almacenamos en la variable local `palabras`.

La instrucción `foreach` en las líneas 31-43 itera a través de cada elemento del arreglo `palabras`. Cada palabra se convierte en minúsculas con el método `string ToLower` y después se almacena en la variable `clavePalabra` (línea 33). Después, la línea 36 llama al método `ContainsKey` de `Hashtable` para determinar si la palabra está en la tabla hash (y, por ende, tuvo una ocurrencia anterior en el objeto `string`). Si el objeto `Hashtable` no contiene una entrada para la palabra, la línea 42 utiliza el método `Add` de `Hashtable` para crear una nueva entrada en la tabla hash, con la palabra en minúsculas como la clave y un objeto que contiene 1 como el valor. Observe que la conversión boxing automática se produce cuando la aplicación pasa el entero 1 al método `Add`, ya que la tabla hash almacena tanto la clave como el valor en referencias al tipo `object`.



Error común de programación 26.4

Al utilizar el método `Add` para agregar una clave que ya exista en la tabla hash, se produce una excepción `ArgumentException`.

Si la palabra ya es una clave en la tabla hash, la línea 38 usa el indexador de `Hashtable` para obtener y establecer el valor asociado de la clave (la cuenta de palabras) en la tabla hash. Primero realizamos una conversión descendente sobre el valor obtenido por el descriptor de acceso `get`, de un `object` a un `int`. Esto provoca una conversión unboxing del valor, para poder incrementarlo en 1. Después, cuando utilizamos el descriptor de acceso `set` del indexador para asignar el valor asociado de la clave, se vuelve a aplicar una conversión boxing al valor incrementado, para que pueda almacenarse en la tabla hash.

Observe que al invocar el descriptor de acceso `get` de un indexador `Hashtable` con una clave que no existe en la tabla hash, se obtiene una referencia `null`. Al utilizar el descriptor de acceso `set` con una clave que no existe en la tabla hash se crea una nueva entrada, como si se hubiera utilizado el método `Add`.

La línea 45 devuelve la tabla hash al método `Main`, que a su vez la pasa al método `MostrarTablahash` (líneas 49-60) para mostrar todas las entradas. Este método utiliza la propiedad de sólo lectura llamada **Keys** (línea 56) para obtener un objeto `ICollection` que contenga todas las claves. Como `ICollection` extiende a `IEnumerable`, podemos usar esta colección en la instrucción `foreach` de las líneas 56-57 para iterar a través de las claves de la tabla hash. Este ciclo accede a cada clave con su valor, y los imprime en pantalla mediante el uso de la variable de iteración y el descriptor de acceso `get` de `tabla`. Cada clave y su valor se muestran en una anchura de campo de -12. La anchura del campo negativo indica que la salida se justifica a la izquierda. Como una tabla hash no está ordenada, los pares clave-valor no se muestran en ningún orden específico. La línea 59 utiliza la propiedad **Count** de `Hashtable` para obtener el número de pares clave-valor en el objeto `Hashtable`.

Las líneas 56-57 podrían haber utilizado también la instrucción `foreach` con el propio objeto `Hashtable`, en vez de utilizar la propiedad `Keys`. Si utiliza una instrucción `foreach` con un objeto `Hashtable`, la variable de iteración será de tipo **DictionaryEntry**. El enumerador de `Hashtable` (o de cualquier otra clase que implemente a **IDictionary**) utiliza la estructura `DictionaryEntry` para almacenar pares clave-valor. Esta estructura proporciona las propiedades `Key` y `Value` para obtener la clave y el valor del elemento actual. Si no necesita la clave, la clase `Hashtable` también proporciona una propiedad **Values** de sólo lectura, la cual obtiene un objeto `ICollection` de todos los valores almacenados en el objeto `Hashtable`. Podemos usar esta propiedad para iterar a través de los valores almacenados en el objeto `Hashtable`, sin necesidad de preocuparnos por ver en dónde se almacenan.

Problemas con las colecciones no genéricas

En la aplicación de conteo de palabras de la figura 26.7, nuestro objeto `Hashtable` almacena sus claves y datos como referencias `object`, aun cuando sólo almacenamos claves `string` y valores `int` por convención. Esto

produce como resultado un código extraño. Por ejemplo, en la línea 38 nos vemos forzados a realizar conversiones unboxing y boxing en los datos `int` almacenados en el objeto `Hashtable`, cada vez que se incrementa la cuenta para una clave específica. Esto es ineficiente. En la línea 56 ocurre un problema similar: la variable de iteración de la instrucción `foreach` es una referencia `object`. Si necesitamos usar cualquiera de sus métodos específicos para `string`, necesitamos una conversión descendente explícita.

Esto puede provocar errores sutiles. Suponga que decidimos mejorar la legibilidad de la figura 26.7, usando el descriptor de acceso `set` del indexador en vez del método `Add` para agregar un par clave/valor en la línea 42, pero accidentalmente escribimos:

```
tabla[ clavePalabra ] = clavePalabra; // inicializa a 1
```

Esta instrucción crea una nueva entrada con una clave `string` y un valor `string`, en vez de un valor `int` de 1. Aunque la aplicación se compilará en forma correcta, es evidente que esto es incorrecto. Si una palabra aparece dos veces, la línea 38 tratará de realizar una conversión descendente sobre este valor `string`, para convertirlo en un `int`, con lo cual se producirá una excepción `InvalidCastException` en tiempo de ejecución. El error que aparece en tiempo de ejecución indicará que el problema está en la línea 38, en donde ocurrió la excepción, *no en la línea 42*. Esto hace que el error sea más difícil de encontrar y depurar, especialmente en aplicaciones de software extensas, en donde la excepción puede ocurrir en un archivo distinto; e incluso hasta en un ensamblado distinto.

En el capítulo 25 presentamos los genéricos. En las siguientes dos secciones, veremos cómo utilizar colecciones genéricas.

26.5 Colecciones genéricas

El espacio de nombres `System.Collections.Generic` en la FCL es una nueva adición para C# 2.0. Este espacio de nombres contiene clases genéricas que nos permiten crear colecciones de tipos específicos. Como vimos en la figura 26.2, muchas de las clases son simplemente versiones genéricas de colecciones no genéricas. Hay un par de clases que implementan nuevas estructuras de datos. En esta sección mostraremos las colecciones genéricas `SortedDictionary` y `LinkedList`.

26.5.1 La clase genérica `SortedDictionary`

Un *diccionario* es el término general para una colección de pares clave-valor. Una tabla hash es una manera de implementar un diccionario. El .NET Framework proporciona varias implementaciones de diccionarios, tanto genéricos como no genéricos (todos los cuales implementan a la interfaz `IDictionary` en la figura 26.1). La aplicación de la figura 26.8 es una modificación de la figura 26.7, que utiliza la clase genérica **`SortedDictionary`**. La clase genérica `SortedDictionary` no utiliza una tabla hash, sino que almacena sus pares clave-valor en un árbol binario de búsqueda. (En la sección 24.5 hablamos con detalle sobre los árboles binarios.) Como el nombre de la clase lo sugiere, las entradas en `SortedDictionary` se ordenan en el árbol por clave. Cuando la clave implementa a la interfaz genérica `IComparable`, `SortedDictionary` utiliza los resultados del método `CompareTo` de `IComparable` para ordenar las claves. Observe que, a pesar de estos detalles de implementación, usamos los mismos métodos `public`, propiedades e indexadores con las clases `Hashtable` y `SortedDictionary`, de la misma forma. De hecho, con la excepción de la sintaxis genérica específica, la figura 26.8 tiene una apariencia asombrosamente similar a la figura 26.7. Esto es lo bello de la programación orientada a objetos.

La línea 6 contiene una directiva `using` para el espacio de nombres `System.Collections.Generic`, el cual contiene la clase `SortedDictionary`. La clase genérica `SortedDictionary` recibe dos argumentos de tipo; el primero especifica el tipo de clave (por ejemplo, `string`) y el segundo especifica el tipo de valor (por decir, `int`). Lo único que hicimos fue sustituir la palabra `Hashtable` en la línea 13 y en las líneas 23-24 con `SortedDictionary< string, int >` para crear un diccionario de valores `int` con claves `string`. Ahora, el compilador puede comprobar y notificarnos si tratamos de almacenar un objeto del tipo incorrecto en el diccionario. Además, como el compilador ahora sabe que la estructura de datos contiene valores `int`, ya no hay necesidad de la conversión descendente en la línea 40. Esto permite que la línea 40 utilice la notación de incremento prefijo (`++`), mucho más precisa. Éstas son las únicas modificaciones que se realizaron en los métodos `Main` y `RecolectarPalabras`.

El método estático `MostrarDiccionario` (líneas 51-63) se modificó para ser completamente genérico. Recibe los parámetros de tipo `K` y `V`. Estos parámetros se utilizan en la línea 52 para indicar que `MostrarDiccionario` recibe un objeto `SortedDictionary` con claves de tipo `K` y valores de tipo `V`. Utilizamos el parámetro de tipo `K`

```

1  // Fig. 26.8: PruebaSortedDictionary.cs
2  // Aplicación que cuenta el número de ocurrencias de cada palabra en una cadena
3  // y las almacena en un diccionario ordenado genérico.
4  using System;
5  using System.Text.RegularExpressions;
6  using System.Collections.Generic;
7
8  public class PruebaSortedDictionary
9  {
10     public static void Main( string[] args )
11     {
12         // crea el diccionario ordenado, con base en la entrada del usuario
13         SortedDictionary< string, int > diccionario = RecolectarPalabras();
14
15         // muestra el contenido del diccionario ordenado
16         MostrarDiccionario( diccionario );
17     } // fin del método Main
18
19     // crea un diccionario ordenado a partir de la entrada del usuario
20     private static SortedDictionary< string, int > RecolectarPalabras()
21     {
22         // crea un nuevo diccionario ordenado
23         SortedDictionary< string, int > diccionario =
24             new SortedDictionary< string, int >();
25
26         Console.WriteLine( "Escriba una cadena: " ); // pide la entrada al usuario
27         string entrada = Console.ReadLine(); // obtiene la entrada
28
29         // divide el texto de entrada en símbolos (tokens)
30         string[] palabras = Regex.Split( entrada, @"\s+" );
31
32         // procesamiento de las palabras de entrada
33         foreach ( string palabra in palabras )
34         {
35             string clavePalabra = palabra.ToLower(); // obtiene palabra en minúsculas
36
37             // si el diccionario contiene la palabra
38             if ( diccionario.ContainsKey( clavePalabra ) )
39             {
40                 diccionario[ clavePalabra ]++;
41             } // fin de if
42             else
43             {
44                 // agrega una nueva palabra con una cuenta de 1 al diccionario
45                 diccionario.Add( clavePalabra, 1 );
46             } // fin de foreach
47
48             return diccionario;
49         } // fin del método RecolectarPalabras
50
51         // muestra el contenido del diccionario
52         private static void MostrarDiccionario< K, V >(
53             SortedDictionary< K, V > diccionario )
54         {
55             Console.WriteLine( "\nEl diccionario ordenado contiene:\n{0,-12}{1,-12}",
56                 "Clave:", "Valor:" );
57
58             // genera resultados para cada clave en el diccionario ordenado,

```

Figura 26.8 | Aplicación que cuenta el número de ocurrencias de cada palabra en un objeto string, y las almacena en un diccionario ordenado genérico. (Parte I de 2).


```

58      // mediante una iteración a través de la propiedad Keys con una instrucción
      foreach
59      foreach ( K clave in diccionario.Keys )
60          Console.WriteLine( "{0,-12}{1,-12}", clave, diccionario[ clave ] );
61
62      Console.WriteLine( "\ntamaño: {0}", diccionario.Count );
63  } // fin del método MostrarDiccionario
64 } // fin de la clase PruebaSortedDictionary

```

Escriba una cadena:

Somos pocos, somos felices y pocos, somos banda de hermanos

El diccionario ordenado contiene:

Clave:	Valor:
banda	1
de	1
felices	1
hermanos	1
pocos,	2
somos	3
y	1

tamaño: 7

Figura 26.8 | Aplicación que cuenta el número de ocurrencias de cada palabra en un objeto string, y las almacena en un diccionario ordenado genérico. (Parte 2 de 2).

otra vez en la línea 59 como el tipo de la clave de iteración. Este uso de los genéricos es un maravilloso ejemplo de reutilización de código. Si decidimos modificar la aplicación para que cuente el número de veces que aparece cada carácter en la cadena, el método `MostrarDiccionario` podría recibir un argumento de tipo `SortedDictionary < char, int >` sin necesidad de modificación.



Tip de rendimiento 26.7

Como la clase `SortedDictionary` mantiene sus elementos ordenados en un árbol binario, para obtener o insertar un par clave-valor se requiere un tiempo definido por $O(\log n)$, que es rápido en comparación con el proceso de realizar una búsqueda lineal y después la inserción.



Error común de programación 26.5

Al invocar el descriptor de acceso `get` de un indexador `SortedDictionary` con una clave que no exista en la colección, se produce una excepción `KeyNotFoundException`. Este comportamiento es distinto al del descriptor de acceso `get` del indexador de `Hashtable`, que devolvería `null`.

26.5.2 La clase genérica `LinkedList`

En el capítulo 24, empezamos nuestra discusión sobre las estructuras de datos con el concepto de una lista enlazada, y terminamos nuestra discusión con la clase genérica `LinkedList` del .NET Framework. La clase `LinkedList` es una lista doblemente enlazada; podemos navegar por la lista hacia delante y hacia atrás, con nodos de la clase genérica `LinkedListNode`. Cada nodo contiene la propiedad `Value`, y las propiedades de sólo lectura `Previous` y `Next`. El tipo de la propiedad `Value` coincide con el parámetro de tipo de `LinkedList`, ya que contiene los datos almacenados en el nodo. La propiedad `Previous` obtiene una referencia al nodo anterior en la lista enlazada (o `null` si el nodo es el primero de la lista). De manera similar, la propiedad `Next` obtiene una referencia a la subsiguiente referencia en la lista enlazada (o `null` si el nodo es el último de la lista). En la figura 26.9 mostramos unas cuantas manipulaciones de las listas enlazadas.

La directiva `using` en la línea 4 nos permite utilizar la clase `LinkedList` por su nombre no calificado. Las líneas 16-23 crean los objetos `LinkedList` `lista1` y `lista2` de objetos `string`, y las llenan con el contenido

```

1  // Fig. 26.9: PruebaLinkedList.cs
2  // Uso de objetos LinkedList.
3  using System;
4  using System.Collections.Generic;
5
6  public class PruebaLinkedList
7  {
8      private static readonly string[] colores = { "negro", "amarillo",
9          "verde", "azul", "violeta", "plata" };
10     private static readonly string[] colores2 = { "oro", "blanco",
11         "café", "azul", "gris" };
12
13     // establece y manipula objetos LinkedList
14     public static void Main( string[] args )
15     {
16         LinkedList< string > lista1 = new LinkedList< string >();
17
18         // agrega elementos a la primera lista enlazada
19         foreach ( string color in colores )
20             lista1.AddLast( color );
21
22         // agrega elementos a la segunda lista enlazada a través del constructor
23         LinkedList< string > lista2 = new LinkedList< string >( colores2 );
24
25         Concatenar( lista1, lista2 ); // concatena lista2 en lista1
26         ImprimirLista( lista1 ); // imprime los elementos de lista1
27
28         Console.WriteLine( "\nConvirtiendo las cadenas en lista1 a mayúsculas\n" );
29         CadenasAMayusculas( lista1 ); // convierte a cadena en mayúsculas
30         ImprimirLista( lista1 ); // imprime los elementos de lista1
31
32         Console.WriteLine( "\nEliminando las cadenas entre NEGRO y CAFÉ\n" );
33         EliminarElementosEntre( lista1, "NEGRO", "CAFÉ" );
34
35         ImprimirLista( lista1 ); // imprime los elementos de lista1
36         ImprimirListaInversa( lista1 ); // imprime la lista en orden inverso
37     } // fin del método Main
38
39     // imprime el contenido de la lista
40     private static void ImprimirLista< E >( LinkedList< E > lista )
41     {
42         Console.WriteLine( "Lista enlazada: " );
43
44         foreach ( E valor in lista )
45             Console.Write( "{0} ", valor );
46
47         Console.WriteLine();
48     } // fin del método ImprimirLista
49
50     // concatena la segunda lista al final de la primera lista
51     private static void Concatenar< E >( LinkedList< E > lista1,
52         LinkedList< E > lista2 )
53     {
54         // concatena las listas copiando los valores de los elementos
55         // en orden, desde la segunda lista hasta la primera lista
56         foreach ( E valor in lista2 )
57             lista1.AddLast( valor ); // agrega un nuevo nodo
58     } // fin del método Concatenar
59

```

Figura 26.9 | Uso de objetos LinkedList. (Parte I de 3).


```

60 // localiza los objetos string y los convierte a mayúsculas
61 private static void CadenasAMayusculas( LinkedList< string > lista )
62 {
63     // itera a través de la lista utilizando los nodos
64     LinkedListNode< string > nodoActual = lista.First;
65
66     while ( nodoActual != null )
67     {
68         string color = nodoActual.Value; // obtiene el valor en el nodo
69         nodoActual.Value = color.ToUpper(); // convierte a mayúsculas
70
71         nodoActual = nodoActual.Next; // obtiene el siguiente nodo
72     } // fin de while
73 } // fin del método CadenasAMayusculas
74
75 // elimina los elementos de la lista entre dos elementos dados
76 private static void EliminarElementosEntre< E >( LinkedList< E > lista,
77     E elementoInicial, E elementoFinal )
78 {
79     // obtiene los nodos que corresponden a los elementos inicial y final
80     LinkedListNode< E > nodoActual = lista.Find( elementoInicial );
81     LinkedListNode< E > nodoFinal = lista.Find( elementoFinal );
82
83     // elimina los elementos que van después del elemento inicial
84     // hasta encontrar el elemento final, o el final de la lista enlazada
85     while ( ( nodoActual.Next != null ) &&
86         ( nodoActual.Next != nodoFinal ) )
87     {
88         lista.Remove( nodoActual.Next ); // elimina el siguiente nodo
89     } // fin de while
90 } // fin del método EliminarElementosEntre
91
92 // imprime la lista al revés
93 private static void ImprimirListaInversa< E >( LinkedList< E > lista )
94 {
95     Console.WriteLine( "Lista invertida:" );
96
97     // itera a través de la lista, utilizando los nodos
98     LinkedListNode< E > nodoActual = lista.Last;
99
100    while ( nodoActual != null )
101    {
102        Console.Write( "{0} ", nodoActual.Value );
103        nodoActual = nodoActual.Previous; // obtiene el nodo anterior
104    } // fin de while
105
106    Console.WriteLine();
107 } // fin del método ImprimirListaInversa
108 } // fin de la clase PruebaLinkedList

```

Lista enlazada:
negro amarillo verde azul violeta plata oro blanco café azul gris

Convirtiendo las cadenas en lista1 a mayúsculas

Lista enlazada:
NEGRO AMARILLO VERDE AZUL VIOLETA PLATA ORO BLANCO CAFÉ AZUL GRIS

Eliminando las cadenas entre NEGRO y CAFÉ

Figura 26.9 | Uso de objetos LinkedList. (Parte 2 de 3).

```

Lista enlazada:
NEGRO CAFÉ AZUL GRIS
Lista invertida:
GRIS AZUL CAFÉ NEGRO

```

Figura 26.9 | Uso de objetos `LinkedList`. (Parte 3 de 3).

de los arreglos `colores` y `colores2`, respectivamente. Observe que `LinkedList` es una clase genérica que tiene un parámetro de tipo, para el cual especificamos el argumento de tipo `string` en este ejemplo (líneas 16 y 23). Mostraremos dos formas de llenar las listas. En las líneas 19-20, utilizamos la instrucción `foreach` y el método **AddLast** para llenar `lista1`. El método `AddLast` crea un nuevo objeto `LinkedListNode` (con el valor dado disponible a través de la propiedad `Value`), y adjunta este nodo al final de la lista. La línea 23 invoca el constructor que recibe un parámetro `IEnumerable< string >`. Todos los arreglos heredan en forma implícita de las interfaces genéricas `IList` e `IEnumerable` con el tipo del arreglo como el argumento de tipo, por lo que el arreglo `string colores2` implementa a `IEnumerable< string >`. El parámetro de tipo de este objeto genérico `IEnumerable` coincide con el parámetro de tipo del objeto `LinkedList` genérico. La llamada a este constructor copia el contenido del arreglo `colores2` a `lista2`.

La línea 25 llama al método genérico `Concatenar` (líneas 51-58) para adjuntar todos los elementos de `lista2` al final de `lista1`. La línea 26 llama al método `ImprimirLista` (líneas 40-48) para imprimir en pantalla el contenido de `lista1`. La línea 29 llama al método `CadenasAMayusculas` (líneas 61-73) para convertir cada elemento `string` a mayúsculas, y después la línea 30 llama a `ImprimirLista` de nuevo para mostrar los objetos `string` modificados. La línea 33 llama al método `EliminarElementosEntre` (líneas 76-90) para eliminar los elementos entre "NEGRO" y "CAFÉ", pero sin incluirlos. La línea 35 imprime en pantalla la lista otra vez, y después la línea 36 invoca al método `ImprimirListaInvertida` (líneas 93-107) para imprimir la lista en orden inverso.

El método genérico `Concatenar` (líneas 51-58) itera a través de `lista2` con una instrucción `foreach` y llama al método `AddLast` para adjuntar cada valor al final de `lista1`. El enumerador de la clase `LinkedList` itera a través de los valores de los nodos, y no de los nodos en sí, por lo que la variable de iteración tiene el tipo `E`. Observe que esto crea un nuevo nodo en `lista1` para cada nodo en `lista2`. Un objeto `LinkedListNode` no puede ser miembro de más de un objeto `LinkedList`. Cualquier intento por agregar un nodo de un objeto `LinkedList` a otro genera una excepción `InvalidOperationException`. Si desea que los mismos datos pertenezcan a más de un objeto `LinkedList`, debe crear una copia del nodo para cada lista.

El método genérico `ImprimirLista` (líneas 40-48) utiliza de manera similar una instrucción `foreach` para iterar a través de los valores en un objeto `LinkedList`, y los imprime en pantalla. El método `CadenasAMayusculas` (líneas 61-73) recibe una lista enlazada de objetos `string` y convierte cada valor `string` a mayúsculas. Este método sustituye los objetos `string` almacenados en la lista, por lo que no podemos utilizar un enumerador (a través de una instrucción `foreach`) como en los dos métodos anteriores. En vez de ello, obtenemos el primer objeto `LinkedListNode` a través de la propiedad `First` (línea 64) y utilizamos una instrucción `while` para iterar a través de la lista (líneas 66-72). Cada iteración de la instrucción `while` obtiene y actualiza el contenido de `nodoActual` mediante la propiedad `Value`, usando el método `string ToUpper` para crear una versión en mayúsculas del objeto `string` `color`. Al final de cada iteración, desplazamos el nodo actual al siguiente nodo en la lista, asignando `nodoActual` al nodo que se obtiene mediante su propia propiedad `Next` (línea 71). La propiedad `Next` del último nodo de la lista obtiene `null`, por lo que cuando la instrucción `while` itera más allá del final de la lista, el ciclo termina.

Observe que no tiene sentido declarar `CadenasAMayusculas` como un método genérico, ya que utiliza los métodos específicos de `string` de los valores en los nodos. Los métodos `ImprimirLista` (líneas 40-48) y `Concatenar` (líneas 51-58) no necesitan utilizar métodos específicos de `string`, por lo que pueden declararse con parámetros de tipo genéricos para promover la máxima reutilización de código.

El método genérico `EliminarElementosEntre` (líneas 76-90) elimina un rango de elementos entre dos nodos. Las líneas 80-81 obtienen los dos nodos "límite" del rango, usando el método **Find**. Este método realiza una búsqueda lineal en la lista, y devuelve el primer nodo que contenga un valor igual al argumento que se pasó. El método `Find` devuelve `null` si no encuentra el valor. Almacenamos el nodo que va antes del rango en la variable local `nodoActual`, y el nodo que va después del rango en `nodoFinal`.

La instrucción `while` en las líneas 85-89 elimina todos los elementos entre `nodoActual` y `nodoFinal`. En cada iteración del ciclo, eliminamos el nodo que va después de `nodoActual`, invocando al método **Remove** (línea 88).

Este método recibe un objeto `LinkedListNode`, extrae ese nodo del objeto `LinkedList` y corrige las referencias de los nodos circundantes. Después de la llamada a `Remove`, la propiedad `Next` de `nodoActual` obtiene ahora el nodo *que va después* del nodo que se acaba de eliminar, y la propiedad `Previous` de ese nodo ahora obtiene `nodoActual`. La instrucción `while` continúa iterando hasta que no quedan nodos entre `nodoActual` y `nodoFinal`, o hasta que `nodoActual` es el último nodo de la lista. (También existe una versión sobrecargada del método `Remove`, que realiza una búsqueda lineal para el valor especificado y elimina el primer nodo en la lista que lo contiene.)

El método `ImprimirListaInversa` (líneas 93-107) imprime la lista al revés, navegando por los nodos en forma manual. La línea 98 obtiene el último elemento de la lista a través de la propiedad **Last** y lo almacena en `nodoActual`. La instrucción `while` en las líneas 100-104 itera a través de la lista al revés, desplazando la referencia `nodoActual` al nodo anterior al final de cada iteración, y después se sale cuando avanzamos más allá del principio de la lista. Observe la similitud entre este código y las líneas 64-72, en donde se itera a través de la lista desde el principio hasta el final.

26.6 Colecciones sincronizadas

En el capítulo 15 hablamos sobre el subprocesamiento múltiple. La mayoría de las colecciones no genéricas carecen de sincronización de manera predeterminada, por lo que pueden operar con eficiencia cuando no se requiere el subprocesamiento múltiple. Sin embargo, como no están sincronizadas, el acceso concurrente a una colección por parte de varios subprocesos podría ocasionar errores. Para evitar problemas potenciales con el subprocesamiento, se utilizan contenedores de sincronización para muchas de las colecciones a las que varios subprocesos podrían acceder. Un objeto *contenedor* recibe llamadas a métodos, agrega la sincronización de subprocesos (para evitar el acceso concurrente a la colección) y transfiere las llamadas al objeto de la colección dentro del contenedor. La mayoría de las clases de colección no genéricas en el .NET Framework contienen el método **static Synchronized**, el cual devuelve un objeto contenedor sincronizado para el objeto especificado. Por ejemplo, el siguiente código crea un objeto `ArrayList` sincronizado:

```
ArrayList listaNoSegura = new ArrayList();
ArrayList listaSeguraSubproceso = ArrayList.Synchronized( listaNoSegura );
```

Las colecciones en el .NET Framework no proporcionan contenedores para un rendimiento seguro con varios subprocesos. Muchas de las colecciones genéricas son intrínsecamente seguras para los subprocesos en operaciones de lectura, pero no de escritura, consulte la documentación de esa clase en la referencia a las bibliotecas de clases del .NET Framework.

Recuerde que cuando se modifica una colección, cualquier enumerador devuelto previamente por el método `GetEnumerator` se invalida y lanza una excepción si se invocan sus métodos. Si utiliza un enumerador o una instrucción `foreach` en una aplicación con subprocesamiento múltiple, emplee la palabra clave `lock` para evitar que otros subprocesos usen la colección o una instrucción `try` para atrapar la excepción `InvalidOperationException`.

26.7 Conclusión

En este capítulo se presentaron las clases de colección del .NET Framework. Aprendió acerca de la jerarquía de interfaces que implementan muchas de las clases de colección. Vio cómo utilizar la clase `Array` para realizar manipulaciones de arreglos. Aprendió que los espacios de nombres `System.Collections` y `System.Collections.Generic` contienen muchas clases de colección no genéricas y genéricas, respectivamente. Presentamos las clases no genéricas `ArrayList`, `Stack` y `Hashtable`, así como las clases genéricas `SortedDictionary` y `LinkedList`.

También aprendió a utilizar enumeradores para recorrer estas estructuras de datos y obtener su contenido. Demostramos el uso de la instrucción `foreach` con muchas de las clases de la FCL, y explicamos que esto funciona mediante el uso de enumeradores “detrás de las cámaras” para recorrer las colecciones. Por último, hablamos sobre algunas de las cuestiones que se deben considerar al utilizar colecciones en aplicaciones con subprocesamiento múltiple.

Las páginas 1041 a 1156, que corresponden a los apéndices de este libro, se encuentran en el CD-ROM que acompaña al libro.



Tabla de precedencia de los operadores

A.1 Precedencia de los operadores

Los operadores se muestran en orden decreciente de precedencia, de arriba hacia abajo; cada nivel de precedencia se separa por una línea negra horizontal (figura A.1). La asociatividad de los operadores se muestra en la columna derecha.

Operador	Tipo	Asociatividad
.	acceso a miembro	de izquierda a derecha
()	llamada a método	
[]	acceso a elemento	
++	unario de postincremento	
--	unario de postdecremento	
new	creación de objetos	
typeof	obtener el objeto System.Type para un tipo	
sizeof	obtener el tamaño en bytes de un tipo	
checked	evaluación comprobada	
unchecked	evaluación no comprobada	
+	unario de suma	de derecha a izquierda
-	unario de resta	
!	negación lógica	
~	complemento a nivel de bits	
++	unario de preincremento	
--	unario de predecremento	
(type)	conversión	

Figura A.1 | Tabla de precedencia de los operadores. (Parte 1 de 2).

Operador	Tipo	Asociatividad
*	multiplicación	de izquierda a derecha
/	división	
%	residuo	
+	suma	de izquierda a derecha
-	resta	
>>	desplazamiento a la derecha	de izquierda a derecha
<<	desplazamiento a la izquierda	
<	menor que	de izquierda a derecha
>	mayor que	
<=	menor o igual que	
>=	mayor o igual que	
is	comparación de tipos	
as	conversión de tipos	
!=	no es igual a	de izquierda a derecha
==	es igual a	
&	AND lógico	de izquierda a derecha
^	XOR lógico	de izquierda a derecha
	OR lógico	de izquierda a derecha
&&	AND condicional	de izquierda a derecha
	OR condicional	de izquierda a derecha
??	asignación de null	de derecha a izquierda
?:	condicional	de derecha a izquierda
=	asignación	de derecha a izquierda
*=	asignación, multiplicación	
/=	asignación, división	
%=	asignación, residuo	
+=	asignación, suma	
-=	asignación, resta	
<<=	asignación, desplazamiento a la izquierda	
>>=	asignación, desplazamiento a la derecha	
&=	asignación, AND lógico	
^=	asignación, XOR lógico	
=	asignación, OR lógico	

Figura A.1 | Tabla de precedencia de los operadores. (Parte 2 de 2).

B

Sistemas numéricos

OBJETIVOS

En este apéndice aprenderá lo siguiente:

- Comprender los conceptos acerca de los sistemas numéricos como base, valor posicional y valor simbólico.
- Trabajar con los números representados en los sistemas numéricos binario, octal y hexadecimal.
- Abreviar los números binarios como octales o hexadecimales.
- Convertir los números octales y hexadecimales en binarios.
- Realizar conversiones hacia y desde números decimales y sus equivalentes en binario, octal y hexadecimal.
- Comprender el funcionamiento de la aritmética binaria y la manera en que se representan los números binarios negativos, utilizando la notación de complemento a dos.

He aquí sólo los números ratificados.

—William Shakespeare

La naturaleza tiene un cierto tipo de sistema de coordenadas aritméticas-geométricas, ya que cuenta con todo tipo de modelos. Lo que experimentamos de la naturaleza está en los modelos, y todos los modelos de la naturaleza son tan bellos. Se me ocurrió que el sistema de la naturaleza debe ser una verdadera belleza, porque en la química encontramos que las asociaciones se encuentran siempre en hermosos números enteros; no hay fracciones.

—Richard Buckminster Fuller

- B.1 Introducción
- B.2 Abreviatura de los números binarios como números octales y hexadecimales
- B.3 Conversión de números octales y hexadecimales a binarios
- B.4 Conversión de un número binario, octal o hexadecimal a decimal
- B.5 Conversión de un número decimal a binario, octal o hexadecimal
- B.6 Números binarios negativos: notación de complemento a dos

B.1 Introducción

En este apéndice presentaremos los sistemas numéricos clave que utilizan los programadores, especialmente cuando trabajan en proyectos de software que requieren de una estrecha interacción con el hardware a nivel de máquina. Entre los proyectos de este tipo están los sistemas operativos, el software de redes computacionales, los compiladores, sistemas de bases de datos y aplicaciones que requieren de un alto rendimiento.

Cuando escribimos un entero, como 227 o -63, en un programa, se asume que el número está en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10). En su interior, las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).

Como veremos, los números binarios tienden a ser mucho más extensos que sus equivalentes decimales. Los programadores que trabajan con lenguajes ensambladores y en lenguajes de alto nivel como C#, que les permiten llegar hasta el nivel de máquina, encuentran que es complicado trabajar con números binarios. Por eso existen otros dos sistemas numéricos, el octal (base 8) y el hexadecimal (base 16), que son populares debido a que permiten abreviar los números binarios de una manera conveniente.

En el sistema numérico octal, los dígitos utilizados son del 0 al 7. Debido a que tanto el sistema numérico binario como el octal tienen menos dígitos que el sistema numérico decimal, sus dígitos son los mismos que sus correspondientes en decimal.

El sistema numérico hexadecimal presenta un problema, ya que requiere de dieciséis dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15. Por lo tanto, en hexadecimal podemos tener números como el 876, que consisten solamente de dígitos similares a los decimales; números como 8A55F que consisten de dígitos y letras; y números como FFE que consisten solamente de letras. En ocasiones un número hexadecimal puede coincidir con una palabra común como FACE o FEED (en inglés); esto puede parecer extraño para los programadores acostumbrados a trabajar con números. Los dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal se sintetizan en las figuras B.1 y B.2.

Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un valor posicional distinto. Por ejemplo, en el número decimal 937 (el 9, el 3 y el 7 se conocen como valores simbólicos) decimos que el 7 se escribe en la posición de las unidades; el 3, en la de las decenas; y el 9, en la de las centenas. Observe que cada una de estas posiciones es una potencia de la base (10) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura B.3).

Para números decimales más extensos, las siguientes posiciones a la izquierda serían: de millares (10 a la tercera potencia), de decenas de millares (10 a la cuarta potencia), de centenas de millares (10 a la quinta potencia), de los millones (10 a la sexta potencia), de decenas de millones (10 a la séptima potencia), y así sucesivamente.

En el número binario 101 decimos que el 1 más a la derecha se escribe en la posición de los unos; el 0, en la posición de los dos; y el 1 de más a la izquierda, en la posición de los cuatros. Observe que cada una de estas posiciones es una potencia de la base (2) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura B.4). Por lo tanto, $101 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 0 + 1 = 5$.

Para números binarios más extensos, las siguientes posiciones a la izquierda serían la posición de los ochos (2 a la tercera potencia), la posición de los dieciséis (2 a la cuarta potencia), la de los treinta y dos (2 a la quinta potencia), la de los sesenta y cuatro (2 a la sexta potencia), y así sucesivamente.

En el número octal 425, decimos que el 5 se escribe en la posición de los unos; el 2, en la posición de los ochos; y el 4, en la posición de los sesenta y cuatro. Observe que cada una de estas posiciones es una potencia

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor de 10 en decimal)
			B (valor de 11 en decimal)
			C (valor de 12 en decimal)
			D (valor de 13 en decimal)
			E (valor de 14 en decimal)
			F (valor de 15 en decimal)

Figura B.1 | Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal.

Atributo	Binario	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito más bajo	0	0	0	0
Dígito más alto	1	7	9	F

Figura B.2 | Comparación de los sistemas binario, octal, decimal y hexadecimal.

Valores posicionales en el sistema numérico decimal			
Dígito decimal	9	3	7
Nombre de la posición	Centenas	Decenas	Unidades
Valor posicional	100	10	1
Valor posicional como potencia de la base (10)	10 ²	10 ¹	10 ⁰

Figura B.3 | Valores posicionales en el sistema numérico decimal.

de la base (8) y que empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura B.5).

Para números octales más extensos, las siguientes posiciones a la izquierda sería la posición de los quinientos doces (8 a la tercera potencia), la de los cuatro mil noventa y seis (8 a la cuarta potencia), la de los treinta y dos mil setecientos sesenta y ocho (8 a la quinta potencia), y así sucesivamente.

En el número hexadecimal 3DA, decimos que la A se escribe en la posición de los unos; la D, en la posición de los dieciséis; y el 3, en la posición de los doscientos cincuenta y seis. Observe que cada una de estas posiciones es una potencia de la base (16) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura B.6).

Valores posicionales en el sistema numérico binario			
Dígito binario	1	0	1
Nombre de la posición	Cuatros	Dos	Unos
Valor posicional	4	2	1
Valor posicional como potencia de la base (2)	2^2	2^1	2^0

Figura B.4 | Valores posicionales en el sistema numérico binario.

Valores posicionales en el sistema numérico octal			
Dígito octal	4	2	5
Nombre de la posición	Sesenta y cuatros	Ochos	Unos
Valor posicional	64	8	1
Valor posicional como potencia de la base (8)	8^2	8^1	8^0

Figura B.5 | Valores posicionales en el sistema numérico octal.

Valores posicionales en el sistema numérico hexadecimal			
Dígito hexadecimal	3	D	A
Nombre de la posición	Doscientos cincuenta y seis	Dieciséis	Unos
Valor posicional	256	16	1
Valor posicional como potencia de la base (16)	16^2	16^1	16^0

Figura B.6 | Valores posicionales en el sistema numérico hexadecimal.

Para números hexadecimales más extensos, las siguientes posiciones a la izquierda serían la posición de los cuatro mil noventa y seis (16 a la tercera potencia), la posición de los sesenta y cinco mil quinientos treinta y seis (16 a la cuarta potencia), y así sucesivamente.

B.2 Abreviatura de los números binarios como números octales y hexadecimales

En computación, el uso principal de los números octales y hexadecimales es para abreviar representaciones binarias demasiado extensas. La figura B.7 muestra que los números binarios extensos pueden expresarse más concisamente en sistemas numéricos con bases mayores que en el sistema numérico binario.

Una relación especialmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2). Considere el siguiente número binario de 12 dígitos y sus equivalentes en octal y hexadecimal. Vea si puede determinar cómo esta relación hace que sea conveniente abreviar los números binarios en octal o hexadecimal. La respuesta sigue después de los números.

Número binario	Equivalente en octal	Equivalente en hexadecimal
100011010001	4321	8D1

Número decimal	Representación binaria	Representación octal	Representación hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Figura B.7 | Equivalentes en decimal, binario, octal y hexadecimal.

Para convertir fácilmente el número binario en octal, sólo divida el número binario de 12 dígitos en grupos de tres bits consecutivos y escriba esos grupos por encima de los dígitos correspondientes del número octal, como se muestra a continuación:

100	011	010	001
4	3	2	1

Observe que el dígito octal que escribió debajo de cada grupo de tres bits corresponde precisamente al equivalente octal de ese número binario de 3 dígitos que se muestra en la figura B.7.

El mismo tipo de relación puede observarse al convertir números de binario a hexadecimal. Divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos y escriba esos grupos por encima de los dígitos correspondientes del número hexadecimal, como se muestra a continuación:

1000	1101	0001
8	D	1

Observe que el dígito hexadecimal que escribió debajo de cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de ese número binario de 4 dígitos que se muestra en la figura B.7.

B.3 Conversión de números octales y hexadecimales a binarios

En la sección anterior vimos cómo convertir números binarios a sus equivalentes en octal y hexadecimal, formando grupos de dígitos binarios y simplemente volviéndolos a escribir como sus valores equivalentes en dígitos octales o hexadecimales. Este proceso puede utilizarse en forma inversa para producir el equivalente en binario de un número octal o hexadecimal.

Por ejemplo, el número octal 653 se convierte en binario simplemente escribiendo el 6 como su equivalente binario de 3 dígitos 110, el 5 como su equivalente binario de 3 dígitos 101 y el 3 como su equivalente binario de 3 dígitos 011 para formar el número binario de 9 dígitos 110101011.

El número hexadecimal FAD5 se convierte en binario simplemente escribiendo la F como su equivalente binario de 4 dígitos 1111, la A como su equivalente binario de 4 dígitos 1010, la D como su equivalente binario de 4 dígitos 1101 y el 5 como su equivalente binario de 4 dígitos 0101, para formar el número binario de 16 dígitos 1111101011010101.

B.4 Conversión de un número binario, octal o hexadecimal a decimal

Como estamos acostumbrados a trabajar con el sistema decimal, a menudo es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que “realmente” vale el número. Nuestros diagramas en la sección B.1 expresan los valores posicionales en decimal. Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor posicional y sume estos productos. Por ejemplo, el número binario 110101 se convierte en el número 53 decimal, como se muestra en la figura B.8.

Para convertir el número 7614 octal en el número 3980 decimal utilizamos la misma técnica, esta vez utilizando los valores posicionales apropiados para el sistema octal, como se muestra en la figura B.9.

Para convertir el número AD3B hexadecimal en el número 44347 decimal utilizamos la misma técnica, esta vez empleando los valores posicionales apropiados para el sistema hexadecimal, como se muestra en la figura B.10.

B.5 Conversión de un número decimal a binario, octal o hexadecimal

Las conversiones de la última sección siguen naturalmente las convenciones de la notación posicional. Las conversiones de decimal a binario, octal o hexadecimal también siguen estas convenciones.

Conversión de un número binario en decimal						
Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	0	1	0	1
Productos:	$1 \times 32 = 32$	$1 \times 16 = 16$	$0 \times 8 = 0$	$1 \times 4 = 4$	$0 \times 2 = 0$	$1 \times 1 = 1$
Suma:	$= 32 + 16 + 0 + 4 + 0 + 1 = 53$					

Figura B.8 | Conversión de un número binario en decimal.

Conversión de un número octal en decimal				
Valores posicionales:	512	64	8	1
Valores simbólicos:	7	6	1	4
Productos:	$7 \times 512 = 3584$	$6 \times 64 = 384$	$1 \times 8 = 8$	$4 \times 1 = 4$
Suma:	$= 3584 + 384 + 8 + 4 = 3980$			

Figura B.9 | Conversión de un número octal en decimal.

Conversión de un número hexadecimal en decimal				
Valores posicionales:	4096	256	16	1
Valores simbólicos:	A	D	3	B
Productos:	$A \times 4096 = 40960$	$D \times 256 = 3328$	$3 \times 16 = 48$	$B \times 1 = 11$
Suma:	$= 40960 + 3328 + 48 + 11 = 44347$			

Figura B.10 | Conversión de un número hexadecimal en decimal.

Suponga que queremos convertir el número 57 decimal en binario. Empezamos escribiendo los valores posicionales de las columnas de derecha a izquierda, hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales:	64	32	16	8	4	2	1

Luego descartamos la columna con el valor posicional de 64, dejando:

Valores posicionales:	32	16	8	4	2	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 57 entre 32 y observamos que hay un 32 en 57, con un residuo de 25, por lo que escribimos 1 en la columna de los 32. Dividimos 25 entre 16 y observamos que hay un 16 en 25, con un residuo de 9, por lo que escribimos 1 en la columna de los 16. Dividimos 9 entre 8 y observamos que hay un 8 en 9 con un residuo de 1. Las siguientes dos columnas producen el cociente de cero cuando se divide 1 entre sus valores posicionales, por lo que escribimos 0 en las columnas de los 4 y de los 2. Por último, 1 entre 1 es 1, por lo que escribimos 1 en la columna de los 1. Esto nos da:

Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	1	0	0	1

y, por lo tanto, el 57 decimal es equivalente al 111001 binario.

Para convertir el número decimal 103 en octal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales:	512	64	8	1

Luego descartamos la columna con el valor posicional de 512, lo que nos da:

Valores posicionales:	64	8	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 103 entre 64 y observamos que hay un 64 en 103 con un residuo de 39, por lo que escribimos 1 en la columna de los 64. Dividimos 39 entre 8 y observamos que el 8 cabe cuatro veces en 39 con un residuo de 7, por lo que escribimos 4 en la columna de los 8. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto nos da:

Valores posicionales:	64	8	1
Valores simbólicos:	1	4	7

y, por lo tanto, el 103 decimal es equivalente al 147 octal.

Para convertir el número decimal 375 en hexadecimal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por consecuencia, primero escribimos:

Valores posicionales:	4096	256	16	1

Luego descartamos la columna con el valor posicional de 4096, lo que nos da:

Valores posicionales:	256	16	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 375 entre 256 y observamos que 256 cabe una vez en 375 con un residuo de 119, por lo que escribimos 1 en la columna de los 256. Dividimos 119 entre 16 y observamos que el 16 cabe siete veces en 119 con un residuo de 7, por lo que escribimos 7 en la columna de los 16. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto produce:

Valores posicionales:	256	16	1
Valores simbólicos:	1	7	7

y, por lo tanto, el 375 decimal es equivalente al 177 hexadecimal.

B.6 Números binarios negativos: notación de complemento a dos

La discusión en este apéndice se ha enfocado hasta ahora en números positivos. En esta sección explicaremos cómo las computadoras representan números negativos mediante el uso de la *notación de complementos a dos*. Primero explicaremos cómo se forma el complemento a dos de un número binario y después mostraremos por qué representa el valor negativo de dicho número binario.

Considere una máquina con enteros de 32 bits. Suponga que se ejecuta la siguiente instrucción:

```
int valor = 13;
```

La representación en 32 bits de `valor` es:

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de `valor`, primero formamos *su complemento a uno* aplicando el operador de complemento a nivel de bits de C# (~):

```
complementoAUnoDeValor = ~valor;
```

Internamente, `~valor` es ahora `valor` con cada uno de sus bits invertidos; los unos se convierten en ceros y los ceros en unos, como se muestra a continuación:

```
valor:
00000000 00000000 00000000 00001101
~valor (es decir, el complemento a uno de valor):
11111111 11111111 11111111 11110010
```

Para formar el complemento a dos de `valor`, simplemente sumamos uno al complemento a uno de `valor`. Por lo tanto:

```
El complemento a dos de valor es:
11111111 11111111 11111111 11110011
```

Ahora, si esto de hecho es igual a -13 , deberíamos poder sumarlo al 13 binario y obtener como resultado 0. Comprobemos esto:

```
00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----
00000000 00000000 00000000 00000000
```

El bit de acarreo que sale de la columna que está más a la izquierda se descarta y evidentemente obtenemos cero como resultado. Si sumamos el complemento a uno de un número a ese mismo número, todos los dígitos del resultado serían iguales a 1. La clave para obtener un resultado en el que todos los dígitos sean cero es que el complemento a dos es 1 más que el complemento a 1. La suma de 1 hace que el resultado de cada columna sea 0 y se acarrea un 1. El acarreo sigue desplazándose hacia la izquierda hasta que se descarta en el bit que está más a la izquierda, con lo que todos los dígitos del número resultante son iguales a cero.

En realidad, las computadoras realizan una suma como:

```
x = a - valor;
```

mediante la suma del complemento a dos de `valor` con `a`, como se muestra a continuación:

```
x = a + (~valor + 1);
```

Suponga que `a` es 27 y que `valor` es 13 como en el ejemplo anterior. Si el complemento a dos de `valor` es en realidad el negativo de éste, entonces al sumar el complemento de dos de `valor` con `a` se produciría el resultado de 14. Comprobemos esto:

```
a (es decir, 27)    00000000 00000000 00000000 00011011
+ (~valor + 1) +11111111 11111111 11111111 11110011
-----
00000000 00000000 00000000 00001110
```

lo que ciertamente da como resultado 14.



Uso del depurador de Visual Studio® 2005

OBJETIVOS

En este apéndice aprenderá lo siguiente:

- Utilizar el depurador para localizar y corregir errores lógicos en un programa.
- Utilizar puntos de interrupción para suspender la ejecución de un programa y poder examinar los valores de las variables.
- Establecer, deshabilitar y quitar puntos de interrupción.
- Usar el comando **Continuar** para continuar la ejecución desde un punto de interrupción.
- Usar la ventana **Variables locales** para ver y modificar los valores de las variables.
- Usar la ventana **Inspección** para evaluar expresiones.
- Usar los comandos **Paso a paso por instrucciones**, **Paso a paso por procedimientos** y **Paso a paso para salir**, para ejecutar un programa línea por línea.
- Usar las nuevas características Editar y Continuar de Visual Studio 2005, junto con la depuración Sólo mi código (Just My Code™).

Estamos contruidos para cometer errores, codificados por error.

—Lewis Thomas

Lo que anticipamos raras veces pasa; lo que menos esperamos es lo que generalmente ocurre.

—Benjamin Disraeli

Una cosa es mostrar a un hombre que está equivocado, y otra es ponerlo en posesión de la verdad.

—John Locke

Puede correr, pero no puede ocultarse.

—Joe Louis

Por lo tanto, yo también atraparé la mosca.

—William Shakespeare

- C.1 Introducción
- C.2 Los puntos de interrupción y el comando Continuar
- C.3 Las ventanas Variables locales e Inspección
- C.4 Control de la ejecución mediante los comandos Paso a paso por instrucciones, Paso a paso por procedimientos, Paso a paso para salir y Continuar
- C.5 Otras características
 - C.5.1 Editar y Continuar
 - C.5.2 Ayudante de excepciones
 - C.5.3 Depuración Sólo mi código (Just My Code™)
 - C.5.4 Otras nuevas características del depurador
- C.6 Conclusión

C.1 Introducción

En el capítulo 3 vimos que hay dos tipos de errores: los de compilación y los errores lógicos; y aprendimos a eliminar del código los errores de compilación. Los errores lógicos, también conocidos como *bugs*, no evitan que un programa se compile con éxito, pero pueden provocar resultados erróneos, o hacer que el programa termine antes de tiempo a la hora de ejecutarse. La mayoría de los distribuidores de compiladores como Microsoft, proporcionan una herramienta conocida como *depurador*, el cual le permite supervisar la ejecución de sus programas para localizar y eliminar los errores lógicos. Un programa debe compilarse con éxito para poder utilizarlo en el depurador; éste nos ayuda a analizar un programa mientras se ejecuta. El depurador nos permite suspender la ejecución de un programa, examinar y establecer los valores de las variables, y mucho más. En este apéndice presentaremos el depurador de Visual Studio, varias de sus herramientas de depuración y las nuevas características que se agregaron en Visual Studio 2005.

C.2 Los puntos de interrupción y el comando Continuar

Empezaremos por investigar los *puntos de interrupción*, que son marcadores que pueden establecerse en cualquier línea de código ejecutable. Cuando un programa en ejecución llega a un punto de interrupción, se pausa la ejecución y podemos examinar los valores de las variables, lo cual nos ayuda a determinar si existen o no errores lógicos. Por ejemplo, podemos examinar el valor de una variable que almacena el resultado de un cálculo para determinar si éste se realizó en forma correcta. También podemos examinar el valor de una expresión.

Para ilustrar las características del depurador, utilizaremos el programa en las figuras C.1 y C.2, el cual crea y manipula un objeto de la clase Cuenta (figura C.1). Este ejemplo es similar al que vimos en el capítulo 4 (figuras 4.15 y 4.16). Por lo tanto, no utiliza las características que presentamos en capítulos posteriores, como el operador `+=` y la instrucción `if...else`. La ejecución empieza en `Main` (líneas 8-29 de la figura C.2). La línea 10 crea un objeto Cuenta con un saldo inicial de \$50.00. El constructor de Cuenta (líneas 10-13 de la figura C.1) acepta un argumento, el cual especifica el `Saldo` inicial de la Cuenta. Las líneas 13-14 de la figura C.2 muestran en pantalla el saldo inicial de la cuenta, usando la propiedad `Saldo` de Cuenta. La línea 16 declara e inicializa la variable local `montoDeposito`. Las líneas 19-20 piden al usuario que introduzca el `montoDeposito`. La línea 23 suma el depósito al `saldo` de la Cuenta, usando su método `Acredita`. Por último, las líneas 26-27 muestran el nuevo `saldo`.

```

1  // Fig. C.01: Cuenta.cs
2  // La clase Cuenta con un constructor para
3  // inicializar la variable de instancia saldo.
4
5  public class Cuenta
6  {
7      private decimal saldo; // variable de instancia que almacena el saldo

```

Figura C.1 | La clase Cuenta con un constructor para inicializar la variable de instancia `saldo`. (Parte 1 de 2).

```

8
9 // constructor
10 public Cuenta( decimal saldoInicial )
11 {
12     Saldo = saldoInicial; // establece el saldo usando la propiedad Saldo
13 } // fin del constructor de Cuenta
14
15 // acredita (suma) un monto a la cuenta
16 public void Acredita( decimal monto )
17 {
18     Saldo = Saldo + monto; // suma el monto al saldo
19 } // fin del método Acredita
20
21 // una propiedad para obtener y establecer el saldo de la cuenta
22 public decimal Saldo
23 {
24     get
25     {
26         return saldo;
27     } // fin de get
28     set
29     {
30         // valida si el valor es mayor que 0;
31         // si no lo es, saldo se establece al valor predeterminado de 0
32         if ( value > 0 )
33             saldo = value;
34
35         if ( value <= 0 )
36             saldo = 0;
37     } // fin de set
38 } // fin de la propiedad Saldo
39 } // fin de la clase Cuenta

```

Figura C.1 | La clase Cuenta con un constructor para inicializar la variable de instancia `saldo`. (Parte 2 de 2).

```

1 // Fig. C.02: PruebaCuenta.cs
2 // Crea y manipula un objeto Cuenta.
3 using System;
4
5 public class PruebaCuenta
6 {
7     // el método Main empieza la ejecución de la aplicación en C#
8     public static void Main( string[] args )
9     {
10         Cuenta cuenta1 = new Cuenta( 50.00M ); // crea un objeto Cuenta
11
12         // muestra el saldo inicial de cada objeto, usando la propiedad Saldo
13         Console.Write( "Saldo de cuenta1: {0:C}\n",
14             cuenta1.Saldo ); // muestra el Saldo
15
16         decimal montoDeposito; // deposita el monto introducido por el usuario
17
18         // pide y obtiene la entrada del usuario
19         Console.Write( "Escriba el monto a depositar para cuenta1: " );
20         montoDeposito = Convert.ToDecimal( Console.ReadLine() );
21         Console.Write( "sumando {0:C} al saldo de la cuenta1\n\n",
22             montoDeposito );

```

Figura C.2 | Creación y manipulación de un objeto Cuenta. (Parte 1 de 2).


```

23     cuenta1.Accredita( montoDeposito ); // suma al saldo de cuenta1
24
25     // muestra el saldo
26     Console.Write( "cuenta1 saldo: {0:C}\n",
27         cuenta1.Saldo );
28     Console.WriteLine();
29 } // fin de Main
30 } // fin de la clase PruebaCuenta

```

Saldo de cuenta1: \$50.00

Escriba el monto a depositar para cuenta1: **49.99**
sumando \$49.99 al saldo de la cuenta1

cuenta1 saldo: \$99.99

Figura C.2 | Creación y manipulación de un objeto Cuenta. (Parte 2 de 2).

En los siguientes pasos, utilizará puntos de interrupción y varios comandos del depurador para examinar el valor de la variable `montoDeposito` (declarada en la figura C.2) mientras el programa se ejecuta.

1. **Insertar puntos de interrupción en Visual Studio.** Primero, asegúrese que esté abierto el archivo `PruebaCuenta.cs` en el editor de código del IDE. Para insertar un punto de interrupción, haga clic con el botón izquierdo del ratón dentro de la *barra indicadora de margen* (el margen gris a la izquierda de la ventana de código en la figura C.3) que está enseguida de la línea de código en la que desea interrumpir la ejecución, o haga clic con el botón derecho del ratón en esa línea de código y seleccione la opción **Punto de interrupción > Insertar punto de interrupción**. Puede establecer todos los puntos de interrupción que desee. Establezca puntos de interrupción en las líneas 19, 23 y 28 de su código. Un *círculo relleno* aparece en la barra indicadora de margen en donde hizo clic y se resalta toda la instrucción de código completa, indicando que se han establecido puntos de interrupción (figura C.3). Cuando el programa se ejecuta, el depurador suspende la ejecución en cualquier línea que contenga un punto de interrupción. Después, el programa entra en *modo de interrupción*. Pueden establecerse puntos de interrupción antes de ejecutar un programa, en modo de interrupción y durante la ejecución.

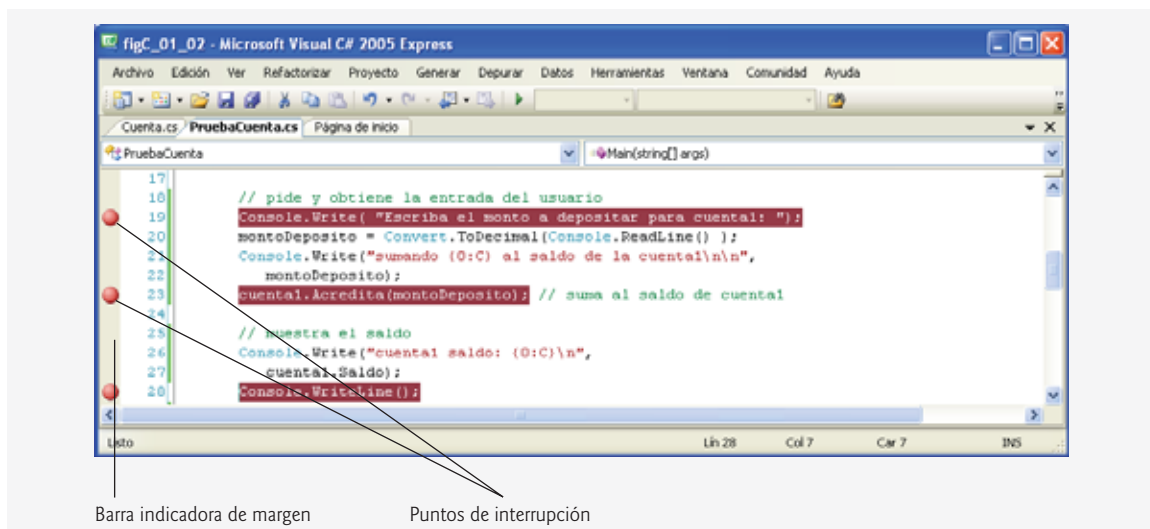


Figura C.3 | Establecimiento de puntos de interrupción.

2. **Empezar el proceso de depuración.** Después de establecer puntos de interrupción en el editor de código, seleccione **Generar > Generar solución** para compilar el programa y después seleccione **Depurar > Iniciar depuración** (u oprima *F5*) para empezar el proceso de depuración. Mientras se depura una aplicación de consola, aparece la ventana Símbolo del sistema (figura C.4), lo cual nos permite interactuar con el programa (entradas y salidas).
3. **Examinar la ejecución del programa.** La ejecución del programa se detiene en el primer punto de interrupción (línea 19), y el IDE se convierte en la ventana activa (figura C.5). La *flecha amarilla* a la izquierda de la línea 19 indica que esta línea contiene la siguiente instrucción a ejecutar. El IDE también resalta la línea.
4. **Uso del comando Continue para continuar la ejecución.** Para continuar la ejecución, seleccione **Depurar > Continuar** (u oprima *F5*). El comando **Continuar** ejecutará las instrucciones a partir del punto actual en el programa, hasta el siguiente punto de interrupción o el final de *Main*, lo que ocurra primero. El programa continúa su ejecución y se detiene para recibir la entrada en la línea 20. Escriba 49.99 en la ventana Símbolo del sistema como el monto a depositar. Cuando oprima *Intro*, el programa se ejecutará hasta detenerse en el siguiente punto de interrupción (línea 23). Observe que si coloca el puntero del ratón encima del nombre de la variable *montoDeposito*, su valor se muestra en un *cuadro de Información rápida* (figura C.6). Como veremos más adelante, este cuadro puede ayudarle a detectar errores lógicos en sus programas.
5. **Continuar la ejecución del programa.** Use el comando **Depurar > Continuar** para ejecutar la línea 23. El programa deberá mostrar el resultado de su cálculo (figura C.7).
6. **Deshabilitar un punto de interrupción.** Para *deshabilitar un punto de interrupción*, haga clic con el botón derecho en una línea de código en la que se haya establecido el punto de interrupción, y seleccione **Punto de interrupción > Deshabilitar punto de interrupción**. También puede hacer clic con el botón derecho del ratón en el mismo punto de interrupción y seleccionar **Deshabilitar punto de interrupción**. El punto de interrupción deshabilitado se indica mediante un círculo sin relleno (figura C.8); para

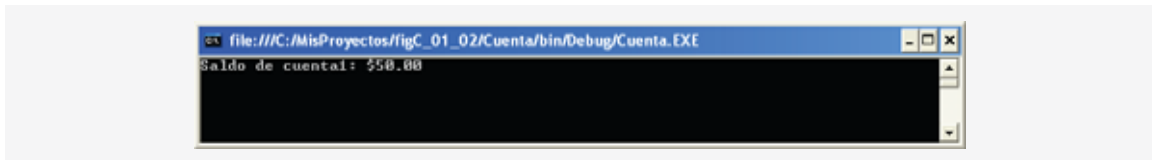


Figura C.4 | Ejecución del programa Cuenta.

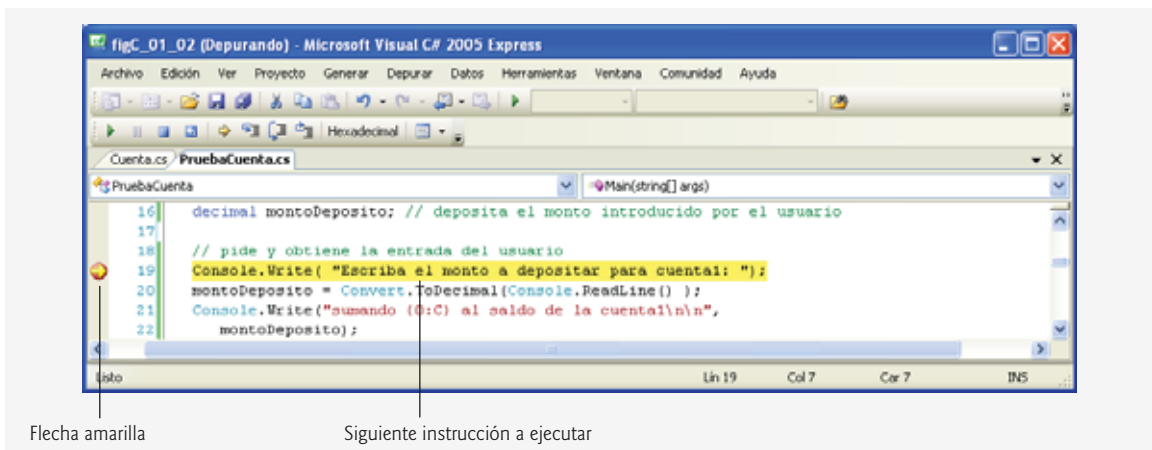


Figura C.5 | La ejecución del programa suspendida en el primer punto de interrupción.

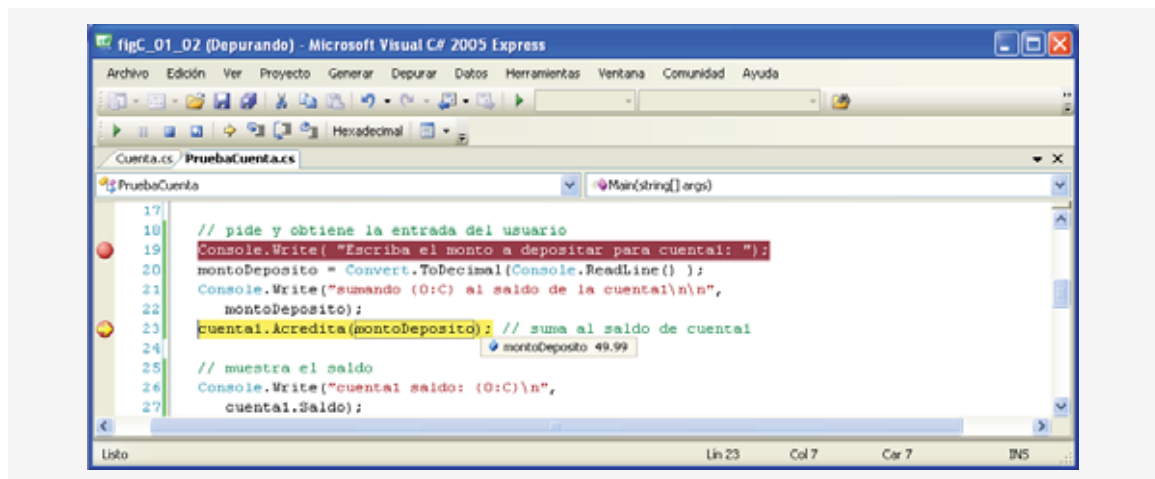


Figura C.6 | El cuadro de Información rápida muestra el valor de la variable `montoDeposito`.

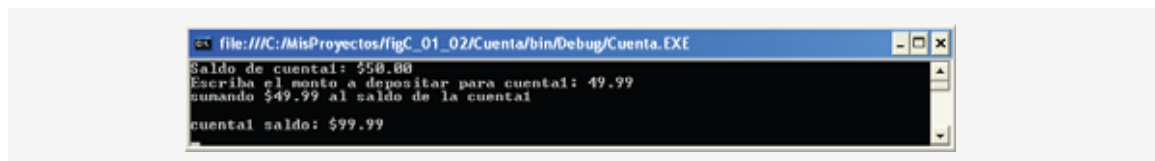


Figura C.7 | Salida del programa.

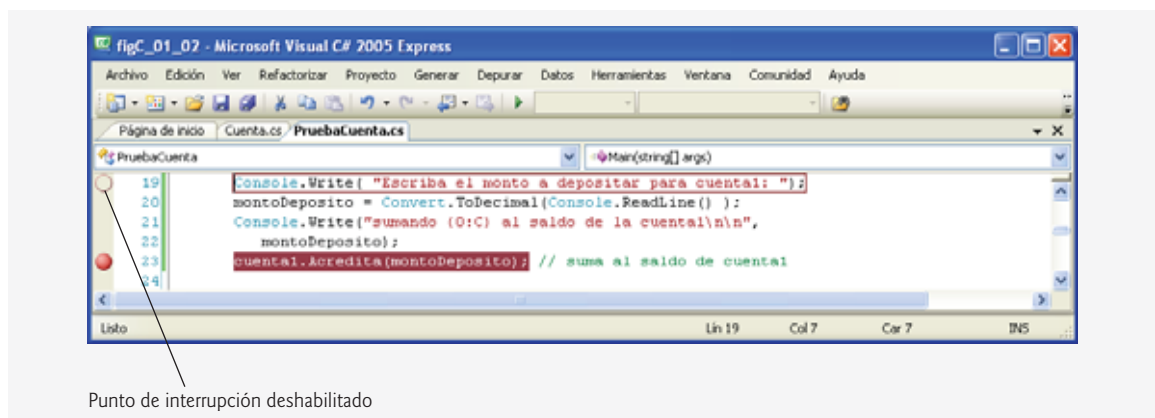


Figura C.8 | Punto de interrupción deshabilitado.

rehabilitar el punto de interrupción, haga clic con el ratón dentro del círculo sin relleno o haga clic con el botón derecho del ratón en la línea marcada por el círculo sin relleno (o el círculo en sí) y seleccione **Punto de interrupción > Habilitar punto de interrupción**.

7. **Eliminar un punto de interrupción.** Para eliminar un punto de interrupción que ya no necesite, haga clic con el botón derecho del ratón en la línea de código en la que se encuentra el punto de interrupción y seleccione **Punto de interrupción > Eliminar punto de interrupción**. También puede eliminar un punto de interrupción haciendo clic en el círculo que se encuentra en la barra indicadora de margen.
8. **Terminar la ejecución del programa.** Seleccione **Depurar > Continuar** para ejecutar el programa hasta que termine.

C.3 Las ventanas Variables locales e Inspección

En la sección anterior aprendió que la característica *Información rápida* nos permite examinar el valor de una variable. En esta sección aprenderá a utilizar la **ventana Variables locales** para asignar nuevos valores a las variables mientras su programa se ejecuta. También utilizará la **ventana Inspección** para examinar los valores de las expresiones.

1. **Insertar puntos de interrupción.** Establezca un punto de interrupción en la línea 23 (figura C.9) del código fuente, haciendo clic con el botón izquierdo del ratón en la barra indicadora de margen a la izquierda de la línea 23. Use la misma técnica para establecer puntos de interrupción en las líneas 26 y 28 también.
2. **Iniciar la depuración.** Seleccione **Depurar > Iniciar depuración**. Escriba 49.99 en el indicador **Escriba el monto a depositar para cuenta 1:** (figura C.10) y oprima **Intro**, para que el programa lea el valor que acaba de introducir. El programa se ejecutará hasta el punto de interrupción en la línea 23.
3. **Suspender la ejecución del programa.** Cuando el programa llega a la línea 23, Visual Studio suspende la ejecución y coloca al programa en modo de interrupción (figura C.11). En este punto, la instrucción en la línea 20 (figura C.2) recibe el `montoDeposito` que usted escribió (49.99), la instrucción en las líneas 21-22 imprime en pantalla un mensaje indicando que el programa está sumando ese monto al saldo de `cuenta1` y la instrucción en la línea 23 es la siguiente a ejecutar.
4. **Examinar los datos.** Una vez que el programa entra al modo de interrupción, puede explorar los valores de sus variables locales mediante la ventana **Variables locales** del depurador. Para ver esta ventana, seleccione **Depurar > Ventanas > Variables locales**. Haga clic en el cuadro con el signo positivo a la izquierda de `cuenta1`, en la columna **Nombre** de la ventana **Variables locales** (figura C.12). Esto le permite ver cada uno de los valores de la variable de instancia `cuenta1` en forma individual, incluyendo el valor de `saldo` (50). Observe que la ventana **Variables locales** muestra las propiedades de una clase como datos, razón por la cual vemos la propiedad `Saldo` y la variable de instancia `saldo` en esta ventana. Además, también se muestran el valor actual de la variable local `montoDeposito` (49.99) y el parámetro `args` de `Main`.
5. **Evaluar expresiones aritméticas y booleanas.** Podemos evaluar expresiones aritméticas y booleanas usando la ventana **Inspección**. Seleccione **Depurar > Ventanas > Inspección** para mostrar esta ventana (figura C.13). En la primera fila de la columna **Nombre** (que al principio debe estar en blanco) escriba `(montoDeposito + 10) * 5` y oprima **Intro**. A continuación aparecerá el valor 299.95 (figura C.13). En la siguiente fila de la

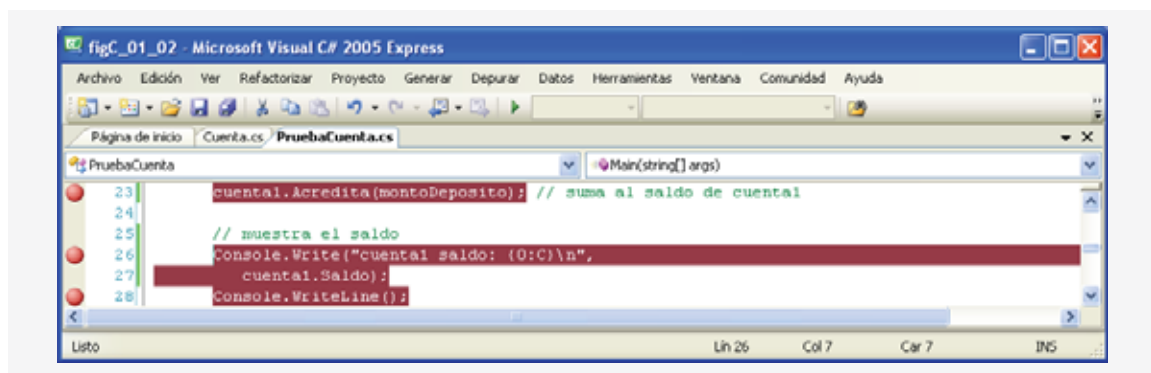


Figura C.9 | Establecer puntos de interrupción en las líneas 23 y 26.

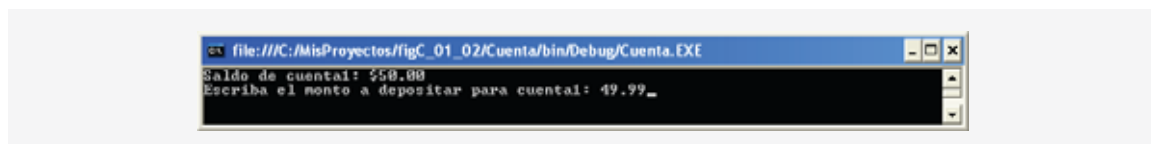


Figura C.10 | Escribir el monto a depositar antes de llegar al punto de interrupción.

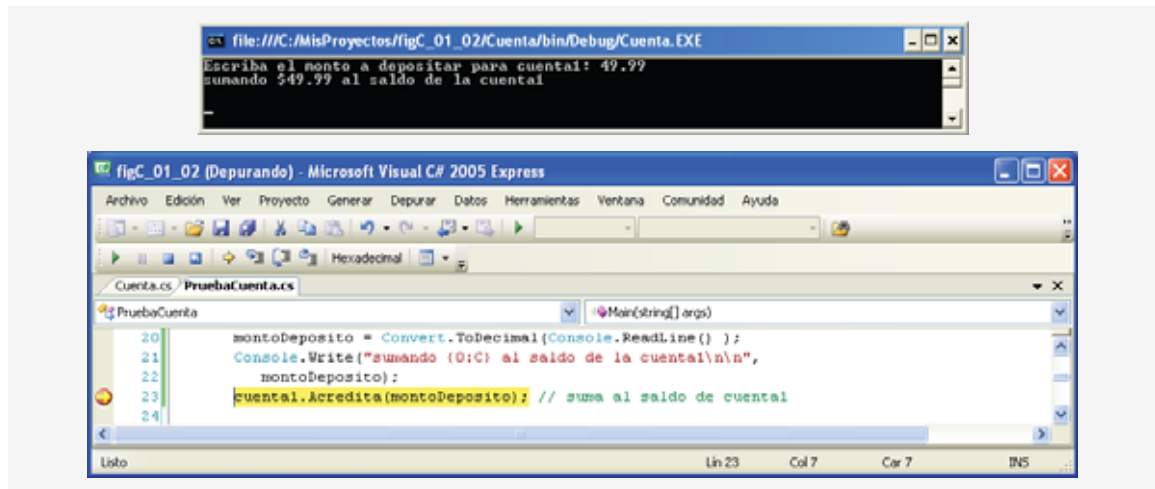


Figura C.11 | La ejecución del programa se detiene cuando el depurador llega al punto de interrupción en la línea 23.

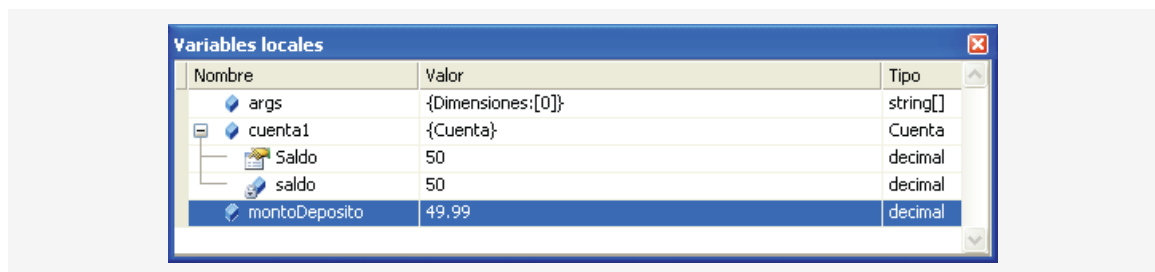


Figura C.12 | Análisis de la variable `montoDeposito`.

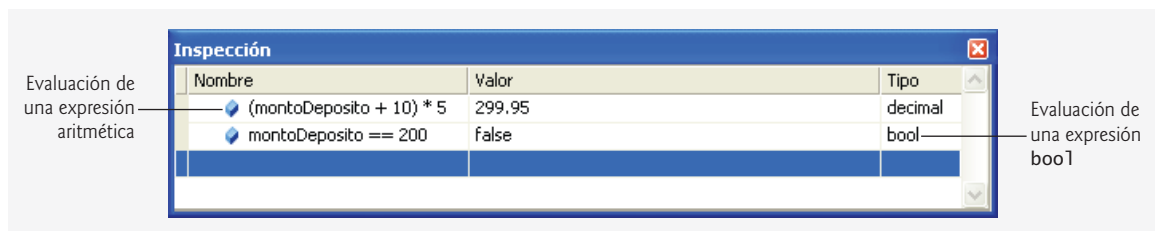


Figura C.13 | Análisis de los valores de las expresiones.

columna **Nombre** en la ventana **Inspección**, escriba `montoDeposito == 200` y oprima **Intro**. Esta expresión determina si el valor que contiene `montoDeposito` es 200. Las expresiones que contienen el símbolo `==` son expresiones `bool`. El valor devuelto es `false` (figura C.13), ya que `montoDeposito` no contiene en ese momento el valor 200.

6. **Continuar la ejecución.** Seleccione **Depurar > Continuar** para continuar con la ejecución. La línea 23 se ejecuta, acreditando a la cuenta el monto de depósito, y el programa regresa al modo de interrupción en la línea 26. Seleccione **Depurar > Ventanas > Variables locales**. Ahora se mostrarán la variable de instancia `saldo` y la propiedad `Saldo` con sus valores actualizados (figura C.14).
7. **Modificar los valores.** Con base en el valor introducido por el usuario (49.99), el saldo de la cuenta producido por el programa debe ser \$99.99. No obstante, puede utilizar la ventana **Variables locales** para modificar los valores de las variables durante la ejecución del programa. Esto puede ser valioso para

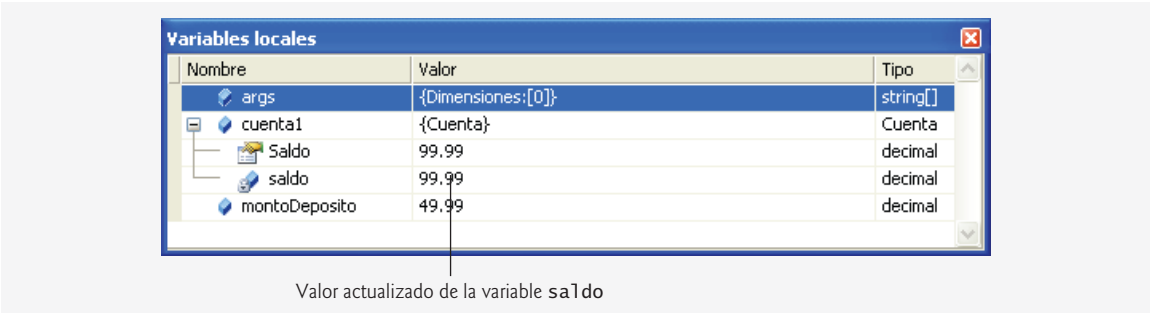


Figura C.14 | Mostrar el valor de las variables locales.

experimentar con distintos valores y para localizar errores lógicos en los programas. En la ventana **Variables locales**, haga clic en el campo **Valor** en la fila **saldo** para seleccionar el valor 99.99. Escriba 66.99 y oprima **Intro**. El depurador cambia el valor de **saldo** (y la propiedad **Saldo** también) y después muestra su nuevo valor en rojo (figura C.15). Ahora seleccione **Depurar > Continuar** para ejecutar las líneas 26-27. Observe que el nuevo valor de **saldo** se muestra en la ventana **Símbolo del sistema**.

- 8. *Detener la sesión de depuración.* Seleccione **Depurar > Detener depuración**. Elimine todos los puntos de interrupción.

C.4 Control de la ejecución mediante los comandos Paso a paso por instrucciones, Paso a paso por procedimientos, Paso a paso para salir y Continuar

Algunas veces es necesario ejecutar un programa línea por línea, para buscar y corregir errores lógicos. El avance paso a paso a lo largo de una parte del programa puede ayudarnos a verificar que el código de un método se ejecute en forma correcta. Los comandos que usted aprenderá en esta sección le permitirán ejecutar un método línea por línea, ejecutar todas las instrucciones de un método o sólo las instrucciones restantes del mismo (si ya ha ejecutado algunas instrucciones dentro del método).

- 1. *Insertar un punto de interrupción.* Establezca un punto de interrupción en la línea 23, haciendo clic con el botón izquierdo en la barra indicadora de margen (figura C.16).
- 2. *Iniciar el depurador.* Seleccione **Depurar > Iniciar depuración**. Introduzca el valor 49.99 en el indicador **Escriba el monto a depositar para cuenta1:**. La ejecución del programa se detendrá cuando éste llegue al punto de interrupción en la línea 23.
- 3. *Uso del comando Paso a paso por instrucciones.* El **comando Paso a paso por instrucciones** ejecuta la siguiente instrucción en el programa (la línea resaltada de la figura C.17) y se detiene de inmediato. Si la instrucción a ejecutar es la llamada a un método, el control se transfiere al método que se llamó.

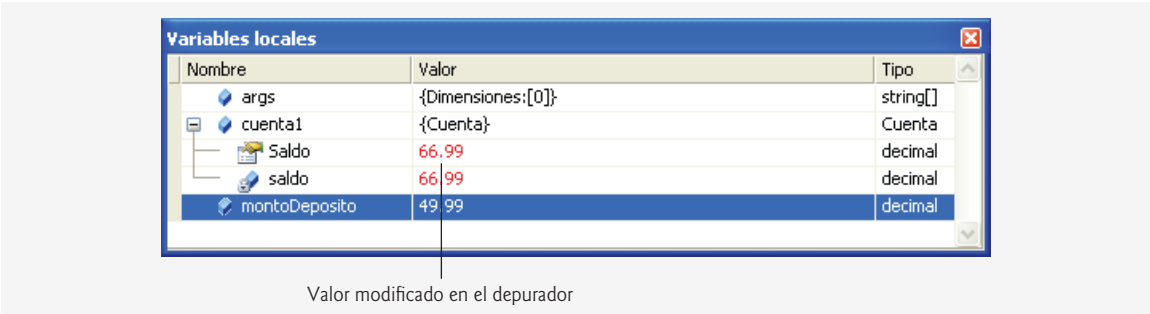


Figura C.15 | Modificar el valor de una variable.

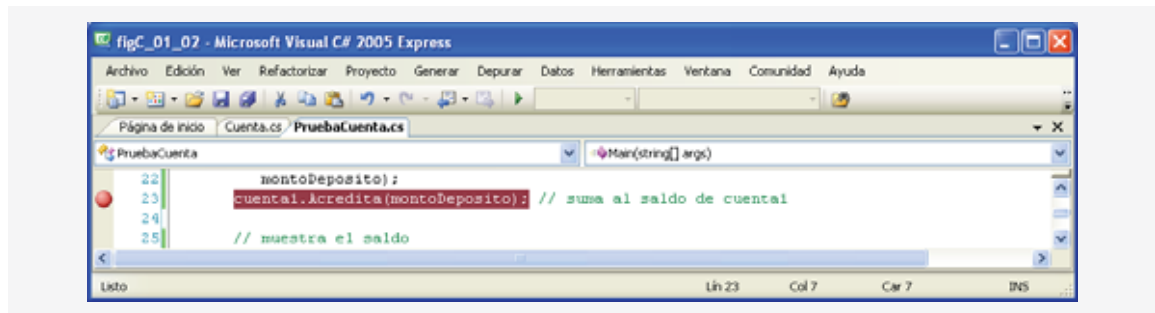


Figura C.16 | Establecimiento de un punto de interrupción en el programa.

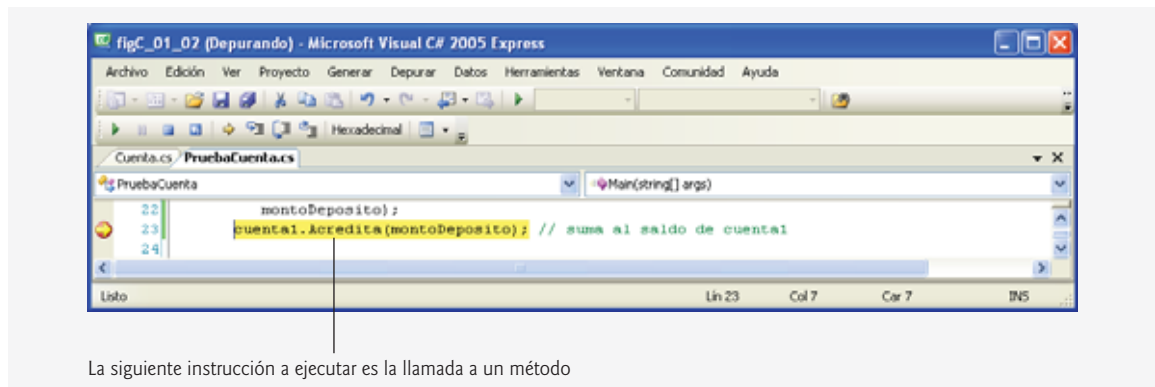


Figura C.17 | Uso del comando **Paso a paso por instrucciones** para ejecutar una instrucción.

El comando **Paso a paso por instrucciones** nos permite seguir la ejecución en un método y confirmar su ejecución, al ejecutar en forma individual cada instrucción dentro del método. Seleccione **Depurar > Paso a paso por instrucciones** (u oprima *F11*) para entrar al método *Acredita* (figura C.18).

4. **Uso del comando Paso a paso por procedimientos.** Seleccione **Depurar > Paso a paso por procedimientos** para entrar al cuerpo del método *Acredita* (línea 17 en la figura C.18) y transferir el control a la línea 18 (figura C.19). El **comando Paso a paso por instrucciones** se comporta de manera similar al comando **Paso a paso por instrucciones** cuando la siguiente instrucción a ejecutar no contiene una llamada a un método ni accede a una propiedad. En el *paso 10* verá la diferencia entre el comando **Paso a paso por procedimientos** y el comando **Paso a paso por instrucciones**.

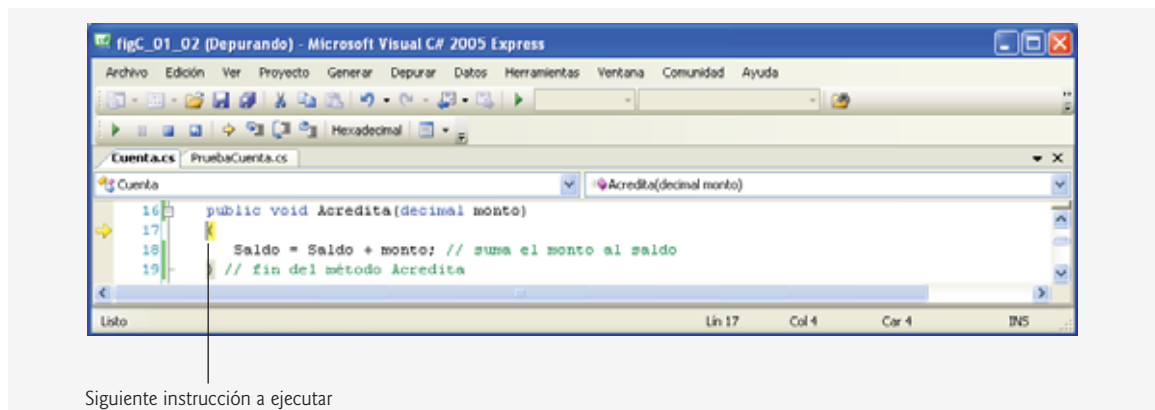


Figura C.18 | Avance paso a paso por el método *Acredita*.

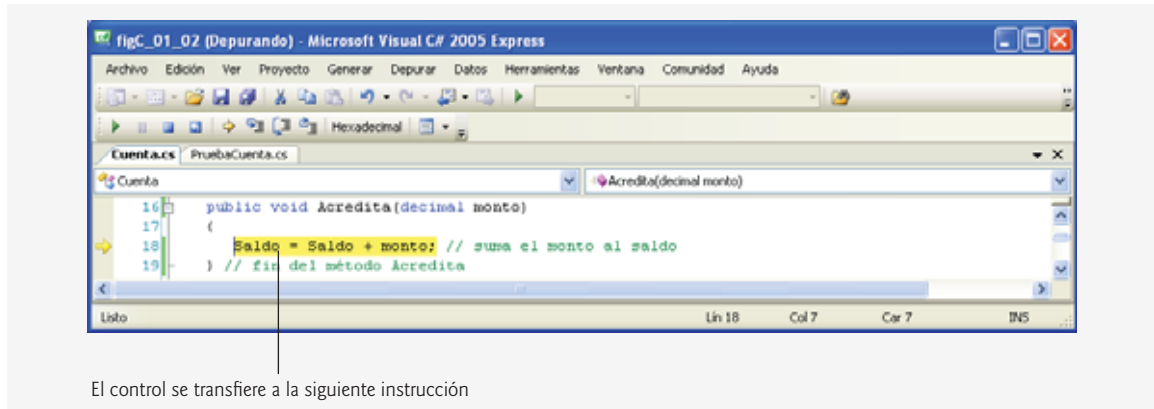


Figura C.19 | Avanzar paso a paso hasta una instrucción dentro del método `Acredita`.

5. *Usar el comando Paso a paso para salir.* Seleccione **Depurar > Paso a paso para salir** para ejecutar el resto de las instrucciones en el método y devolver el control al método que hizo la llamada. A menudo, en métodos extensos, es conveniente analizar unas cuantas líneas clave de código y después continuar depurando el código del método que hace la llamada. El **comando Paso a paso para salir** es útil para ejecutar el resto de un método y regresar al método que hizo la llamada.
6. *Establecer un punto de interrupción.* Establezca un punto de interrupción (figura C.20) en la línea 28 de la figura C.2. Utilizará este punto de interrupción en el siguiente paso.
7. *Usar el comando Continuar.* Seleccione **Depurar > Continuar** para ejecutar el programa hasta llegar al siguiente punto de interrupción en la línea 28. Esta característica ahorra tiempo cuando no deseamos avanzar línea por línea a través de muchas líneas de código, para llegar al siguiente punto de interrupción.
8. *Detener el depurador.* Seleccione **Depurar > Detener depuración** para terminar la sesión de depuración.
9. *Iniciar el depurador.* Antes de poder demostrar la siguiente característica del depurador, debe reiniciarlo. Inicie el depurador como en el *paso 2*, y escriba el mismo valor (49.99). A continuación, el depurador suspenderá la ejecución en la línea 23.
10. *Usar el comando Paso a paso por procedimientos.* Seleccione **Depurar > Paso a paso por procedimientos** (figura C.21). Recuerde que este comando se comporta de manera similar al comando **Paso a paso por instrucciones**, cuando la siguiente instrucción a ejecutar no contiene una llamada a un método. Si la siguiente instrucción a ejecutar contiene una llamada a un método, el método que recibe la llamada se ejecuta en su totalidad (sin pasar la ejecución a ninguna de las instrucciones que contiene, a menos que haya un punto de interrupción en el método), y la flecha avanza a la siguiente línea ejecutable (después

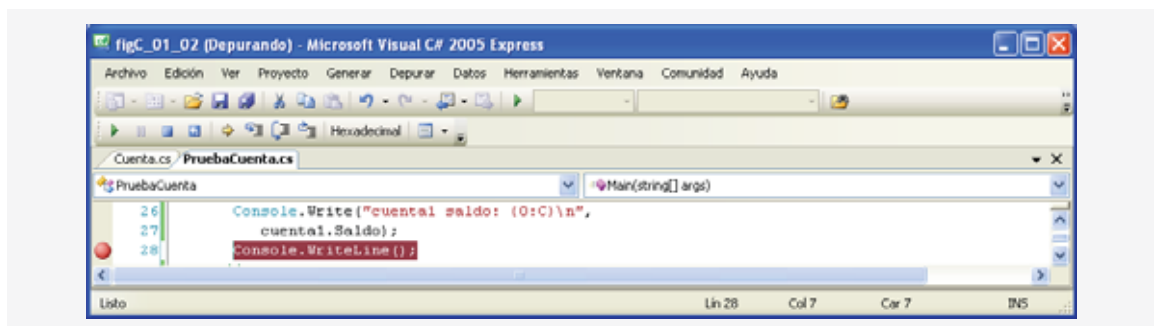
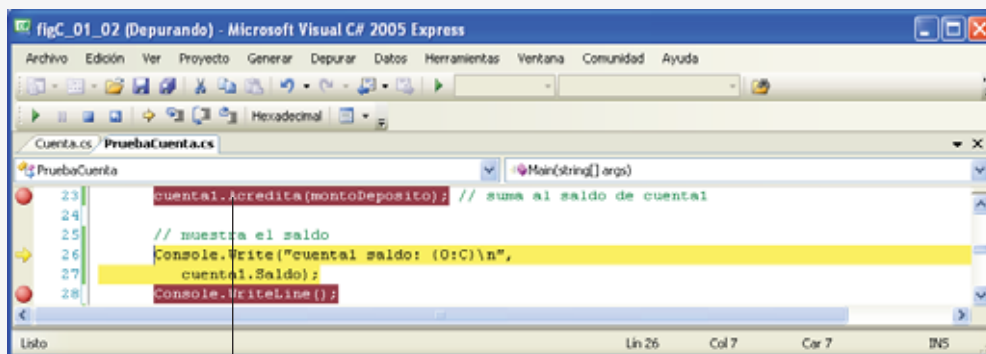


Figura C.20 | Establecimiento de un segundo punto de interrupción en el programa.



El método `Acredita` se ejecuta sin entrar paso a paso en él cuando se selecciona el comando **Paso a paso por procedimientos**.

Figura C.21 | Uso del comando **Paso a paso por procedimientos** del depurador.

de la llamada al método) en el método actual. En este caso, el depurador ejecuta la línea 23, que se encuentra en `Main` (figura C.2). La línea 23 llama al método `Acredita`. Después, el depurador detiene la ejecución en la línea 26, la siguiente instrucción ejecutable.

11. **Detener el depurador.** Seleccione **Depurar > Detener depuración**. Elimina todos los puntos de interrupción restantes.

C.5 Otras características

Visual Studio 2005 proporciona muchas características de depuración que simplifican el proceso de prueba y depuración. En esta sección, hablaremos sobre algunas de estas características.

C.5.1 Editar y Continuar

La característica *Editar y continuar* nos permite realizar modificaciones o cambios en el código en modo de depuración, y después continuar ejecutando el programa sin tener que recompilar el código.

1. **Establecer un punto de interrupción.** Establezca un punto de interrupción en la línea 19 en su ejemplo (figura C.22).
2. **Iniciar el depurador.** Seleccione **Depurar > Iniciar depuración**. Cuando empieza la ejecución, se muestra el saldo de `cuenta1`. El depurador entra al modo de interrupción cuando llega al punto de interrupción en la línea 19.

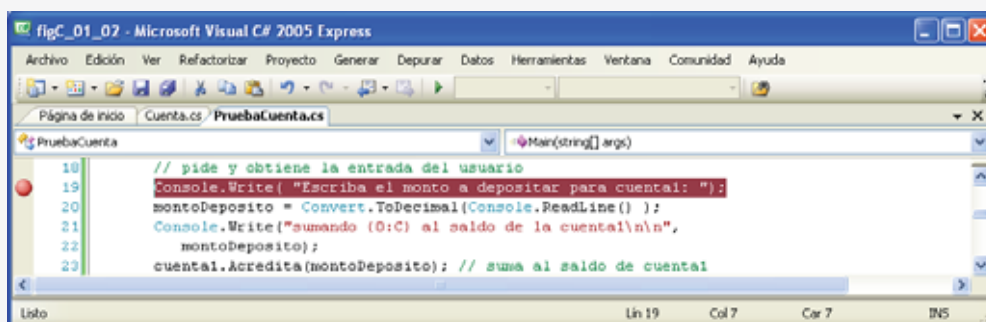


Figura C.22 | Establecimiento de un punto de interrupción en la línea 19.

3. **Modificar el texto de información de entrada.** Suponga que desea modificar el texto indicador de entrada, para proveer al usuario un rango de valores para la variable `montoDeposito`. En vez de detener el proceso de depuración, agregue el texto "(de \$1 a \$500):" al final del mensaje "Escriba el monto a depositar para cuenta1" en la línea 19, en la ventana de vista de código (figura C.23). Seleccione **Depurar > Continuar**. La aplicación le pedirá que introduzca los datos, usando el texto actualizado (figura C.24).

En este ejemplo quisimos hacer una modificación en el texto para nuestro mensaje indicador de entrada antes de ejecutar la línea 19. No obstante, si desea realizar una modificación en una línea que ya se ejecutó, debe seleccionar una instrucción anterior en su código desde la que se va a continuar la ejecución.

1. **Establecer un punto de interrupción.** Establezca un punto de interrupción en la línea 21 (figura C.25).
2. **Iniciar el depurador.** Elimine el texto "(de \$1 a \$500)" que acaba de agregar en los pasos anteriores. Seleccione **Depurar > Iniciar depuración**. Al comenzar la ejecución, aparece el indicador **Escriba el monto a depositar para cuenta 1:**. Introduzca el valor 650 en el indicador (figura C.26). El depurador entrará en el modo de interrupción en la línea 21 (figura C.26).

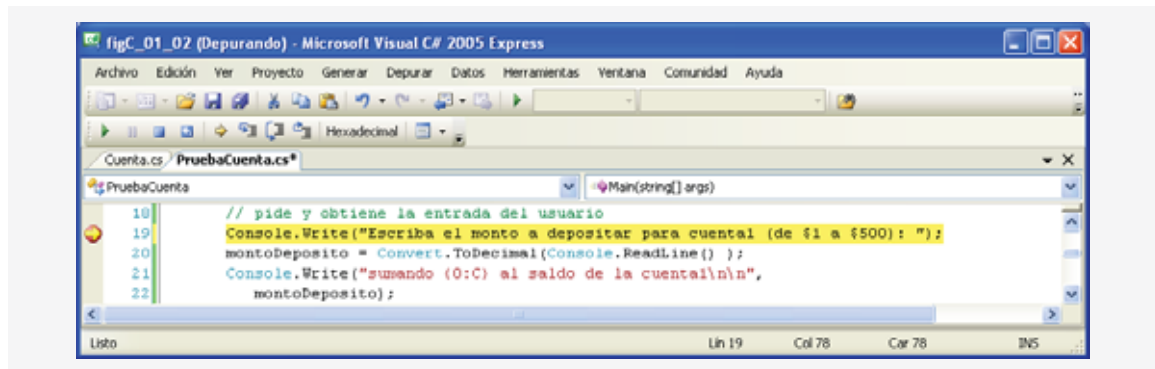


Figura C.23 | Modificación del texto del indicador de entrada, mientras la aplicación se encuentra en modo de **Depuración**.

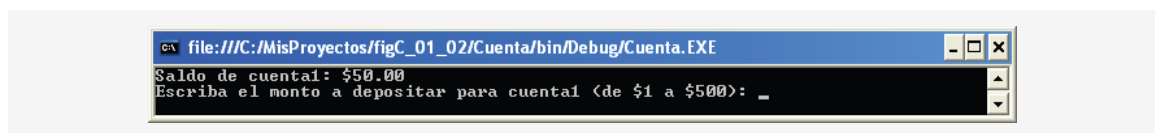


Figura C.24 | Indicador de entrada de la aplicación, que muestra el texto actualizado.



Figura C.25 | Establecimiento de un punto de interrupción en la línea 21.

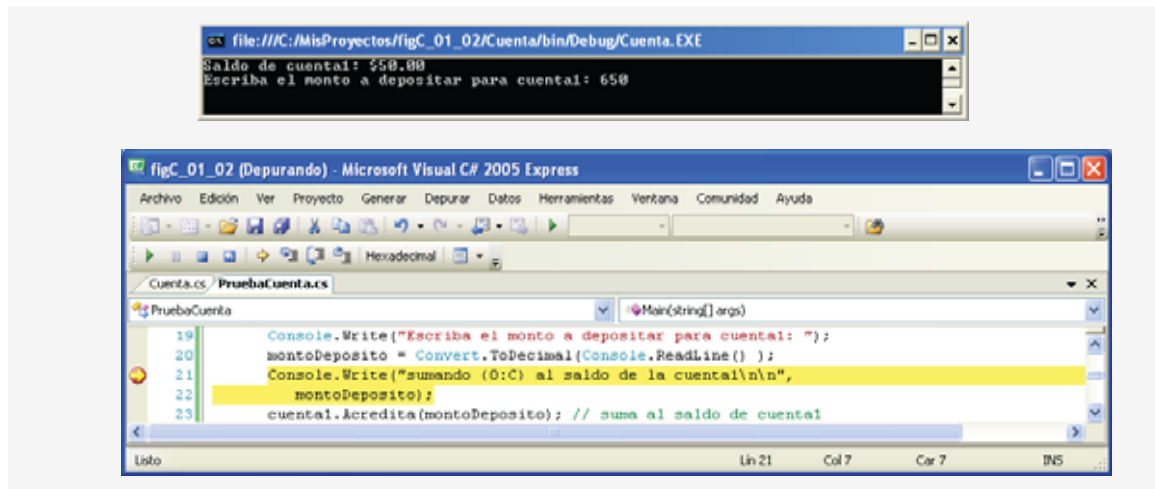


Figura C.26 | Detener la ejecución en el punto de interrupción en la línea 21.

3. *Modificar el texto del indicador de entrada.* Digamos que una vez más desea modificar el texto del indicador de entrada, para proveer al usuario un rango de valores para la variable `montoDeposito`. Agregue el texto "(de \$1 a \$500):" al final del mensaje "Escriba el monto a depositar para cuenta1" en la línea 19, dentro de la ventana de vista de código.
4. *Establecer la siguiente instrucción.* Para que el programa actualice el texto del indicador de entrada en forma correcta, debe establecer el punto de ejecución en una línea anterior de código. Haga clic con el botón derecho del ratón en la línea 16 y seleccione la opción **Establecer instrucción siguiente** del menú que aparece (figura C.27).
5. Seleccione **Depurar > Continuar**. La aplicación le pedirá de nuevo la entrada, usando el texto actualizado (figura C.28).
6. *Detener el depurador.* Seleccione **Depurar > Detener depuración**.

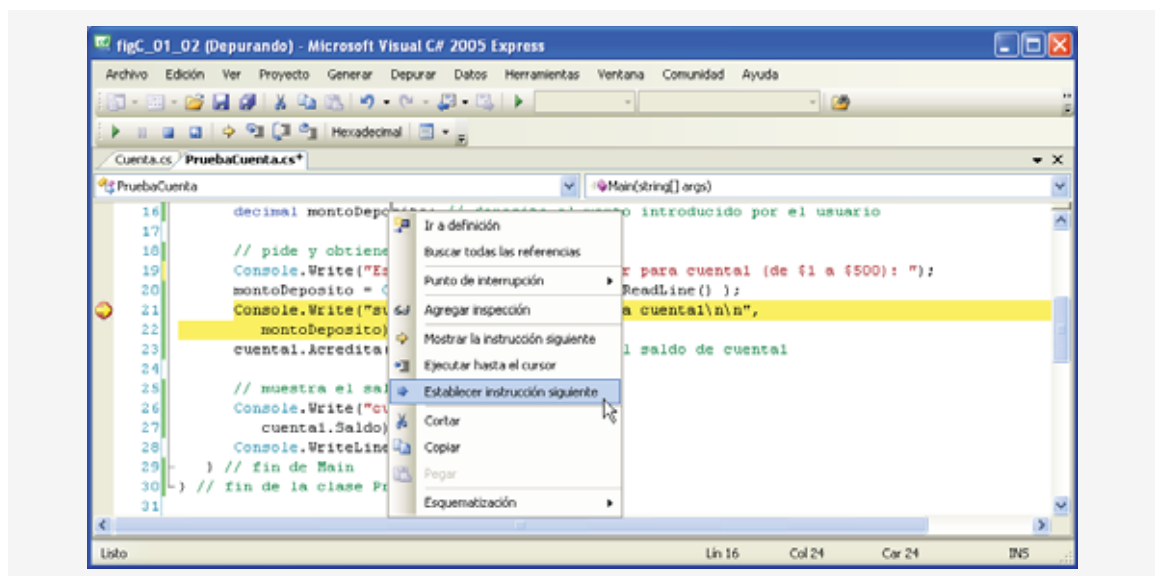


Figura C.27 | Establecer la siguiente instrucción a ejecutar.

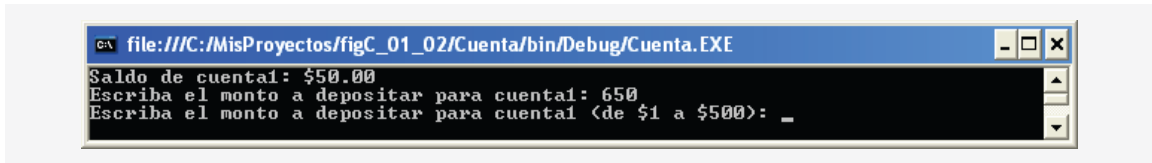


Figura C.28 | La ejecución del programa continúa con el texto del indicador de entrada actualizado.

Una vez que el programa empieza a ejecutarse, no se permiten ciertos tipos de modificaciones con la característica Editar y Continuar. Estas modificaciones incluyen cambiar los nombres de las clases, agregar o eliminar parámetros de los métodos, agregar campos `public` a una clase, y agregar o eliminar métodos. Si desea hacer una modificación específica a su programa, que no se permita durante el proceso de depuración, Visual Studio muestra un cuadro de diálogo como el de la figura C.29.

C.5.2 Ayudante de excepciones

El Asistente de excepciones es otra nueva característica en Visual Studio 2005. Puede ejecutar un programa, seleccionando ya sea **Depurar > Iniciar depuración** o **Depurar > Iniciar sin depurar**. Si selecciona la opción **Depurar > Iniciar depuración** y el entorno en tiempo de ejecución detecta excepciones sin atrapar, la aplicación se detiene y aparece una ventana llamada **Ayudante de excepciones**, la cual indica en dónde ocurrió la excepción, el tipo de la misma y vínculos a información útil acerca de cómo manejar la excepción. En la sección 12.4.3 hablamos con detalle sobre el Ayudante de excepciones.

C.5.3 Depuración Sólo mi código (Just My Code™)

A lo largo de este libro hemos producido programas cada vez más robustos, que a menudo incluyen una combinación de código escrito por el programador y código generado por Visual Studio. El código generado por el IDE puede ser difícil de entender para los principiantes (e incluso hasta para los programadores experimentados); por fortuna, raras veces necesitará ver este código. Visual Studio 2005 cuenta con una nueva característica de depuración llamada *Sólo mi código* (*Just My Code™*), la cual permite a los programadores probar y depurar sólo la parte del código que escribieron. Cuando se habilita esta opción, el depurador siempre avanza paso a paso, ignorando las llamadas a los métodos de las clases que usted no escribió.

Puede modificar esta configuración en las opciones del depurador. Seleccione **Herramientas > Opciones**. En el cuadro de diálogo **Opciones**, seleccione la categoría **Depuración** para ver las herramientas de depuración y las opciones disponibles. Después haga clic en la casilla de verificación que aparezca enseguida de la opción **Habilitar Sólo mi código (sólo administrado)** (figura C.30), para habilitar o deshabilitar esta característica.

C.5.4 Otras nuevas características del depurador

Todas las características que hemos visto hasta ahora en esta sección están disponibles en todas las versiones de Visual Studio, incluyendo Visual C# 2005 Express Edition. El depurador de Visual Studio 2005 ofrece nuevas características adicionales, como visualizadores, puntos de seguimiento y más, de las que encontrará más información en [msdn2.microsoft.com/es-es/library/01xdt7cs\(VS.80\).aspx](http://msdn2.microsoft.com/es-es/library/01xdt7cs(VS.80).aspx).

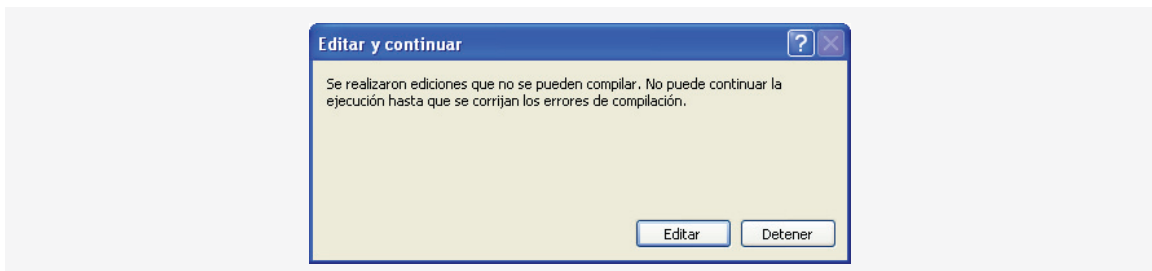


Figura C.29 | Cuadro de diálogo que indica que ciertas ediciones en el programa no se permiten durante su ejecución.

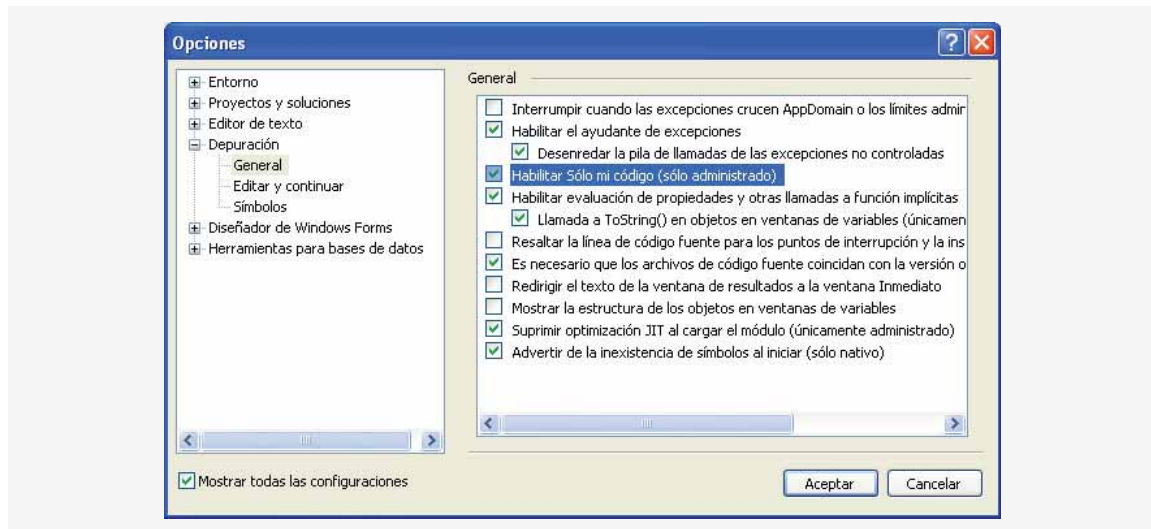


Figura C.30 | Habilitar la característica de depuración **Sólo mi código** en Visual Studio.

C.6 Conclusión

En este apéndice aprendió a habilitar el depurador y establecer puntos de interrupción, para poder examinar su código y los resultados mientras se ejecuta un programa. Esta capacidad le permite localizar y corregir los errores lógicos en sus programas. También aprendió a continuar la ejecución después de que un programa suspende su ejecución en un punto de interrupción, y cómo deshabilitar y eliminar los puntos de interrupción.

Le mostramos cómo utilizar las ventanas **Inspección** y **Variables locales** del depurador para evaluar expresiones aritméticas y booleanas. También demostramos cómo modificar el valor de una variable durante la ejecución del programa, para poder ver cómo los cambios en los valores afectan a sus resultados.

Aprendió a utilizar el comando **Paso a paso por instrucciones** del depurador, para depurar métodos a los que se llama durante la ejecución de su programa. Vio cómo puede usarse el comando **Paso a paso por procedimientos** para ejecutar la llamada a un método sin detener el método al que se llamó. Utilizó el comando **Paso a paso para salir** para continuar la ejecución hasta el final del método actual. También aprendió que el comando **Continuar** continúa la ejecución hasta encontrar otro punto de interrupción o el fin del programa.

Por último, hablamos sobre las nuevas características del depurador de Visual Studio 2005, incluyendo Editar y Continuar, el Ayudante de excepciones y la depuración Sólo mi código (Just My Code).



Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	n1	vt	ff	cr	so	si	d1e	dc1	c2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

Figura D.1 | El conjunto de caracteres ASCII.

La columna izquierda de la tabla contiene los dígitos izquierdos de los equivalentes decimales (0-127) del código de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos del código de caracteres. Por ejemplo, el código de carácter para la “F” es 70, mientras que el código de carácter para el “&” es 38.

La mayoría de los usuarios de este libro estarán interesados en el conjunto de caracteres ASCII utilizado para representar los caracteres del idioma español en muchas computadoras. El conjunto de caracteres ASCII es un subconjunto del conjunto de caracteres Unicode utilizado por C# para representar caracteres de la mayoría de los lenguajes existentes en el mundo. Para obtener más información acerca del conjunto de caracteres Unicode, vea el apéndice E.



Unicode®

OBJETIVOS

En este apéndice aprenderá lo siguiente:

- Los fundamentos de Unicode.
- La misión del consorcio Unicode.
- La base del diseño de Unicode.
- Las tres formas de codificación de Unicode: UTF-8, UTF-16 y UTF-32.
- Los caracteres y glifos.
- Las ventajas y desventajas de usar Unicode.

- E.1 Introducción
- E.2 Formatos de transformación de Unicode
- E.3 Caracteres y glifos
- E.4 Ventajas y desventajas de Unicode
- E.5 Uso de Unicode
- E.6 Rangos de caracteres

E.1 Introducción

El uso de *codificaciones* de caracteres (es decir, valores numéricos asociados con caracteres) inconsistentes al desarrollar productos de software globales provoca graves problemas, ya que las computadoras procesan la información utilizando números. Por ejemplo, el carácter “a” se convierte en un valor numérico para que una computadora pueda manipular esa pieza de información. Muchos países y corporaciones han desarrollado sus propios sistemas de codificación, que son incompatibles con los sistemas de codificación de otros países y corporaciones. Por ejemplo, el sistema operativo Microsoft Windows asigna el valor 0xC0 al carácter “A con un acento grave”; mientras que el sistema operativo Apple Macintosh asigna ese mismo valor a un signo de interrogación de apertura. Esto produce una mala representación y la posible corrupción de los datos, ya que éstos no se procesan de la manera esperada.

A falta de un estándar universal de codificación de caracteres implementado ampliamente, los desarrolladores de software de todo el mundo tuvieron que localizar sus productos extensivamente, antes de distribuirlos. La *localización* incluye la traducción del lenguaje y la adaptación cultural del contenido. El proceso de localización generalmente incluye modificaciones considerables al código fuente (como la conversión de valores numéricos y las suposiciones subyacentes por parte de los programadores), lo cual produce un aumento en costos y retrasos en la liberación del software. Por ejemplo, algunos programadores de habla inglesa podrían diseñar productos de software global suponiendo que un solo carácter puede ser representado por un byte. Sin embargo, cuando esos productos se localizan en mercados asiáticos las suposiciones del programador ya no son válidas, por lo cual la mayor parte del código (si no es que todo) necesita volver a escribirse. La localización es necesaria cada vez que se libera una versión. Para cuando el producto de software se localiza para un mercado específico ya está lista una nueva versión, que también necesita localizarse. Como resultado, es muy complicado y costoso producir y distribuir productos de software global en un mercado en el que no existe un estándar universal de codificación de caracteres.

En respuesta a esta situación se creó el *Estándar Unicode*, que facilita la producción y distribución de software; este estándar resume una especificación para producir la codificación consistente de los caracteres y símbolos de todo el mundo. Los productos de software que se encargan del texto codificado en el estándar Unicode necesitan localizarse, pero este proceso es más simple y eficiente debido a que los valores numéricos no necesitan convertirse y las suposiciones de los programadores acerca de la codificación de caracteres son universales. El estándar Unicode es mantenido por una organización sin fines de lucro conocida como *Consorcio Unicode*, cuyos miembros incluyen a Apple, IBM, Microsoft, Oracle, Sun Microsystems, Sybase y muchos otros.

Cuando el Consorcio ideó y desarrolló el estándar Unicode, querían un sistema de codificación que fuera universal, eficiente, uniforme y sin ambigüedades. Un sistema de codificación universal comprende a todos los caracteres que se utilizan comúnmente. Un sistema de codificación eficiente permite que los archivos de texto se analicen con facilidad. Un sistema de codificación uniforme asigna valores fijos a todos los caracteres. Un sistema de codificación sin ambigüedades representa a un carácter dado en una manera consistente. A estos cuatro términos se les conoce como la *base del diseño* del estándar Unicode.

E.2 Formatos de transformación de Unicode

Aunque Unicode incorpora el conjunto de caracteres (es decir, una colección de caracteres) ASCII limitado, comprende un conjunto de caracteres más comprensivo. En ASCII, cada carácter está representado por un byte que contiene 0s y 1s. Un byte es capaz de almacenar los números binarios de 0 a 255. A cada carácter se le asigna un número entre 0 y 255, por lo cual los sistemas basados en ASCII sólo soportan 256 caracteres, una minúscula fracción de los caracteres existentes en el mundo. Unicode extiende el conjunto de caracteres ASCII al codificar la gran mayoría

de los caracteres de todo el mundo. El estándar Unicode codifica a todos esos caracteres en un espacio numérico uniforme, de 0 a 10FFFF en hexadecimal. Una implementación expresará estos números en uno de varios formatos de transformación, seleccionando el que se adapte mejor a la aplicación específica en consideración.

Hay tres de esos formatos en uso: *UTF-8*, *UTF-16* y *UTF-32*, dependiendo del tamaño de las unidades (en bits) utilizadas. UTF-8, un formato de codificación de anchura variable, requiere de uno a cuatro bytes para expresar cada uno de los caracteres Unicode. Los datos de UTF-8 consisten en bytes de 8 bits (secuencias de uno, dos, tres o cuatro bytes, dependiendo del carácter que se vaya a codificar) y son bastante adecuados para los sistemas basados en ASCII, cuando predominan los caracteres de un byte (ASCII representa los caracteres como un byte). En la actualidad, UTF-8 se implementa ampliamente en sistemas UNIX y bases de datos.

El formato de codificación UTF-16 de anchura variable expresa los caracteres Unicode en unidades de 16 bits (es decir, como dos bytes adyacentes, o un entero corto en muchos equipos). La mayoría de los caracteres de Unicode se expresan en una sola unidad de 16 bits. Sin embargo, los caracteres con valores por arriba de FFFF en hexadecimal se expresan mediante un par ordenado de unidades de 16 bits, conocidas como *sustitutos*. Los sustitutos son enteros de 16 bits en el rango de D800 a DFFF, que se utilizan únicamente para el propósito de “escapar” hacia caracteres con una numeración más alta. De esta forma pueden expresarse aproximadamente un millón de caracteres. Aunque un par de sustitutos requiere de 32 bits para representar caracteres, al utilizar estas unidades de 16 bits el uso del espacio se hace eficiente. Los sustitutos son caracteres raros en las implementaciones actuales. Muchas implementaciones para el manejo de cadenas se escriben en términos de UTF-16. [Nota: los detalles y el código de ejemplo para el manejo de UTF-16 están disponibles en el sitio Web del consorcio Unicode, en www.unicode.org.]

Las implementaciones que requieren de un uso considerable de caracteres raros o secuencias de comandos completas con una codificación por encima del FFFF hexadecimal deben usar UTF-32, un formato de codificación de 32 bits con anchura fija que, por lo general, requiere del doble de memoria que los caracteres codificados con UTF-16. La principal ventaja del formato de codificación UTF-32 de anchura fija es que expresa uniformemente a todos los caracteres, por lo que es fácil de manejar en los arreglos.

La figura E.1 muestra las distintas formas en las que los tres formatos manejan la codificación de caracteres. Hay unos cuantos lineamientos que indican cuándo utilizar un formato de codificación específico. El mejor formato a utilizar depende de los sistemas computacionales y los protocolos de negocios, no de la información en sí. Generalmente debe utilizarse el formato de codificación UTF-8 cuando los sistemas computacionales y los protocolos de negocios requieren que la información se maneje en unidades de 8 bits, especialmente en los sistemas heredados que se están actualizando, ya que esto a menudo simplifica los cambios que deben realizarse en los programas existentes. Por esta razón, UTF-8 se ha convertido en el formato de codificación preferido en Internet. De la misma forma, UTF-16 es el formato de codificación preferido en aplicaciones para Microsoft Windows. Es probable que UTF-32 se utilice más ampliamente en el futuro, a medida que se codifiquen más caracteres con valores por encima del FFFF hexadecimal. Además, UTF-32 requiere de un manejo menos sofisticado que UTF-16, debido a la presencia de los pares de sustitutos.

E.3 Caracteres y glifos

El Estándar Unicode consiste de caracteres, componentes escritos (es decir, alfabetos, números, signos de puntuación, acentos, etc.) que pueden ser representados por valores numéricos. Algunos ejemplos de caracteres son: U+0041, letra A mayúscula en latín. En la primera representación de caracteres, U+ *yyyy* es un *valor de código*, en

Carácter	UTF-8	UTF-16	UTF-32
Letra A mayúscula en latín	0x41	0x0041	0x00000041
Letra ALFA mayúscula en griego	0xCD 0x91	0x0391	0x00000391
Ideograma unificado CJK-4E95	0xE4 0xBA 0x95	0x4E95	0x00004E95
Letra A en cursiva antigua	0xF0 0x80 0x83 0x80	0xDC00 0xDF00	0x00010300

Figura E.1 | La correlación entre los tres formatos de codificación.

donde U+ se refiere a los valores de código de Unicode, a diferencia de los demás valores hexadecimales. Las letras *yyyy* representan un número hexadecimal de cuatro dígitos, perteneciente a un carácter codificado. Los valores de códigos son combinaciones de bits que representan caracteres codificados. Los caracteres se representan utilizando *glifos*, varias figuras, tipos de letra y tamaños para mostrar caracteres. No hay valores de código para los glifos en el estándar Unicode. En la figura E.2 se muestran ejemplos de glifos.

El estándar Unicode comprende los alfabetos, ideogramas, silabarios, signos de puntuación, *signos diacríticos*, operadores matemáticos, etc., que comprenden los lenguajes escritos y manuscritos del mundo. Un signo diacrítico es un signo especial que se agrega a un carácter para diferenciarlo de otra letra, o para indicar un acento (por ejemplo, en español se utiliza la tilde “~” por encima del carácter “n”). Actualmente, Unicode proporciona los valores de código para 94,140 representaciones de caracteres, con más de 880,000 valores de código reservados para una expansión en el futuro.

E.4 Ventajas y desventajas de Unicode

El estándar Unicode tiene varias ventajas considerables que promueven su uso. Una es el impacto que tiene sobre el rendimiento de la economía internacional. Unicode estandariza los caracteres para los sistemas de escritura mundiales en un modelo uniforme que promueve la transferencia y la compartición de información. Los programas que se desarrollan usando este esquema mantienen su precisión, ya que cada carácter tiene una sola definición (es decir, *a* es siempre U+0061, *%* es siempre U+0025). Esto permite a las corporaciones administrar las altas demandas de los mercados internacionales, al procesar distintos sistemas de escritura al mismo tiempo. Además, los caracteres pueden administrarse en forma idéntica, evitando así la confusión ocasionada por las distintas arquitecturas de códigos de caracteres. Lo que es más, al administrar los datos de una manera consistente se elimina la corrupción de los mismos, ya que la información puede ordenarse, buscarse y manipularse utilizando un proceso consistente.

Otra ventaja del estándar Unicode es la portabilidad (es decir, software que puede ejecutarse en computadoras distintas o con distintos sistemas operativos). La mayoría de los sistemas operativos, bases de datos, lenguajes de programación y navegadores Web soportan actualmente, o planean soportar, Unicode.

Una desventaja del estándar Unicode es la cantidad de memoria que requieren UTF-16 y UTF-32. Los conjuntos de caracteres ASCII tienen una longitud de 8 bits, por lo que requieren de una menor capacidad de almacenamiento que el conjunto de caracteres Unicode predeterminado de 16 bits. Sin embargo, el *conjunto de caracteres de doble byte (DBCS)* y el *conjunto de caracteres multibyte (MBCS)* que codifican caracteres asiáticos (ideogramas) requieren de dos a cuatro bytes, respectivamente. En tales casos, pueden usarse los formatos de codificación UTF-16 o UTF-32 con pocas restricciones en cuanto a memoria y rendimiento.

E.5 Uso de Unicode

Visual Studio utiliza la codificación UTF-16 de Unicode para representar a todos los caracteres. La figura E.3 usa C# para mostrar el texto “Bienvenido a Unicode” en ocho lenguajes: inglés, francés, alemán, japonés, portugués, ruso, español y chino tradicional. [Nota: el sitio Web del Consorcio Unicode contiene un vínculo a las tablas de caracteres en las que se enlistan los valores de código de Unicode de 16 bits.]

El primer mensaje de bienvenida (líneas 19-23) contiene los códigos hexadecimales para el texto en inglés. La página **Code Charts** en el sitio Web del Consorcio Unicode contiene un documento que lista los valores de códigos para el bloque (o categoría) **Basic Latin** (latín básico), que incluye el alfabeto en inglés. Los códigos hexadecimales en las líneas 19-20 corresponden a “Bienvenido” y un carácter de espacio (\u0020). Los caracteres de Unicode en C# utilizan el formato `\uyyyy`, en donde *yyyy* representa la codificación Unicode hexadecimal. Por ejemplo, la letra “B” (de “Bienvenido”) se representa por el código `\u0057`. Los valores hexadecimales para la palabra “a” y un carácter de espacio aparecen en la línea 21, y la palabra “Unicode” en la línea 23. “Unicode” no está codificada, ya que es una marca registrada y no tiene traducción equivalente en la mayoría de los lenguajes.



Figura E.2 | Distintos glifos para la letra A.

```

1  // Fig. E.3: UnicodeForm.cs
2  // Demostración de la codificación Unicode.
3  using System;
4  using System.Windows.Forms;
5
6  namespace DemoUnicode
7  {
8      public partial class UnicodeForm : Form
9      {
10         public UnicodeForm()
11         {
12             InitializeComponent();
13         }
14
15         // asigna cadenas Unicode a cada control Label
16         private void UnicodeForm_Load( object sender, EventArgs e )
17         {
18             // Inglés
19             char[] ingles = { '\u0057', '\u0065', '\u006C',
20                             '\u0063', '\u006F', '\u006D', '\u0065', '\u0020',
21                             '\u0074', '\u006F', '\u0020' };
22             inglesLabel.Text = new string( ingles ) +
23                             "Unicode" + '\u0021';
24
25             // Francés
26             char[] frances = { '\u0042', '\u0069', '\u0065',
27                               '\u006E', '\u0076', '\u0065', '\u006E', '\u0075',
28                               '\u0065', '\u0020', '\u0061', '\u0075', '\u0020' };
29             francesLabel.Text = new string( frances ) +
30                             "Unicode" + '\u0021';
31
32             // Alemán
33             char[] aleman = { '\u0057', '\u0069', '\u006C',
34                              '\u006B', '\u006F', '\u006D', '\u006D', '\u0065',
35                              '\u006E', '\u0020', '\u0074', '\u0075', '\u0020' };
36             alemanLabel.Text = new string( aleman ) +
37                             "Unicode" + '\u0021';
38
39             // Japonés
40             char[] japones = { '\u3078', '\u3087', '\u3045',
41                               '\u3053', '\u305D', '\u0021' };
42             japonesLabel.Text = "Unicode" + new string( japones );
43
44             // Portugués
45             char[] portugues = { '\u0053', '\u0065', '\u0061',
46                                  '\u0061', '\u0020', '\u0062', '\u0065', '\u006D',
47                                  '\u0020', '\u0076', '\u0069', '\u006E', '\u0064',
48                                  '\u006F', '\u0020', '\u0061', '\u0020' };
49             portuguesLabel.Text = new string( portugues ) +
50                             "Unicode" + '\u0021';
51
52             // Ruso
53             char[] ruso = { '\u0414', '\u043E', '\u0431',
54                            '\u0440', '\u043E', '\u0020', '\u043F', '\u043E',
55                            '\u0436', '\u0430', '\u043B', '\u043E', '\u0432',
56                            '\u0430', '\u0442', '\u044A', '\u0020', '\u0432', '\u0020' };
57             rusoLabel.Text = new string( ruso ) +
58                             "Unicode" + '\u0021';
59

```

Figura E.3 | Aplicación Windows que demuestra la codificación Unicode. (Parte I de 2).

```

60      // Español
61      char[] espaniol = { '\u0042', '\u0069', '\u0065',
62                          '\u006E', '\u0076', '\u0065', '\u006E', '\u0069',
63                          '\u0064', '\u006F', '\u0020', '\u0061', '\u0020' };
64      espaniolLabel.Text = new string( espaniol ) +
65                          "Unicode" + '\u0021';
66
67      // Chino simplificado
68      char[] chino = { '\u6B22', '\u8FCE', '\u4F7F',
69                      '\u7528', '\u0020' };
70      chinoLabel.Text = new string( chino ) +
71                      "Unicode" + '\u0021';
72  } // fin del método UnicodeForm_Load
73 } // fin de la clase UnicodeForm
74 } // fin del espacio de nombres DemoUnicode

```

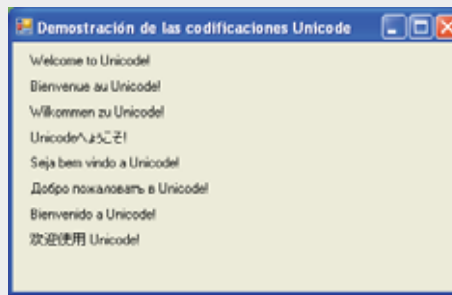


Figura E.3 | Aplicación Windows que demuestra la codificación Unicode. (Parte 2 de 2).

El resto de los mensajes de bienvenida (líneas 26-71) contienen los códigos hexadecimales para los otros siete lenguajes. Los valores de código utilizados para el texto en francés, alemán, portugués y español se encuentran en el bloque **Basic Latin**, los valores de código utilizados para el texto en chino tradicional se encuentran en el bloque **CJK Unified Ideographs**, los valores de código usados para el texto en ruso se encuentran en el bloque **Cyrillic** y los valores de código usados para el texto en japonés se encuentran en el bloque **Hiragana**.

[Nota: para representar los caracteres asiáticos en una aplicación Windows, necesita instalar los archivos de lenguaje apropiados en su computadora. Para ello, abra el cuadro de diálogo **Configuración regional y de idioma** del **Panel de control** (Inicio > Panel de control). Al final de la ficha **Idiomas** hay una lista de lenguajes. Seleccione las casillas de verificación **Japonés** y **Chino tradicional**, y haga clic en **Aplicar**. Siga las indicaciones del asistente de instalación para instalar los lenguajes. Para obtener ayuda adicional, visite www.unicode.org/help/display_problems.html.

E.6 Rangos de caracteres

El estándar Unicode asigna valores de código, que varían de 0000 (**Latín básico**) a E007F (**Marcas**), a los caracteres escritos de todo el mundo. Actualmente existen valores de código para 94,140 caracteres. Para simplificar la búsqueda de un carácter y su valor de código asociado, el estándar Unicode generalmente agrupa los valores de código por *escritura* y función (es decir, los caracteres en latín se agrupan en un bloque, los operadores matemáticos en otro bloque, etc.). Como regla, una escritura es un sistema de escritura individual que se utiliza para varios lenguajes (por ejemplo, la escritura del latín se utiliza para el inglés, francés, español, etc.). La página **Code Charts** (Tablas de códigos) en el sitio Web del Consorcio Unicode enlista todos los bloques definidos y sus respectivos valores de códigos. En la figura E.4 se enlistan algunos bloques (escrituras) del sitio Web y su rango de valores de código.

Escritura	Rango de valores de código
Arábica	U+0600–U+06FF
Latín básico	U+0000–U+007F
Bengalí (La India)	U+0980–U+09FF
Cherokee (nativo de América)	U+13A0–U+13FF
Ideogramas Unificados CJK (Este de Asia)	U+4E00–U+9FAF
Cirílico (Rusia y Este de Europa)	U+0400–U+04FF
Etiope	U+1200–U+137F
Griego	U+0370–U+03FF
Hangul Jamo (Corea)	U+1100–U+11FF
Hebreo	U+0590–U+05FF
Hiragana (Japón)	U+3040–U+309F
Khmer (Cambodia)	U+1780–U+17FF
Lao (Laos)	U+0E80–U+0EFF
Mongol	U+1800–U+18AF
Myanmar	U+1000–U+109F
Ogham (Irlanda)	U+1680–U+169F
Runic (Alemania y Escandinavia)	U+16A0–U+16FF
Sinhala (Sri Lanka)	U+0D80–U+0DFF
Telugu (La India)	U+0C00–U+0C7F
Thai	U+0E00–U+0E7F

Figura E.4 | Algunos rangos de caracteres.



Introducción a XHTML: parte I

OBJETIVOS

En este apéndice aprenderá lo siguiente:

- Comprender los componentes importantes de los documentos en XHTML.
- Utilizar el lenguaje XHTML para crear páginas Web.
- Añadir imágenes a páginas Web.
- Crear y utilizar los hipervínculos para navegar en las páginas Web.
- Realizar el marcado de listas de información.

*Leer entre líneas
fue más fácil que seguir
el texto.*

—Henry James

*Los pensamientos de alto
nivel deben poseer
un lenguaje de alto nivel.*

—Aristófanes

- F.1 Introducción
- F.2 Edición de XHTML
- F.3 El primer ejemplo en XHTML
- F.4 Servicio de validación de XHTML del W3C
- F.5 Encabezados
- F.6 Vínculos
- F.7 Imágenes
- F.8 Caracteres especiales y más saltos de línea
- F.9 Listas desordenadas
- F.10 Listas anidadas y ordenadas
- F.11 Recursos Web

F.1 Introducción

Bienvenidos al mundo de oportunidades creadas por World Wide Web. Internet tiene ya tres décadas de edad pero no fue sino hasta que la Web se volvió popular en la década de 1990 que comenzó la explosión de oportunidades que aún estamos experimentando. Casi a diario ocurren nuevos y excitantes desarrollos; el ritmo de la innovación nunca ha sido compartido por ninguna otra tecnología. En este apéndice desarrollaremos nuestras propias páginas Web. A medida que progreseemos en el libro, crearemos páginas Web cada vez más agradables y poderosas. En la parte final de este libro aprenderemos a crear aplicaciones completas basadas en la Web.

En este apéndice comenzamos a desatar el poder del desarrollo de las aplicaciones basadas en Web con el lenguaje *XHTML* (*Lenguaje de marcado de hipertexto extensible*). En el siguiente apéndice introduciremos técnicas para XHTML más sofisticadas como las *tablas*, que son particularmente útiles para estructurar información que se obtiene de *bases de datos* (en otras palabras, el software que almacena conjuntos estructurados de datos) y las *hojas de estilo en cascada* (CSS), que hacen que las páginas Web sean más agradables a la vista.

A diferencia de los lenguajes de programación por procedimientos como C, Fortran, Cobol y Pascal, XHTML es un *lenguaje de marcado* que especifica el formato del texto que se presenta a través de un explorador Web como *Internet Explorer* de Microsoft o *Netscape*.

Un aspecto importante a tomar en cuenta cuando utilizamos XHTML es la separación de la *presentación* de un documento (es decir, la apariencia del documento cuando se presenta a través de un explorador Web) de la *estructura* de la información de dicho documento. El lenguaje XHTML está basado en el lenguaje HTML (Lenguaje de marcado de hipertexto): una tecnología estandarizada por el Consorcio World Wide Web (W3C). En HTML era común especificar el contenido, la estructura y el formato de un documento. El formato puede especificar en dónde va a colocar el explorador un elemento en una página Web, o el tipo de letra y los colores utilizados para presentar un elemento. La versión 1.1 de XHTML (la versión más reciente del W3C de la Recomendación para XHTML, al momento de publicar este libro) sólo permite que el contenido y la estructura de un documento aparezcan en un documento en XHTML válido, y no su formato. Por lo general, dicho formato se especifica mediante las Hojas de estilo en cascada. Todos nuestros ejemplos en este apéndice se basan en la *Recomendación para XHTML 1.1*.

F.2 Edición de XHTML

En este apéndice escribiremos XHTML en su *formato de código fuente*. También crearemos *documentos XHTML* a través de un editor de texto (por ejemplo, *Bloc de notas*, *Wordpad*, *vi*, *emacs*) y los guardaremos con la extensión de archivo *.html* o *.htm*.



Buena práctica de programación F.1

Asigne a sus documentos nombres de archivo que describan su función. Esta práctica le ayudará a identificar documentos con más rapidez. También ayuda a las personas que desean crear un vínculo a una página, al darles un nombre fácil de recordar. Por ejemplo, si está escribiendo un documento en XHTML que contiene información sobre productos, quizás sea conveniente llamarlo *productos.html*.

Las computadoras que ejecutan software especializado, llamadas *servidores Web*, almacenan documentos en XHTML. Los clientes (en otras palabras, los exploradores Web) solicitan al servidor Web *recursos* específicos, tales como los documentos en XHTML. Por ejemplo, al escribir la dirección `www.deitel.com/libros/descargas.html` en el campo de dirección del explorador Web, se solicita el archivo `descargas.html` al servidor de Web que se ejecuta en `www.deitel.com`. Este documento está ubicado en el servidor, dentro de un directorio llamado `libros`. Por ahora, simplemente colocaremos los documentos en XHTML en nuestro equipo y los abriremos mediante Internet Explorer.

F.3 El primer ejemplo en XHTML

En este apéndice y en el siguiente, presentaremos marcado de XHTML y proporcionaremos capturas de pantalla para mostrar cómo Internet Explorer representa (es decir, visualiza) el XHTML.¹ Cada documento en XHTML que mostramos incluye números de línea para la conveniencia del lector. Estos números de línea no son parte de los documentos en XHTML.

Nuestro primer ejemplo (figura F.1) es un documento en XHTML llamado `principal.html`, que presenta el mensaje “¡Bienvenido a XHTML!” en el explorador.

La línea clave en el programa es la línea 14, la cual comunica al explorador que muestre el mensaje “¡Bienvenido a XHTML!”. Ahora examinemos cada línea del programa.

Las líneas 1-3 son requeridas en los documentos XHTML para conformarse a la sintaxis apropiada de XHTML. Por ahora sólo copiaremos y pegaremos estas líneas en cada documento XHTML que vamos a crear.

Las líneas 5-6 son *comentarios de XHTML*. Los creadores de documentos en XHTML agregan comentarios para mejorar la legibilidad del marcado y para describir el contenido de un documento. Los comentarios también ayudan a que otras personas puedan leer y comprender el marcado y el contenido de un documento en XHTML.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.1: principal.html -->
6  <!-- Nuestra primera pagina Web -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10      <title>Cómo programar en Internet y WWW - Bienvenidos</title>
11    </head>
12
13    <body>
14      <p>¡Bienvenidos a XHTML!</p>
15    </body>
16  </html>

```

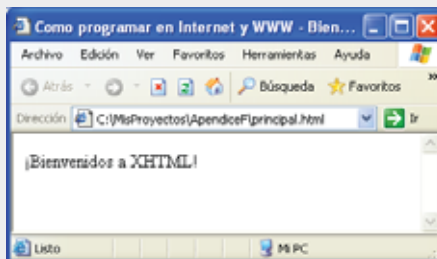


Figura F.1 | Primer ejemplo en XHTML.

1. Todos los ejemplos presentados en este libro están disponibles en `www.deitel.com`.

Los comentarios no requieren de ninguna acción del explorador Web, cuando el usuario carga el documento en XHTML para visualizarlo. Los comentarios en XHTML siempre comienzan con `<!--` y terminan con `-->`. Cada uno de nuestros ejemplos en XHTML incluye comentarios que especifican el número de la figura y el nombre del archivo, y proporcionan una breve descripción del propósito del ejemplo. Los ejemplos que siguen incluyen comentarios en el marcado y, en especial, para resaltar nuevas características.



Buena práctica de programación F.2

Agregue comentarios a lo largo de su marcado. Los comentarios ayudan a que otros programadores puedan comprender el marcado, asisten en la depuración y listan información útil que usted no quiera que muestre el explorador. Los comentarios también nos ayudan a entender nuestro propio marcado cuando regresamos a un documento para modificarlo o actualizarlo en el futuro.

El marcado en XHTML contiene texto que representa el contenido de un documento y los *elementos* que especifican la estructura del mismo. Algunos elementos importantes de un documento en XHTML son el elemento **html**, el elemento **head** y el elemento **body**. El elemento **html** enmarca la *sección del encabezado* (representada por el elemento **head**) y la *sección del cuerpo* (representada por el elemento **body**). La sección del encabezado contiene información acerca del documento en XHTML, tal como su *título*. La sección del encabezado también puede contener instrucciones de formato especial para documentos, llamadas *hojas de estilo*, y programas del lado cliente llamados *secuencias de comandos*, para crear páginas Web dinámicas. La sección del cuerpo contiene el contenido de la página que el explorador presenta cuando el usuario visita la página Web.

Los documentos en XHTML delimitan a un elemento con las etiquetas *inicial* y *final*. Una etiqueta inicial consiste en el nombre de un elemento entre los signos `<` y `>` (por ejemplo, `<html>`). Una etiqueta final consiste en el nombre del elemento precedido por un `/`, entre los signos `<` y `>` (por ejemplo, `</html>`). En este ejemplo, las líneas 8 y 16 definen el inicio y el fin del elemento **html**. Observe que la etiqueta final en la línea 16 tiene el mismo nombre que la etiqueta inicial, pero va precedida por un `/` dentro de los signos `<` y `>`. Muchas etiquetas iniciales tienen *atributos*, los cuales proporcionan información adicional acerca de un elemento. Los exploradores pueden utilizar esta información adicional para determinar cómo procesar el elemento. Cada atributo tiene un *nombre* y un *valor*, separados por un signo de igual (`=`). La línea 8 especifica el atributo requerido (`xmlns`) y el valor (`http://www.w3.org/1999/xhtml`) para el elemento **html** en un documento XHTML. Por ahora simplemente copiaremos y pegaremos la etiqueta de inicio para el elemento **html** de la línea 8 en nuestros documentos en XHTML.



Error común de programación F.1

Si no enmarcamos los valores de los atributos con comillas sencillas o comillas dobles, se produce un error de sintaxis. Aún así, es posible que algunos exploradores Web puedan representar el elemento correctamente.



Error común de programación F.2

Si utilizamos letras mayúsculas en un elemento en XHTML, o en el nombre de un atributo, se produce un error de sintaxis. Sin embargo, es posible que algunos exploradores Web puedan representar el elemento correctamente.

Un documento en XHTML divide el elemento **html** en dos secciones: el encabezado y el cuerpo. Las líneas 9-11 definen la sección del encabezado de la página Web con un elemento **head**. La línea 10 especifica un elemento **title**. A éste se le conoce como *elemento anidado*, debido a que está enmarcado dentro de las etiquetas de inicio y fin del elemento **head**. El elemento **head** es también un elemento anidado, ya que está enmarcado dentro de las etiquetas de inicio y fin del elemento **html**. El elemento **title** (título) describe la página Web. Por lo general, los títulos aparecen en la *barra de título*, en la parte superior de la ventana del explorador, y también aparecen como el texto que identifica a una página cuando el usuario añade la página en su lista de **Favoritos**, que les permiten regresar a sus sitios Web favoritos. Los algoritmos de búsqueda (en otras palabras, los sitios que permiten a los usuarios realizar búsquedas en la Web) también utilizan el título (**title**) para propósitos de clasificación.



Buena práctica de programación F.3

El uso de sangría en los elementos anidados enfatiza la estructura del documento y promueve su legibilidad.



Error común de programación F.3

XHTML no permite que las etiquetas se traslapen; la etiqueta final de un elemento anidado debe aparecer en el documento antes de la etiqueta final del elemento que lo está enmarcando. Por ejemplo, las etiquetas de XHTML anidadas `<head><title>hola</head></title>` generan un error de sintaxis, debido a que la etiqueta final `</html>` del elemento `head` aparece antes de la etiqueta final `</title>` del elemento anidado `title`.



Buena práctica de programación F.4

Use una nomenclatura consistente para los títulos de todas las páginas en un sitio. Por ejemplo, si el sitio se llama “El sitio Web de Bailey”, entonces el título de la página con los vínculos podría ser “El sitio Web de Bailey – Vínculos”. Esta práctica puede ayudar a que los usuarios comprendan mejor la estructura del sitio Web.

La línea 13 abre el elemento `body` (cuerpo) del documento. La sección del cuerpo de un documento en XHTML especifica el contenido del documento, el cual puede incluir texto y etiquetas.

Algunas etiquetas, tales como las *etiquetas de párrafo* (`<p>` y `</p>`) en la línea 14, marcan al texto para presentarlo en un explorador Web. Todo el texto incluido entre las etiquetas `<p>` y `</p>` representa un solo párrafo. Cuando el explorador representa un párrafo, se coloca una línea en blanco antes y después del texto del párrafo.

Este documento termina con dos etiquetas finales (líneas 15-16). Estas etiquetas cierran los elementos `body` y `html`, respectivamente. La etiqueta `</html>` en un documento en XHTML informa al explorador que ha terminado el marcado en XHTML.

Para visualizar este ejemplo en Internet Explorer, realice los siguientes pasos:

1. Copie los ejemplos del apéndice F en su computadora (puede descargar los ejemplos de www.deitel.com).
2. Inicie Internet Explorer y seleccione la opción **Abrir...** del menú **Archivo**. A continuación aparecerá el cuadro de diálogo **Abrir**.
3. Haga clic en el botón **Explorar...** del cuadro de diálogo **Abrir** para mostrar el cuadro de diálogo de archivos de Microsoft Internet Explorer.
4. Navegue hasta el directorio que contiene los ejemplos del apéndice F y seleccione el archivo llamado `principal.html`; después haga clic en **Abrir**.
5. Haga clic en **Aceptar** para que Internet Explorer visualice el documento. Los otros ejemplos pueden abrirse de una manera similar.

En este punto, la ventana de su explorador deberá tener una apariencia similar a la captura de pantalla del ejemplo que se muestra en la figura F.1. (Observe que hemos cambiado el tamaño de la ventana del explorador para ahorrar espacio en el libro.)

F.4 Servicio de validación de XHTML del W3C

La programación de aplicaciones basadas en Web puede ser muy compleja y los documentos en XHTML deben escribirse en forma correcta, para asegurar que los exploradores los procesen de la manera apropiada. Para promover la escritura correcta de documentos, el Consorcio World Wide Web (W3C) proporciona un *servicio de validación* (validator.w3.org) para verificar la sintaxis de un documento. Los documentos pueden validarse, ya sea desde una dirección que especifique la ubicación de un archivo, o enviando un archivo al sitio validator.w3.org/file-upload.html. Al enviar un archivo, éste se copia de la computadora del usuario a otra computadora en Internet. La figura F.2 muestra cómo se envía el documento `principal.html` (figura F.1) para su validación. La página Web del W3C indica que el nombre del servicio es **Servicio de validación de marcado** (MarkUp Validation Service), y el servicio de validación es capaz de validar la sintaxis de documentos en XHTML. Todos los ejemplos de XHTML en este libro se validaron con éxito utilizando validator.w3.org.

Al hacer clic en **Examinar...**, los usuarios pueden seleccionar archivos en sus propias computadoras para enviarlos. Después de seleccionar un archivo, al hacer clic en el botón de **Validate this file** (Validar este archivo) el archivo se envía y se valida. La figura F.3 muestra el resultado de la validación del archivo `principal.html`. Este documento no contiene errores de sintaxis. Si un documento contiene errores de sintaxis, entonces el servicio de validación presenta mensajes de error, en los que describe dichos errores.



Figura F.2 | Validación de un documento en XHTML. [Cortesía del Consorcio World Wide Web (W3C)].

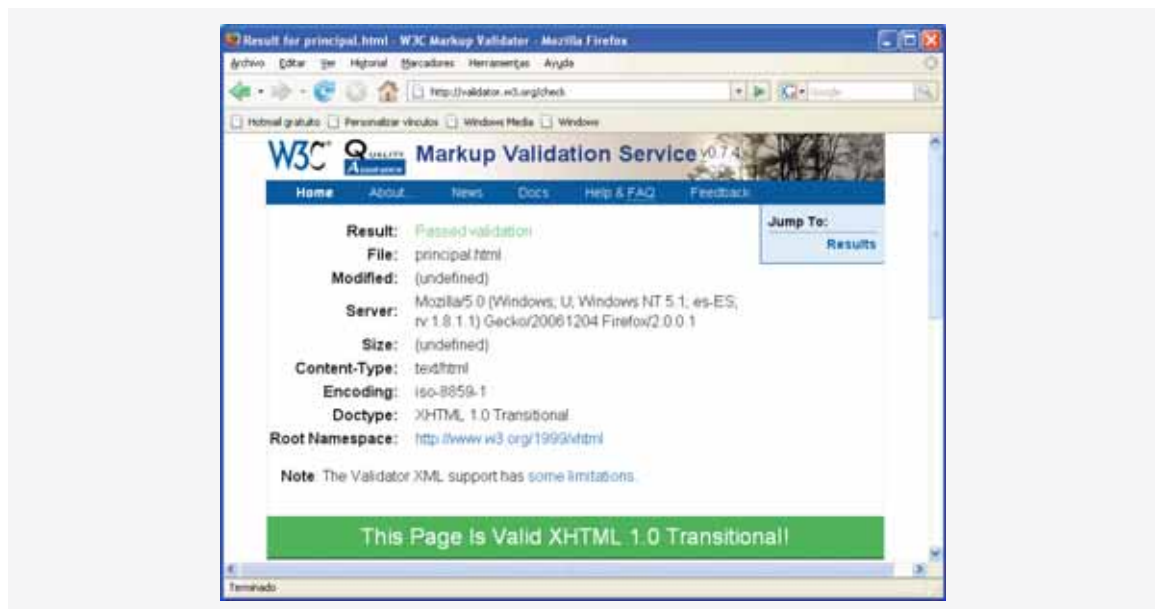


Figura F.3 | Resultados de la validación de un documento en XHTML. [Cortesía del Consorcio World Wide Web (W3C)].



Tip de prevención de errores F.1

La mayoría de los exploradores Web actuales intentan representar los documentos en XHTML, aun cuando éstos sean inválidos. A menudo, esto produce resultados inesperados y tal vez no deseados. Utilice un servicio de validación, tal como el servicio de validación de marcado de W3C, para confirmar que un documento en XHTML esté sintácticamente correcto.

F.5 Encabezados

Cierto texto dentro de un documento XHTML puede ser más importante que algún otro texto. Por ejemplo, el texto en esta sección se considera más importante que una nota al pie de página. El lenguaje XHTML proporciona seis *encabezados*, llamados *elementos de encabezado*, para especificar la importancia relativa de la información. La figura F.4 demuestra estos elementos (h1 a h6). El elemento de encabezado h1 (línea 15) se considera el más significativo y, por lo general, se presenta en un tipo de letra más grande que los otros cinco encabezados (líneas 16-20). Cada elemento de encabezado sucesivo (en otras palabras, h2, h3, etc.) se presenta en un tipo de letra cada vez más pequeño.



Tip de portabilidad F.1

El tamaño del texto utilizado para presentar cada elemento de encabezado puede variar de manera considerable entre los exploradores Web disponibles.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.4: encabezado.html -->
6  <!-- Encabezados en XHTML -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Encabezados</title>
11    </head>
12
13    <body>
14
15     <h1>Encabezado de nivel 1</h1>
16     <h2>Encabezado de nivel 2</h2>
17     <h3>Encabezado de nivel 3</h3>
18     <h4>Encabezado de nivel 4</h4>
19     <h5>Encabezado de nivel 5</h5>
20     <h6>Encabezado de nivel 6</h6>
21
22    </body>
23  </html>

```

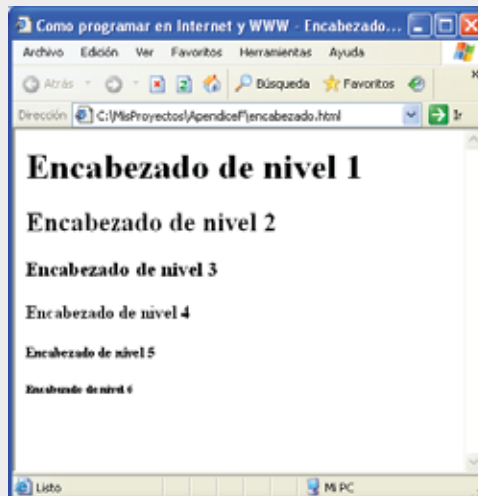


Figura F.4 | Elementos de encabezado h1 a h6.



Observación acerca de la presentación F.1

Si colocamos un encabezado en la parte superior de cada página en XHTML, ayudamos a que los usuarios comprendan el propósito de cada página.



Observación acerca de la presentación F.2

Utilice encabezados más grandes para enfatizar las secciones más importantes de una página Web.

F.6 Vínculos

Una de las características más importantes del lenguaje XHTML es el *hipervínculo*, el cual hace referencia (o *vincula*) a otros recursos, como otros documentos en XHTML o imágenes. En XHTML, tanto el texto como las imágenes pueden funcionar como hipervínculos. Por lo general, los exploradores Web subrayan los hipervínculos de texto y colorean su texto en azul de manera predeterminada, para que los usuarios puedan distinguir los hipervínculos del texto normal. En la figura F.5 creamos hipervínculos de texto a cuatro sitios Web distintos.

La línea 17 introduce el elemento **strong**. Por lo general, los exploradores representan este texto en negritas. Los vínculos se crean mediante el *elemento a (ancla)*. La línea 20 define un hipervínculo que vincula el texto Deitel con la dirección asignada al atributo **href**, el cual especifica la ubicación de un recurso vinculado, como una página Web, un archivo o una dirección de correo electrónico. Este elemento de ancla específico está vinculado a una página Web ubicada en <http://www.deitel.com>. Cuando una dirección Web no indica un documento específico en el sitio Web, el servidor Web devuelve una página Web predeterminada. A esta página se le conoce a menudo como `index.html`; sin embargo, la mayoría de los servidores Web pueden configurarse para utilizar cualquier archivo como la página Web predeterminada para el sitio. (Abra el sitio <http://www.deitel.com> en una ventana del explorador y <http://www.deitel.com/indice.html> en una segunda ventana

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.5: vinculos.html -->
6  <!-- Introducción a los hipervínculos -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Cómo programar en Internet y WWW - Vínculos</title>
11    </head>
12
13    <body>
14
15     <h1>Aquí estan mis sitios favoritos</h1>
16
17     <p><strong>Haga clic en uno de los nombres para ir a esa pagina.</strong></p>
18
19     <!-- Crea cuatro hipervínculos de texto -->
20     <p><a href = "http://www.deitel.com">Deitel</a></p>
21
22     <p><a href = "http://www.prenhall.com">Prentice Hall</a></p>
23
24     <p><a href = "http://www.yahoo.com">Yahoo!</a></p>
25
26     <p><a href = "http://www.usatoday.com">USA Today</a></p>
27
28    </body>
29  </html>

```

Figura F.5 | Creación de vínculos a otras páginas Web. (Parte I de 2).

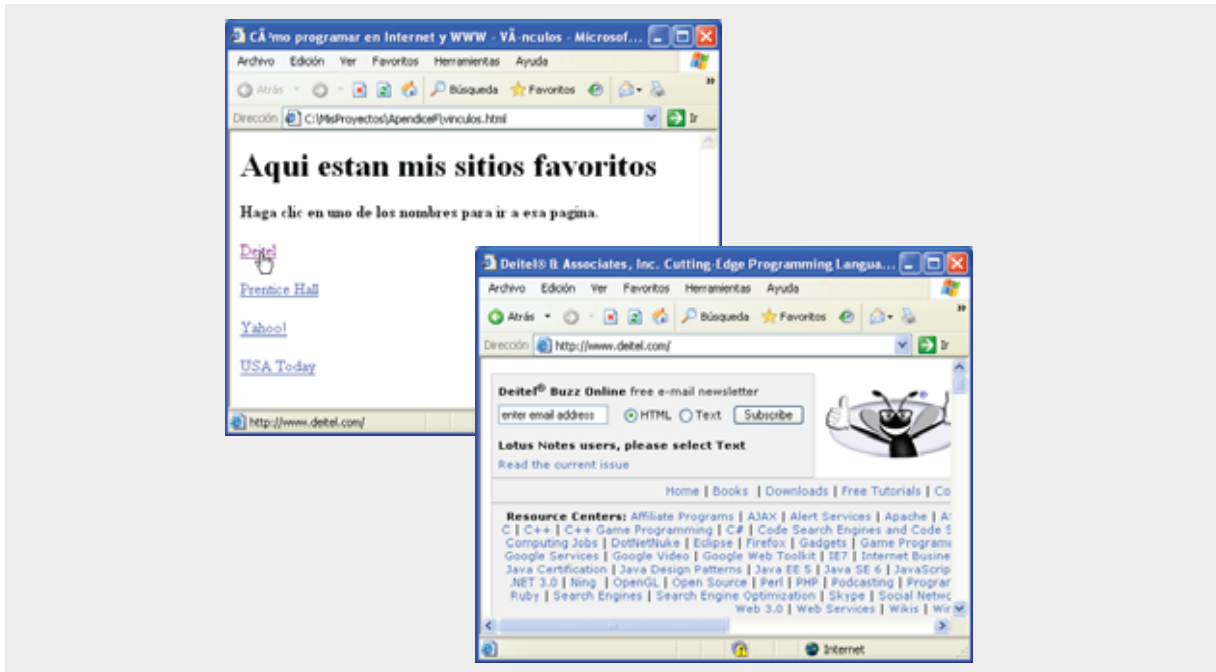


Figura F.5 | Creación de vínculos a otras páginas Web. (Parte 2 de 2).

del explorador, para confirmar que son idénticas.) Si un servidor Web no puede localizar un documento solicitado, devuelve una indicación de error al explorador Web, y éste muestra una página Web que contiene un mensaje de error para el usuario.

Las anclas pueden crear vínculos a direcciones de correo electrónico, utilizando un URL *mailto:*. Cuando alguien hace clic en este tipo de vínculo anclado, la mayoría de los exploradores ejecutan el programa para correo electrónico predeterminado (por ejemplo, Outlook Express), para permitir al usuario que escriba un mensaje de correo electrónico a la dirección vinculada. La figura F.6 muestra este tipo de ancla. Las líneas 17-19 contienen un vínculo a un correo electrónico. La forma del vínculo de correo electrónico es `<a href = "mailto:direccion_de_correo">...</a>`. En este caso estamos creando un vínculo a la dirección de correo electrónico `deitel@deitel.com`.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.6: contacto.html -->
6  <!-- Agregando hiperligas de correo electrónico -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Pagina de Contactos</title>
11    </head>
12
13    <body>
14
15     <p>
16       Mi direccion de correo electronico es
17       <a href = "mailto:deitel@ deitel.com">
18         deitel@deitel.com

```

Figura F.6 | Vínculo a una dirección de correo electrónico. (Parte 1 de 2).


```

19      </a>
20      . Haga clic en la dirección y su explorador
21      abra un mensaje de correo electrónico y me lo enviara a mi.
22      </p>
23      </body>
24      </html>

```

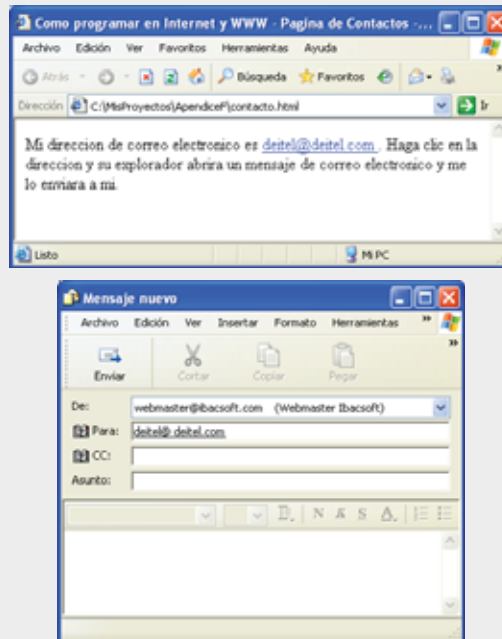


Figura F.6 | Vínculo a una dirección de correo electrónico. (Parte 2 de 2).

F.7 Imágenes

Los ejemplos que hemos visto hasta ahora demuestran cómo marcar documentos que contienen sólo texto. Sin embargo, la mayoría de las páginas Web contienen tanto texto como imágenes. De hecho, las imágenes son una parte equivalente, sino es que esencial, del diseño de una página Web. Los tres formatos para imágenes más populares utilizados por los desarrolladores Web son el Formato de intercambio de gráficos (GIF), el formato del Grupo unido de expertos en fotografía (JPEG) y las imágenes de Gráficos de red portables (PNG). Los usuarios pueden crear imágenes utilizando aplicaciones de software especializadas, tales como Photoshop Elements 2.0 de Adobe (www.adobe.com), Fireworks de Macromedia (www.macromedia.com) y Paint Shop Pro de Jasc (www.jasc.com). También se pueden adquirir imágenes de varios sitios Web tales como la galería de imágenes de Yahoo! (gallery.yahoo.com). La figura F.7 demuestra cómo incorporar imágenes a las páginas Web.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3      "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.7: imagen.html -->
6  <!-- Agregar imágenes con XHTML -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9      <head>
10         <title>Como programar en Internet y WWW - Bienvenidos</title>

```

Figura F.7 | Imágenes en archivos de XHTML. (Parte 1 de 2).

```

11     </head>
12
13     <body>
14
15         <p>
16             <img src = "xmlh1tp.jpg" height = "238" width = "183"
17                 alt = "Portada del libro Como programar en XML" />
18             <img src = "jh1tp.jpg" height = "238" width = "183"
19                 alt = "Portada del libro Como programar en Java" />
20         </p>
21     </body>
22 </html>

```



Figura F.7 | Imágenes en archivos de XHTML. (Parte 2 de 2).

Las líneas 16-17 utilizan el elemento *img* para insertar una imagen en el documento. La ubicación del archivo de la imagen se especifica mediante el atributo **src** del elemento *img*. En este caso, la imagen está ubicada en el mismo directorio que este documento XHTML, así que sólo se requiere el nombre de archivo de la imagen. Los atributos opcionales **width** y **height** especifican la anchura y altura de la imagen, respectivamente. El autor del documento puede escalar la imagen al incrementar o decrementar los valores de sus atributos **width** y **height**. Si se omiten estos atributos, el explorador utiliza la anchura y altura actual de la imagen. Las imágenes se miden en *píxeles* (“elementos de imagen”), los cuales representan puntos de color en la pantalla. La imagen en la figura F.7 tiene 183 píxeles de anchura y 238 píxeles de altura.



Buena práctica de programación F.5

Siempre debemos incluir la anchura (width) y altura (height) de una imagen dentro de la etiqueta . Cuando el explorador cargue el archivo XHTML, sabrá inmediatamente a través de estos atributos cuánto espacio en la pantalla debe proporcionar para la imagen y la colocará adecuadamente en la página incluso antes de que se haya descargado a la imagen.



Tip de rendimiento F.1

Si incluimos los atributos de anchura (width) y altura (height) en una etiqueta , el explorador podría cargar y representar las páginas con más rapidez.



Error común de programación F.4

Si introducimos nuevas medidas para una imagen, de manera que cambien su proporción natural de anchura-altura, se distorsionará la apariencia de la imagen. Por ejemplo, si la imagen es de 200 píxeles de anchura y 100 píxeles de altura, siempre hay que asegurarnos de que cualquier medida nueva que se utilice tenga una proporción de anchura-altura de 2:1.

Cada elemento `img` en un documento en XHTML posee un atributo **`alt`**. Si un explorador no puede presentar una imagen, presenta el valor del atributo `alt`. Tal vez el explorador no pueda presentar una imagen por distintas razones. Puede ser que no soporte las imágenes; tal es el caso de un *explorador basado en texto* (en otras palabras, un explorador que sólo puede mostrar texto), o tal vez el cliente haya deshabilitado la visualización de imágenes para reducir el tiempo de descarga. La figura F.7 muestra la forma en que Internet Explorer 6 presenta el valor del atributo `alt`, cuando un documento hace referencia a un archivo de imagen inexistente (`jhttp.jpg`).

El atributo `alt` es importante para la creación de páginas Web *accesibles* a usuarios con discapacidad, en especial para aquéllos con discapacidad visual que utilizan exploradores basados en texto. A menudo, la gente discapacitada utiliza un software especializado llamado *sintetizador de voz*. Esta aplicación de software “dice” el valor del atributo `alt` para que el usuario sepa lo que el explorador está presentando.

Algunos elementos de XHTML (llamados *elementos vacíos*) sólo contienen atributos y no poseen funciones para el marcado de texto (en otras palabras, el texto no está ubicado entre las etiquetas inicial y final). Los elementos vacíos (por ejemplo, `img`) deben terminarse ya sea utilizando el *carácter de barra diagonal* (/) dentro del signo mayor que (>) de la etiqueta inicial, o incluyendo la etiqueta final de manera explícita. Al utilizar el carácter de barra diagonal, agregamos un espacio antes del carácter para mejorar la legibilidad (como se muestra al final de las líneas 17 y 19). En vez de utilizar el carácter de barra diagonal, podríamos escribir las líneas 18-19 con una etiqueta `</img>` de cierre, como se muestra a continuación:

```
<img src = "jhttp.jpg" height = "238" width = "183"
alt = "Portada del libro Como programar en Java"></img>
```

Al utilizar imágenes como hipervínculos, los desarrolladores Web pueden crear páginas Web con gráficos que están vinculados a otros recursos. En la figura F.8 creamos seis diferentes hipervínculos de imágenes.

Las líneas 17-20 crean un *hipervínculo de imagen* al anidar un elemento `img` en un elemento de ancla (a). El valor del atributo `src` del elemento `img` especifica que esta imagen (`vinculos.jpg`) está ubicada en un directorio llamado `botones`. El directorio `botones` y el documento en XHTML se encuentran en el mismo directorio. También se puede hacer referencia a las imágenes de otros documentos Web (después de obtener permiso del dueño del documento), asignando el nombre y la ubicación de la imagen al atributo `src`. Al hacer clic en un hipervínculo el usuario es llevado a la página Web especificada por el atributo `href` del elemento de ancla que enmarca al vínculo.

En la línea 20 introducimos el *elemento br*, el cual la mayoría de los exploradores presentan como un *salto de línea*. Cualquier marcado o texto que siga después de un elemento `br` se presenta en la siguiente línea. Al igual que el elemento `img`, `br` es un ejemplo de un elemento vacío terminado por un carácter de barra diagonal. Hemos agregado un espacio antes del carácter de barra diagonal para mejorar la legibilidad. [Nota: los últimos dos hipervínculos de imágenes en la figura F.8 vinculan a documentos en XHTML (en otras palabras, `tabla1.html` y `formulario.html`) que presentamos como ejemplos en el apéndice G, y se incluyen en el directorio de ejemplos del apéndice G. Si hace clic en estos vínculos ahora se producirán errores.]

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.8: nav.html -->
6  <!-- Uso de imágenes como anclas de vínculos -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Barra de navegación
11     </title>
12   </head>
13
14   <body>
15
16     <p>
```

Figura F.8 | Imágenes como anclas de vínculos. (Parte I de 2).

```

17 <a href = "vinculos.html">
18   <img src = "botones/vinculos.jpg" width = "65"
19     height = "50" alt = "Pagina de vinculos" />
20 </a><br />
21
22 <a href = "lista.html">
23   <img src = "botones/lista.jpg" width = "65"
24     height = "50" alt = "Pagina de ejemplo de una lista" />
25 </a><br />
26
27 <a href = "contacto.html">
28   <img src = "botones/contacto.jpg" width = "65"
29     height = "50" alt = "Pagina de contacto" />
30 </a><br />
31
32 <a href = "encabezado.html">
33   <img src = "botones/encabezado.jpg" width = "65"
34     height = "50" alt = "Pagina de encabezado" />
35 </a><br />
36
37 <a href = "tabla1.html">
38   <img src = "botones/tabla.jpg" width = "65"
39     height = "50" alt = "Pagina de tabla" />
40 </a><br />
41
42 <a href = "formulario.html">
43   <img src = "botones/formulario.jpg" width = "65"
44     height = "50" alt = "Formulario de retroalimentacion" />
45 </a><br />
46 </p>
47
48 </body>
49 </html>

```

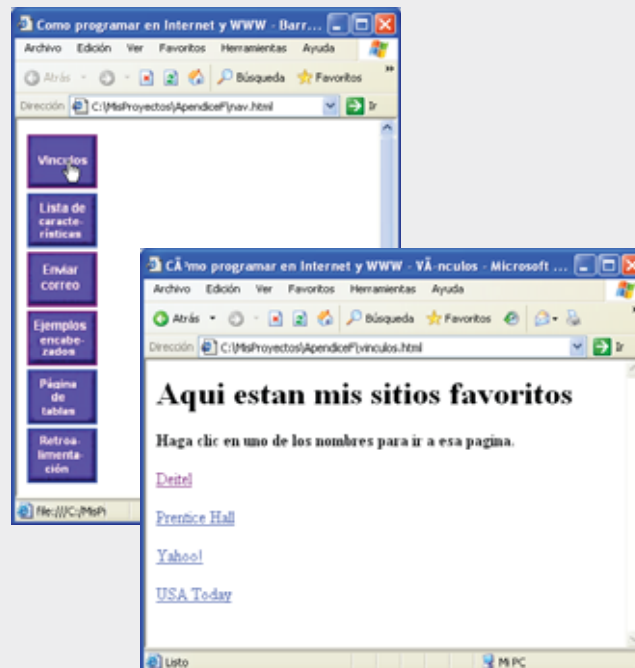


Figura F.8 | Imágenes como anclas de vínculos. (Parte 2 de 2).

F.8 Caracteres especiales y más saltos de línea

Cuando marcamos texto, ciertos caracteres o símbolos (por ejemplo, <) pueden ser difíciles de insertar directamente en un documento en XHTML. Algunos teclados no proporcionan estos símbolos o la presencia de estos símbolos puede provocar errores de sintaxis. Por ejemplo, el marcado

```
<p>si x < 10 entonces incrementa x en 1</p>
```

produce un error de sintaxis, porque utiliza el carácter menor que (<), el cual se reserva para las etiquetas inicial y final, tales como <p> y </p>. El lenguaje XHTML proporciona *referencias a una entidad de carácter* (de la forma &código;) para representar caracteres especiales. Podemos entonces corregir la línea anterior si escribimos

```
<p>si x &lt; 10 entonces incrementa x por 1</p>
```

en la cual se utiliza la referencia a una entidad de carácter < para el símbolo menor que.

La figura F.9 demuestra cómo utilizar caracteres especiales en un documento en XHTML. En el apéndice H, Caracteres especiales en HTML/XHTML, podrá ver una lista de los caracteres especiales.

Las líneas 27-28 contienen otros caracteres especiales, los cuales se pueden expresar ya sea como referencias a entidades de caracteres (en otras palabras, abreviaciones tales como amp para el símbolo & y copy para copyright, o derechos reservados) o *referencias a caracteres numéricos*: valores decimales o *hexadecimales (hex)* que representan caracteres especiales. Por ejemplo, el carácter & se representa en notación decimal y hexadecimal como & y &; respectivamente. Los números en hexadecimal son números en base 16; los dígitos en un número hexadecimal tienen valores entre 0 y 15 (un total de 16 valores diferentes). Las letras A-F representan a los dígitos hexadecimales correspondientes a los valores decimales 10-15. Por lo tanto, en notación hexadecimal podemos tener números como 876, que consisten solamente de dígitos similares a decimales, números tales como DA19F, que consisten de dígitos y letras, y números tales como DCB, que consisten sólo de letras. En el apéndice B, Sistemas numéricos, veremos los números hexadecimales con mayor detalle.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.9: contacto2.html -->
6  <!-- Inserción de caracteres especiales -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Pagina de contacto
11     </title>
12   </head>
13
14   <body>
15
16     <!-- los caracteres especiales se agregan -->
17     <!-- utilizando la forma &código; -->
18     <p>
19       Haga clic
20       <a href = "mailto:deitel@deitel.com">aqui</a>
21       para abrir un mensaje de correo electronico dirigido a
22       deitel@deitel.com
23     </p>
24
25     <hr /> <!-- agrega una regla horizontal -->
26
27     <p>Toda la informacion en este sitio es <strong>&copy;</strong>
28     Deitel <strong>&amp;</strong> Associates, Inc. 2004. </p>
```

Figura F.9 | Caracteres especiales en XHTML. (Parte I de 2).

```

29
30 <!-- para texto tachado utilice las etiquetas <del> -->
31 <!-- para texto de subíndice utilice las etiquetas <sub> -->
32 <!-- para texto de superíndice utilice las etiquetas <sup> -->
33 <!-- estas etiquetas están anidadas dentro de otras etiquetas -->
34 <p><del>Puede descargar 3.14 x 10<sup>2</sup></del>
35 caracteres en total de información de este sitio.</del>
36 Solo se permite <sub>una</sub> descarga por hora.</p>
37
38 <p>Nota: <strong>&lt; &frac14; </strong> de la información
39 aquí presentada se actualiza diariamente. </p>
40
41 </body>
42 </html>

```

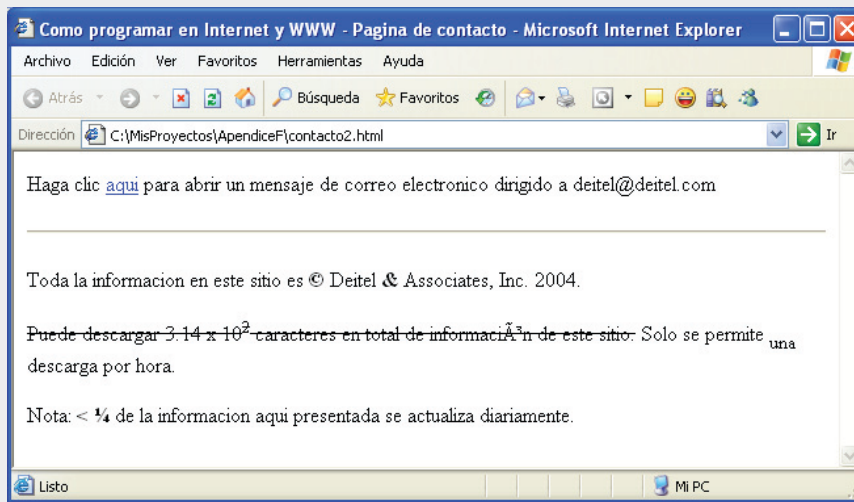


Figura F.9 | Caracteres especiales en XHTML. (Parte 2 de 2).

En las líneas 34-36 introducimos tres nuevos elementos. La mayoría de los exploradores Web presentan el elemento *del* como texto tachado. Mediante el uso de este formato, los usuarios pueden indicar con facilidad las revisiones a los documentos. Para convertir texto a *superíndice* (en otras palabras, elevar el texto en una línea con un tamaño de letra más pequeño) o para convertir texto a *subíndice* (en otras palabras, reducir el texto en una línea con un tamaño de letra más pequeño), utilizamos el elemento *sup* o *sub*, respectivamente. También utilizamos la referencia a una entidad de carácter *<*; para el signo menor que y *¼* para la fracción 1/4 (línea 38).

Además de los caracteres especiales, este documento introduce una *regla horizontal*, que se indica mediante la etiqueta *<hr />* en la línea 25. La mayoría de los exploradores Web presentan una regla horizontal como una línea horizontal. La etiqueta *<hr />* también inserta un salto de línea por encima y por debajo de la línea horizontal.

F.9 Listas desordenadas

Hasta ahora hemos presentado elementos y atributos básicos en XHTML para crear vínculos a recursos, crear encabezados, utilizar caracteres especiales e incorporar imágenes. En esta sección hablaremos sobre cómo organizar la información en una página Web mediante el uso de listas. La figura F.10 presenta texto en una *lista desordenada* (es decir, una lista que no ordena sus elementos por letra o por número). El elemento *ul* para una lista desordenada crea una lista en donde cada elemento empieza con un símbolo de viñeta (también conocido como *disco*). Cada elemento en una lista desordenada (el elemento *ul* en la línea 20) es un elemento *li* (*elemento de lista*), como en las líneas 23, 25, 27 y 29. La mayoría de los exploradores Web representan estos elementos con un salto de línea y un símbolo de viñeta con sangría desde el principio de una nueva línea.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.10: vinculos2.html -->
6  <!-- Listas desordenadas con hipervínculos -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Vinculos</title>
11    </head>
12
13    <body>
14
15     <h1>Aqui estan mis sitios favoritos</h1>
16
17     <p><strong>Haga clic en uno de los nombres para visitar esa pagina.</strong></p>
18
19     <!-- crear una lista desordenada -->
20     <ul>
21
22         <!-- agrega cuatro elementos a la lista -->
23         <li><a href = "http://www.deitel.com">Deitel</a></li>
24
25         <li><a href = "http://www.w3.org">W3C</a></li>
26
27         <li><a href = "http://www.yahoo.com">Yahoo</a></li>
28
29         <li><a href = "http://www.cnn.com">CNN</a></li>
30     </ul>
31 </body>
32 </html>

```

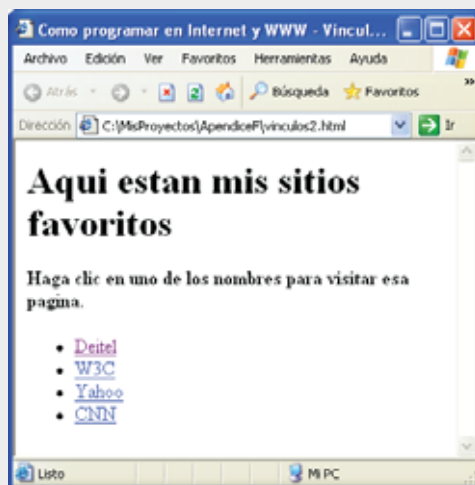


Figura F.10 | Listas desordenadas en XHTML.

F.10 Listas anidadas y ordenadas

Las listas pueden anidarse para representar relaciones jerárquicas, por ejemplo en un formato de bosquejo. La figura F.11 demuestra las listas anidadas y las *listas ordenadas*. El elemento **ol** de la lista ordenada crea una lista, en donde cada elemento empieza con un número.

Un explorador Web aplica sangría a cada lista anidada para indicar una relación jerárquica. La primera lista ordenada empieza en la línea 33. Los elementos en una lista ordenada se enumeran como uno, dos, tres y

así sucesivamente. Las listas ordenadas anidadas se enumeran de la misma manera. Los elementos en la lista desordenada más externa (línea 18) están precedidos por discos. Los elementos de lista anidados dentro de la lista desordenada de la línea 18 están precedidos por *círculos*. Aunque no lo demostramos en este ejemplo, los elementos subsiguientes de una lista anidada están precedidos por *cuadrados*.

```

1  <?xml versión = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Figura F.11: lista.html -->
6  <!-- Listas avanzadas: anidadas y ordenadas -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Listas</title>
11    </head>
12
13    <body>
14
15     <h1>Las mejores características de Internet</h1>
16
17     <!-- crea una lista desordenada -->
18     <ul>
19       <li>Podemos conocer gente de todos los países alrededor del
20         mundo.</li>
21       <li>
22         Tenemos acceso a nuevos medios electrónicos, conforme se introducen al público:
23
24         <!-- esto inicia una lista anidada, la cual utiliza una -->
25         <!-- viñeta modificada. La lista termina cuando -->
26         <!-- cerramos la etiqueta <ul>. -->
27         <ul>
28           <li>Juegos nuevos</li>
29           <li>
30             Nuevas aplicaciones</li>
31
32           <!-- lista ordenada anidada -->
33           <ol>
34             <li>De negocios</li>
35             <li>De entretenimiento</li>
36           </ol>
37         </li>
38
39         <li>Noticias durante el día</li>
40         <li>Algoritmos de búsqueda</li>
41         <li>Compras</li>
42         <li>
43           Programación en
44
45           <!-- otra lista ordenada y anidada -->
46           <ol>
47             <li>XML</li>
48             <li>Java</li>
49             <li>XHTML</li>
50             <li>Secuencias de comandos</li>
51             <li>Nuevos lenguajes</li>
52           </ol>
53         </li>

```

Figura F.11 | Listas anidadas y ordenadas en XHTML. (Parte 1 de 2).


```

54         </li>
55
56     </ul> <!-- termina la lista anidada de la línea 27 -->
57 </li>
58
59 <li>Vínculos</li>
60 <li>Mantener el contacto con viejas amistades</li>
61 <li>Es la tecnología del futuro!</li>
62
63 </ul> <!-- termina la lista sin orden de la línea 18 -->
64
65 </body>
66 </html>

```



Figura F.11 | Listas anidadas y ordenadas en XHTML. (Parte 2 de 2).

F.11 Recursos Web

www.w3.org/TR/xhtml11

La *Recomendación para XHTML 1.1* contiene información general acerca del lenguaje XHTML 1.1, además de cuestiones sobre compatibilidad, información acerca de la definición del tipo de documento, definiciones, terminología y demás.

www.xhtml.org

XHTML.org proporciona noticias acerca del desarrollo en XHTML y vínculos a otros recursos para XHTML incluyendo libros y artículos.

www.w3schools.com/xhtml/default.asp

La *Escuela de XHTML* cuenta con exámenes y referencias para XHTML. Esta página también contiene vínculos a la sintaxis, validación y definiciones del tipo de documento en XHTML.

validator.w3.org

Éste es el sitio para el servicio de validación de XHTML de W3C.

hotwired.lycos.com/webmonkey/00/50/index2a.html

Este sitio presenta un artículo acerca del XHTML. Las secciones clave en el artículo repasan las generalidades acerca del XHTML; además cuenta con discusiones sobre las etiquetas, atributos y anclas.

wdvl.com/Authoring/Languages/XML/XHTML

La *Biblioteca virtual para desarrolladores Web* proporciona una introducción al lenguaje XHTML. Este sitio contiene también artículos, ejemplos y vínculos a otras tecnologías.

www.w3.org/TR/2001/REC-xhtml11-20010531

El sitio de la documentación de la DTD de XHTML 1.1 proporciona especificaciones técnicas de la sintaxis para XHTML 1.1.



Introducción a XHTML: parte 2

*Así es, de la tabla
de mi memoria
borraré todos los registros
triviales y apasionados.*

—William Shakespeare

OBJETIVOS

En este apéndice aprenderá lo siguiente:

- Crear tablas con filas y columnas de datos.
- Controlar el formato de las tablas.
- Crear y utilizar formularios.
- Crear y utilizar mapas de imágenes para ayudar en la navegación de páginas Web.
- Crear páginas Web accesibles para los motores de búsqueda, mediante el uso de las etiquetas `<meta>`.
- Utilizar el elemento `frameset` para mostrar varias páginas Web en una sola ventana del explorador.

- G.1 Introducción
- G.2 Tablas básicas en XHTML
- G.3 Tablas intermedias en XHTML y formatos
- G.4 Formularios básicos en XHTML
- G.5 Formularios más complejos en XHTML
- G.6 Vínculos internos
- G.7 Creación y uso de mapas de imágenes
- G.8 Elementos meta
- G.9 Elemento frameset
- G.10 Elementos frameset anidados
- G.11 Recursos Web

G.1 Introducción

En el apéndice anterior presentamos el XHTML. Creamos varias páginas Web completas con texto, hipervínculos, imágenes, reglas horizontales y saltos de línea. En este apéndice hablaremos sobre las características más complejas de XHTML, incluyendo la presentación de la información en *tablas* e incorporando *formularios* para recolectar la información de un visitante de la página Web. También presentaremos los *vínculos internos* y los *mapas de imágenes* para mejorar la navegación por las páginas Web, y los *marcos* para mostrar varios documentos en el explorador. Al final de este apéndice estará familiarizado con las características de más uso común de XHTML, y podrá crear documentos Web más complejos.

G.2 Tablas básicas en XHTML

Las tablas se utilizan para organizar los datos en filas y columnas. Nuestro primer ejemplo (figura G.1) crea una tabla con seis filas y dos columnas, para mostrar la información de las frutas.

Las tablas se definen con el elemento **table** (líneas 16-66). Las líneas 16-18 especifican la marca inicial para un elemento **table** que tiene varios atributos. El atributo **border** especifica la anchura del borde de la tabla en píxeles. Para crear una tabla sin un borde, establezca **border** en "0". Este ejemplo asigna al atributo **width** el valor "40%", para establecer la anchura de la tabla al 40 por ciento de la anchura del explorador. Un desarrollador también puede establecer el atributo **width** a un número especificado de píxeles. Trate de cambiar el tamaño de la ventana del explorador, para ver cómo la anchura de la ventana afecta a la anchura de la tabla.

Como su nombre lo implica, el atributo **summary** (líneas 17-18) describe el contenido de una tabla. Los dispositivos de voz utilizan este atributo para que la tabla sea más accesible para los usuarios con discapacidad visual. El elemento **caption** (línea 22) describe el contenido de la tabla y ayuda a que los exploradores basados en texto interpreten los datos de la tabla. La mayoría de los exploradores presentan el texto dentro de la etiqueta **<caption>** por encima de la tabla. El atributo **summary** y el elemento **caption** son dos de las muchas características de XHTML que hacen las páginas Web más accesibles para los usuarios con discapacidad.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. G.1: tabla1.html -->
6  <!-- Creación de una tabla básica -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10   <title>Una tabla simple en XHTML</title>

```

Figura G.1 | Tabla de XHTML. (Parte 1 de 3).

```

11  </head>
12
13  <body>
14
15      <!-- la etiqueta <table> abre una tabla -->
16      <table border = "1" width = "40%"
17          summary = "Esta tabla proporciona información acerca
18              del precio de la fruta">
19
20          <!-- la etiqueta <caption> resume el contenido de la tabla -->
21          <!-- (esto ayuda a las personas con discapacidad visual) -->
22          <caption><strong>Precio de la fruta</strong></caption>
23
24          <!-- la etiqueta <thead> es la primera sección de una tabla -->
25          <!-- da formato al área de encabezados de la tabla -->
26          <thead>
27              <tr>                                <!-- <tr> inserta una fila en la tabla -->
28                  <th>Fruta</th>                <!-- inserta una celda de encabezado -->
29                  <th>Precio</th>
30              </tr>
31          </thead>
32
33          <!-- la etiqueta <tfoot> es la última sección de una tabla -->
34          <!-- da formato al pie de la tabla -->
35          <tfoot>
36              <tr>
37                  <th>Total</th>
38                  <th>$3.75</th>
39              </tr>
40          </tfoot>
41
42          <!-- todo el contenido de la tabla va encerrado -->
43          <!-- dentro de la etiqueta <tbody> -->
44          <tbody>
45              <tr>
46                  <td>Manzana</td> <!-- inserta una celda de datos -->
47                  <td>$0.25</td>
48              </tr>
49
50              <tr>
51                  <td>Naranja</td>
52                  <td>$0.50</td>
53              </tr>
54
55              <tr>
56                  <td>Platano</td>
57                  <td>$1.00</td>
58              </tr>
59
60              <tr>
61                  <td>Coco</td>
62                  <td>$2.00</td>
63              </tr>
64          </tbody>
65
66      </table>
67
68  </body>
69  </html>

```

Figura G.1 | Tabla de XHTML. (Parte 2 de 3).

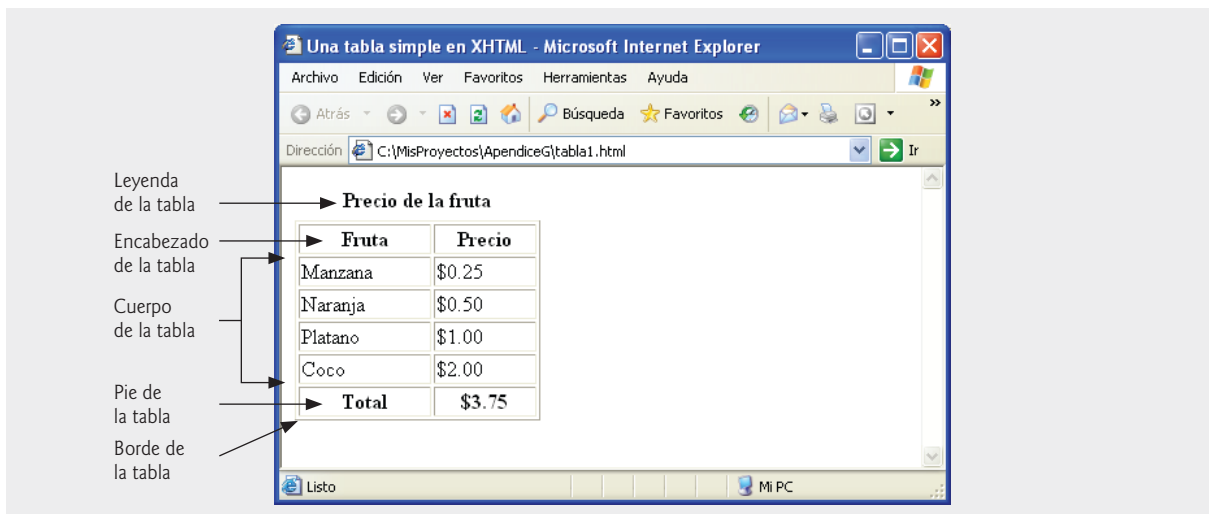


Figura G.1 | Tabla de XHTML. (Parte 3 de 3).

Una tabla tiene tres secciones distintas: *encabezado*, *cuerpo* y *pie*. La sección (o celda) de encabezado se define mediante un elemento **thead** (líneas 26-31), el cual contiene información correspondiente al encabezado, como los nombres de las columnas. Cada elemento **tr** (líneas 27-30) define una sola *fila de la tabla*. Las columnas en la sección de encabezado se definen mediante los elementos **th**. La mayoría de los exploradores centran el texto que tiene el formato de los elementos **th** (columna de encabezado de la tabla) y lo muestran en negritas. Los elementos de encabezado de una tabla se anidan dentro de los elementos de fila.

La sección del pie (líneas 35-40) se define mediante un elemento **tfoot** (pie de la tabla). Por lo general, el texto que se coloca en el pie incluye los resultados de cálculos y las notas al pie. Al igual que las demás secciones, el pie de la tabla puede contener filas, y cada fila puede contener columnas.

La sección del cuerpo, o *cuerpo de la tabla*, contiene los datos principales de la misma. El cuerpo de la tabla (líneas 44-64) se define en un elemento **tbody**. En el cuerpo, cada elemento **tr** especifica una fila. Las *celdas de datos* contienen piezas de datos individuales, y se definen con elementos **td** (*datos de tabla*) dentro de cada fila.

G.3 Tablas intermedias en XHTML y formatos

En la sección anterior exploramos la estructura de una tabla básica. En la figura G.2 profundizaremos más en nuestra discusión sobre las tablas, mediante la introducción de elementos y atributos que permiten al autor de documentos crear tablas más complejas.

La tabla empieza en la línea 17. El elemento **colgroup** (líneas 22-27) agrupa y da formato a las columnas. El elemento **col** (línea 26) especifica dos atributos en este ejemplo. El atributo **align** determina la alineación del texto en la columna. El atributo **span** determina a cuántas columnas va a dar formato el elemento **col**. En este caso, establecemos el valor de **align** en "right" y el valor de **span** en "1" para alinear el texto a la derecha, en la primera columna (la que contiene la imagen del camello en la captura de pantalla de ejemplo).

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. G.2: tabla2.html -->
6  <!-- Diseño de una tabla intermedia -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
```

Figura G.2 | Tabla de XHTML compleja. (Parte 1 de 3).

```

9      <head>
10      <title>Como programar en Internet y WWW - Tablas</title>
11      </head>
12
13      <body>
14
15          <h1>Pagina de ejemplo de tablas</h1>
16
17          <table border = "1">
18              <caption>He aqui una tabla de ejemplo mas compleja.</caption>
19
20              <!-- las etiquetas <colgroup> y <col> se usan -->
21              <!-- para dar formato a columnas completas -->
22              <colgroup>
23
24                  <!-- el atributo span determina a cuántas -->
25                  <!-- columnas afecta la etiqueta <col> -->
26                  <col align = "right" span = "1" />
27              </colgroup>
28
29              <thead>
30
31                  <!-- los atributos rowspans y colspans combinan el número -->
32                  <!-- especificado de celdas en forma horizontal o vertical -->
33                  <tr>
34
35                      <!-- combina dos filas -->
36                      <th rowspan = "2">
37                          <img src = "camello.gif" width = "205"
38                              height = "167" alt = "Imagen de un camello" />
39                      </th>
40
41                      <!-- combina cuatro columnas -->
42                      <th colspan = "4" valign = "top">
43                          <h1>Comparacion de camelidos</h1><br />
44                          <p>Fecha aproximada: 9/2002</p>
45                      </th>
46                  </tr>
47
48                  <tr valign = "bottom">
49                      <th># de jorobas</th>
50                      <th>Region indigena</th>
51                      <th>Escape?</th>
52                      <th>Produce lana?</th>
53                  </tr>
54
55              </thead>
56
57              <tbody>
58
59                  <tr>
60                      <th>Camellos (bactrian)</th>
61                      <td>2</td>
62                      <td>Africa/Asia</td>
63                      <td>Si</td>
64                      <td>Si</td>
65                  </tr>
66
67                  <tr>

```

Figura G.2 | Tabla de XHTML compleja. (Parte 2 de 3).

```

68         <th>Llamas</th>
69         <td>1</td>
70         <td>Montañas de los Andes</td>
71         <td>Si</td>
72         <td>Si</td>
73     </tr>
74 </tbody>
75 </table>
76
77 </table>
78
79 </body>
80 </html>

```

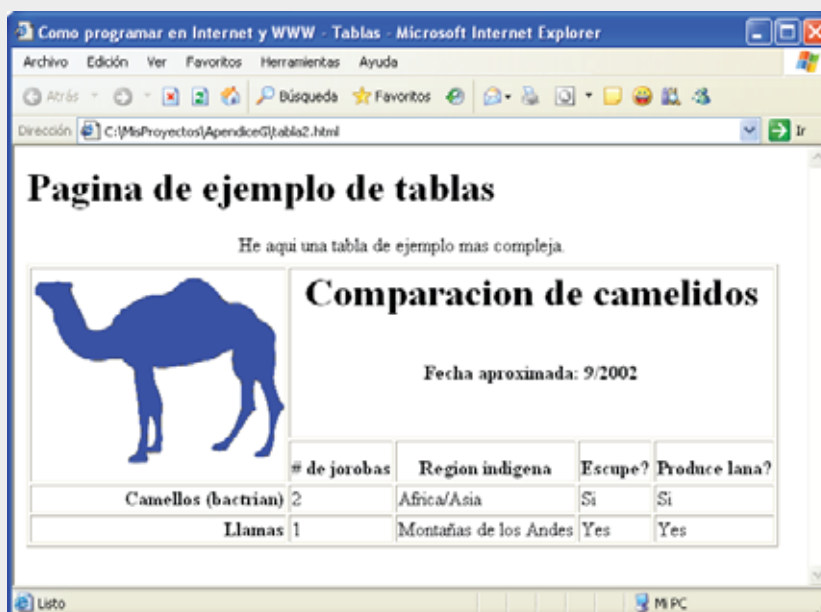


Figura G.2 | Tabla de XHTML compleja. (Parte 3 de 3).

El tamaño de las celdas de la tabla se cambia para ajustarse a los datos que contienen. Los autores de documentos pueden crear celdas de datos más grandes mediante el uso de los atributos **rowspan** y **colspan**. Los valores que se asignan a estos atributos especifican el número de filas o columnas que ocupa una celda. El elemento **th** en las líneas 36-39 utiliza el atributo **rowspan = "2"** para permitir que la celda que contiene la imagen del camello utilice dos celdas verticales adyacentes (por lo cual, la celda abarca dos filas). El elemento **th** en las líneas 42-45 utiliza el atributo **colspan = "4"** para ampliar la celda de encabezado (que contiene Comparacion de camelidos y Fecha aproximada: 9/2002), de manera que abarque cuatro celdas.



Error común de programación G.1

*Al utilizar **colspan** y **rowspan** para ajustar el tamaño de las celdas de datos de una tabla, tenga en cuenta que las celdas modificadas ocuparán más de una columna o fila. Otras filas o columnas de la tabla deben compensar las filas o columnas adicionales que ocupan las celdas individuales. Si no lo hacen, el formato de su tabla se distorsionará y podría crear de manera involuntaria más columnas y filas de lo que esperaba.*

La línea 42 introduce el atributo **valign**, el cual alinea los datos en forma vertical y puede recibir uno de cuatro valores: "top" alinea los datos con la parte superior de la celda, "middle" centra los datos en forma vertical (la opción predeterminada para todas las celdas de datos y de encabezado), "bottom" alinea los datos con la parte

inferior de la celda y "baseline" ignora las fuentes usadas para los datos de la fila y establece la parte inferior de todo el texto en la fila en una *línea de base* común (es decir, la línea horizontal, en la que se alinea cada carácter en una palabra).

G.4 Formularios básicos en XHTML

Al explorar sitios Web, los usuarios necesitan a menudo proporcionar información tal como palabras clave de búsqueda, direcciones de correo electrónico y códigos postales. XHTML cuenta con un mecanismo, llamado *formulario*, para recolectar tales datos de un usuario.

Por lo general, los datos que introducen los usuarios en una página Web se envían a un servidor Web, el cual proporciona acceso a los recursos de un sitio (por ejemplo, documentos en XHTML, imágenes). Estos recursos se ubican en el mismo equipo que el servidor Web, o en un equipo al que el servidor Web puede acceder a través de la red. Cuando un explorador solicita una página Web o un archivo que se encuentra en un servidor, éste procesa la solicitud y devuelve el recurso solicitado. Una solicitud contiene el nombre y la ruta del recurso deseado, junto con el método de comunicación (conocido como *protocolo*). Los documentos en XHTML utilizan el Protocolo de transferencia de hipertexto (HTTP).

La figura G.3 envía los datos del formulario al servidor Web, el cual pasa esos datos a una secuencia de comandos (es decir, un programa) *CGI (Interfaz de compuerta común)* escrita en Perl, C o en algún otro lenguaje. La secuencia de comandos procesa los datos recibidos del servidor Web y, por lo general, devuelve información al servidor Web. Después, el servidor Web envía la información como un documento de XHTML al explorador Web. [Nota: este ejemplo demuestra la funcionalidad del lado cliente. Si el formulario se envía (haciendo clic en **Enviar sus datos**) se produce un error, ya que no hemos configurado aún la funcionalidad requerida del lado servidor.]

Los formularios pueden contener componentes visuales y no visuales. Los componentes visuales incluyen botones en los que se puede hacer clic, y otros componentes de la interfaz gráfica de usuario con los que los usuarios interactúan. Los componentes no visuales, llamados *entradas ocultas*, almacenan los datos que especifica el autor del documento, como las direcciones de correo electrónico y los nombres de archivo de los documentos en XHTML que actúan como vínculos. El formulario se define en las líneas 23-52 mediante un elemento **form**. El atributo *method* (línea 23) especifica cómo se van a enviar los datos del formulario al servidor Web.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. G.3: formulario.html          -->
6  <!-- Ejemplo de diseño de formularios 1 -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Formularios</title>
11    </head>
12
13    <body>
14
15     <h1>Formulario de retroalimentacion</h1>
16
17     <p>Llene este formulario para ayudarnos
18       a mejorar nuestro sitio.</p>
19
20     <!-- esta etiqueta inicia el formulario, proporciona -->
21     <!-- el método para enviar información y la          -->
22     <!-- ubicación de las secuencias de comandos         -->
23     <form method = "post" action = "/cgi-bin/formmail">
24

```

Figura G.3 | Formulario con campos ocultos y un cuadro de texto. (Parte 1 de 2).


```

25     <p>
26         <!-- las entradas ocultas contienen -->
27         <!-- información no visual -->
28         <input type = "hidden" name = "recipient"
29             value = "deitel@deitel.com" />
30         <input type = "hidden" name = "subject"
31             value = "Formulario de retroalimentacion" />
32         <input type = "hidden" name = "redirect"
33             value = "principal.html" />
34     </p>
35
36     <!-- <input type = "text"> inserta un cuadro de texto -->
37     <p><label>Nombre:
38         <input name = "nombre" type = "text" size = "25"
39             maxlength = "30" />
40     </label></p>
41
42     <p>
43         <!-- los tipos de entrada "submit" y "reset" insertan -->
44         <!-- botones para enviar y borrar el contenido -->
45         <!-- del formulario -->
46         <input type = "submit" value =
47             "Enviar sus entradas" />
48         <input type = "reset" value =
49             "Borrar sus entradas" />
50     </p>
51
52 </form>
53
54 </body>
55 </html>

```

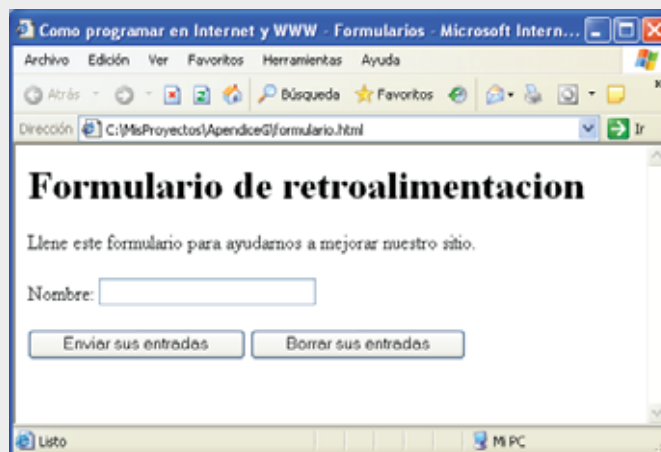


Figura G.3 | Formulario con campos ocultos y un cuadro de texto. (Parte 2 de 2).

Si utilizamos **method = "post"**, los datos del formulario se adjuntan a la solicitud del explorador, la cual contiene el protocolo (es decir, HTTP) y el URL del recurso solicitado. Las secuencias de comandos ubicadas en la computadora del servidor Web (o en una computadora accesible a través de la red) pueden acceder a los datos enviados del formulario como parte de la solicitud. Por ejemplo, una secuencia de comandos podría tomar la información del formulario y actualizar una lista de correo electrónico. El otro valor posible, **method = "get"**, adjunta los datos del formulario directamente al final del URL. Por ejemplo, al URL `/cgi-bin/formmail` se le podría adjuntar la información del formulario `nombre = bob`.

El atributo **action** en la etiqueta `<form>` especifica el URL de una secuencia de comandos en el servidor Web; en este caso, especifica una secuencia de comandos que envía los datos del formulario a una dirección de correo electrónico. La mayoría de los Proveedores de servicio de Internet (ISPs) tienen una secuencia de comandos como ésta en su sitio; pregunte al administrador del sitio Web acerca de cómo configurar un documento en XHTML para utilizar esta secuencia de comandos en forma correcta.

Las líneas 28-33 definen tres elementos `input` que especifican los datos que se van a proporcionar a la secuencia de comandos que procesa el formulario (a la cual también se le conoce como el *manejador del formulario*). Estos tres elementos `input` tienen el atributo **type** "hidden", el cual permite al autor del documento enviar los datos del formulario que el usuario no introduce.

Las tres entradas ocultas son: una dirección de correo electrónico a la que se enviarán los datos, la línea del asunto del correo electrónico y un URL al que se redirigirá el explorador después de enviar el formulario. Los otros dos atributos `input` son **name**, que identifica al elemento `input`, y **value**, que proporciona el valor que se enviará al (o se publicará en el) servidor Web.



Buena práctica de programación G.1

Coloque los elementos `input` ocultos al principio de un formulario, justo después de la etiqueta inicial `<form>`. Este posicionamiento permite a los autores de documentos localizar los elementos `input` ocultos con rapidez.

En las líneas 38-39 presentamos otro tipo de entrada. El elemento `input "text"` inserta un *cuadro de texto* en el formulario. Los usuarios pueden escribir datos en los cuadros de texto. El elemento **label** (líneas 37-40) proporciona a los usuarios información acerca del propósito del elemento `input`.



Observación de apariencia visual G.1

Incluya un elemento `label` para cada elemento del formulario, para ayudar a los usuarios a determinar el propósito de cada elemento del formulario.

El atributo **size** del elemento `input` especifica el número de caracteres visibles en el cuadro de texto. El atributo opcional **maxlength** limita el número de caracteres introducidos en el cuadro de texto. En este caso, al usuario no se le permite escribir más de 30 caracteres en el cuadro de texto.

En las líneas 46-49 hay otros dos tipos de elementos `input`. El elemento `input "submit"` es un botón. Cuando el usuario oprime un botón "submit", el explorador envía los datos en el formulario al servidor Web para que los procese. El atributo **value** establece el texto que se va a mostrar en el botón (el valor predeterminado es **Enviar consulta**). El elemento `input "reset"` permite a un usuario restablecer todos los elementos del formulario a sus valores predeterminados. El atributo **value** del elemento `input "reset"` establece el texto que se va a mostrar en el botón (el valor predeterminado es **Restablecer**).

G.5 Formularios más complejos en XHTML

En la sección anterior, presentamos los formularios básicos. En esta sección presentaremos elementos y atributos para crear formularios más complejos. La figura G.4 contiene un formulario que solicita retroalimentación del usuario acerca de un sitio Web.

El elemento **textarea** (líneas 37-39) inserta en el formulario un cuadro de texto con varias líneas, conocido como *área de texto*. El número de filas se especifica con el atributo **rows**, y el número de columnas (es decir, caracteres) se especifica con el atributo **cols**. En este ejemplo, el elemento **textarea** es de cuatro filas de alto y 36 caracteres de ancho. Para mostrar texto predeterminado en el área de texto, coloque el texto deseado entre las etiquetas `<textarea>` y `</textarea>`. Podemos especificar texto predeterminado en otros tipos de entradas, como los cuadros de texto, mediante el uso del atributo **value**.

El elemento de entrada **"password"** en las líneas 46-47 inserta un cuadro de contraseña con el tamaño especificado por **size**. Un cuadro de contraseña permite a los usuarios introducir información delicada, como los números de tarjetas de crédito y las contraseñas, para lo cual "enmascara" la entrada de información con asteriscos (*). El valor actual que se introduce es el que se envía al servidor Web, no los caracteres que enmascaran la entrada.

Las líneas 54-71 introducen el elemento *checkbox* del formulario. Las casillas de verificación permiten a los usuarios realizar selecciones a partir de un conjunto de opciones. Cuando un usuario selecciona una casilla, aparece una marca de verificación. Si no la selecciona, permanece vacía. Cada elemento `input "checkbox"` crea una nueva casilla de verificación. Estas casillas pueden usarse por separado o en grupos. A las casillas de verificación que pertenecen a un grupo se les asigna el mismo atributo `name` (en este caso, "cosasquegustaron").

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. G.4: formulario2.html -->
6  <!-- Ejemplo de diseño de formularios 2 -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Formularios</title>
11    </head>
12
13    <body>
14
15     <h1>Formulario de retroalimentacion</h1>
16
17     <p>Llene este formulario para ayudarnos
18       a mejorar nuestro sitio.</p>
19
20     <form method = "post" action = "/cgi-bin/formmail">
21
22       <p>
23         <input type = "hidden" name = "recipient"
24           value = "deitel@deitel.com" />
25         <input type = "hidden" name = "subject"
26           value = "Formulario de retroalimentacion" />
27         <input type = "hidden" name = "redirect"
28           value = "principal.html" />
29       </p>
30
31       <p><label>Nombre:
32         <input name = "nombre" type = "text" size = "25" />
33       </label></p>
34
35       <!-- <textarea> crea un cuadro de texto con varias líneas -->
36       <p><label>Comentarios:<br />
37         <textarea name = "comentarios" rows = "4" cols = "36">
38         Escriba aquí sus comentarios.
39       </textarea>
40     </label></p>
41
42     <!-- <input type = "password"> inserta un -->
43     <!-- cuadro de texto en el que se muestran -->
44     <!-- asteriscos para cubrir los datos de entrada -->
45     <p><label>Direccion de e-mail:
46       <input name = "email" type = "password"
47         size = "25" />
48     </label></p>
49
50     <p>
51       <strong>Cosas que le gustaron:</strong><br />

```

Figura G.4 | Formulario con áreas de texto, un cuadro de contraseña y casillas de verificación. (Parte 1 de 3).

```

52
53     <label>Diseño del sitio
54     <input name = "cosasquegustaron" type = "checkbox"
55         value = "Diseño" /></label>
56
57     <label>Vinculos
58     <input name = "cosasquegustaron" type = "checkbox"
59         value = "Vinculos" /></label>
60
61     <label>Facilidad de uso
62     <input name = "cosasquegustaron" type = "checkbox"
63         value = "Facilidad" /></label>
64
65     <label>Imagenes
66     <input name = "cosasquegustaron" type = "checkbox"
67         value = "Imagenes" /></label>
68
69     <label>Codigo fuente
70     <input name = "cosasquegustaron" type = "checkbox"
71         value = "Codigo" /></label>
72 </p>
73
74 <p>
75     <input type = "submit" value =
76         "Enviar sus datos" />
77     <input type = "reset" value =
78         "Borrar sus datos" />
79 </p>
80
81 </form>
82
83 </body>
84 </html>

```

Como programar en Internet y WWW - Formularios - Microsoft Internet Exp...

Archivo Edición Ver Favoritos Herramientas Ayuda

Abre Búsqueda Favoritos

Dirección C:\MisProyectos\Apéndice\formulario2.html Ir

Formulario de retroalimentacion

Llene este formulario para ayudarnos a mejorar nuestro sitio.

Nombre:

Comentarios:
Escriba aqui sus comentarios.

Dirección de e-mail:

Cosas que le gustaron:

Diseño del sitio ☐ Vinculos ☐ Facilidad de uso ☐ Imagenes ☐ Codigo fuente ☐

Listo Mi PC

Figura G.4 | Formulario con áreas de texto, un cuadro de contraseña y casillas de verificación. (Parte 2 de 3).

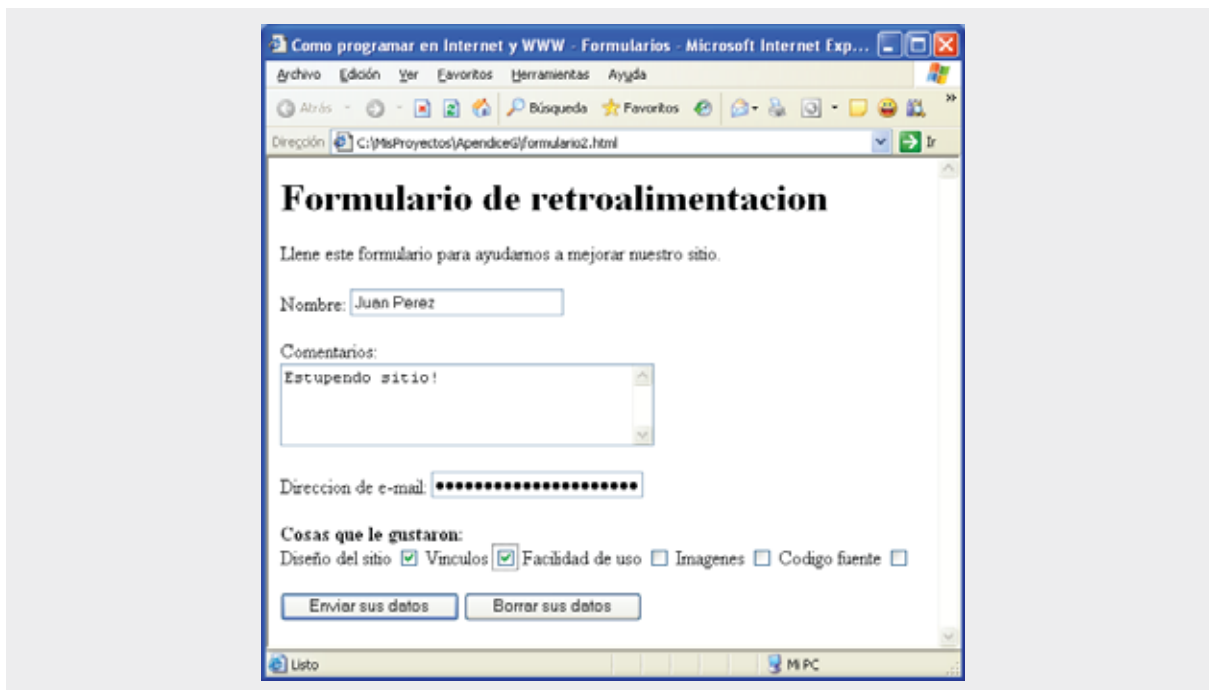


Figura G.4 | Formulario con áreas de texto, un cuadro de contraseña y casillas de verificación. (Parte 3 de 3).



Error común de programación G.2

Cuando su formulario tiene varias casillas de verificación con el mismo nombre (name), debe asegurarse que tengan distintos valores (value), o las secuencias de comandos que se ejecutan en el servidor Web no podrán distinguirlas.

Para continuar nuestra discusión acerca de los formularios, vamos a presentar un tercer ejemplo que introduce varios elementos adicionales de un formulario, en los cuales los usuarios pueden realizar selecciones (figura G.5). En este ejemplo, presentamos dos nuevos tipos de elementos input. El primer tipo es el *botón de opción* (líneas 76-94), que se especifica mediante el tipo "radio". Los botones de opción son similares a las casillas de verificación, excepto que sólo puede seleccionarse un botón de opción en un grupo de botones en un momento dado. Todos los botones de opción en un grupo tienen los mismos atributos name y se diferencian con base en sus distintos atributos value. El par atributo-valor checked = "checked" (línea 77) indica cuál botón de opción está seleccionado en un principio (si hay alguno). El atributo checked también se aplica a las casillas de verificación.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. G.5: formulario3.html -->
6  <!-- Ejemplo de diseño de formularios 3 -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Formularios</title>
11    </head>

```

Figura G.5 | Formulario que incluye botones de opción y una lista desplegable. (Parte 1 de 4).

```

12
13 <body>
14
15 <h1>Formulario de retroalimentacion</h1>
16
17 <p>Llene este formulario para ayudarnos
18 a mejorar nuestro sitio.</p>
19
20 <form method = "post" action = "/cgi-bin/formmail">
21
22 <p>
23 <input type = "hidden" name = "recipient"
24 value = "deitel@deitel.com" />
25 <input type = "hidden" name = "subject"
26 value = "Formulario de retroalimentacion" />
27 <input type = "hidden" name = "redirect"
28 value = "principal.html" />
29 </p>
30
31 <p><label>Nombre:
32 <input name = "nombre" type = "text" size = "25" />
33 </label></p>
34
35 <p><label>Comentarios:<br />
36 <textarea name = "comentarios" rows = "4"
37 cols = "36"></textarea>
38 </label></p>
39
40 <p><label>Direccion de e-mail:
41 <input name = "email" type = "password"
42 size = "25" /></label></p>
43
44 <p>
45 <strong>Cosas que le gustaron:</strong><br />
46
47 <label>Disenio del sitio
48 <input name = "cosasquegustaron" type = "checkbox"
49 value = "Disenio" /></label>
50
51 <label>Vinculos
52 <input name = "cosasquegustaron" type = "checkbox"
53 value = "Vinculos" /></label>
54
55 <label>Facilidad de uso
56 <input name = "cosasquegustaron" type = "checkbox"
57 value = "Facilidad" /></label>
58
59 <label>Imagenes
60 <input name = "cosasquegustaron" type = "checkbox"
61 value = "Imagenes" /></label>
62
63 <label>Codigo fuente
64 <input name = "cosasquegustaron" type = "checkbox"
65 value = "Codigo" /></label>
66 </p>
67
68 <!-- <input type = "radio" /> crea un botón de -->
69 <!-- opción. La diferencia entre los botones de -->

```

Figura G.5 | Formulario que incluye botones de opción y una lista desplegable. (Parte 2 de 4).

```

70      <!-- opción y las casillas de verificación es que -->
71      <!-- sólo se puede seleccionar uno en un grupo. -->
72      <p>
73          <strong>Como llego a nuestro sitio?:</strong><br />
74
75          <label>Motor de búsqueda
76              <input name = "comollegoalsitio" type = "radio"
77                  value = "motor de búsqueda" checked = "checked" />
78          </label>
79
80          <label>Vinculos de otro sitio
81              <input name = "comollegoalsitio" type = "radio"
82                  value = "vinculo" /></label>
83
84          <label>Sitio Web Deitel.com
85              <input name = "comollegoalsitio" type = "radio"
86                  value = "deitel.com" /></label>
87
88          <label>Referencia en un libro
89              <input name = "comollegoalsitio" type = "radio"
90                  value = "libro" /></label>
91
92          <label>Otro
93              <input name = "comollegoalsitio" type = "radio"
94                  value = "otro" /></label>
95
96      </p>
97
98      <p>
99          <label>Califique nuestro sitio:
100
101              <!-- la etiqueta <select> presenta una lista -->
102              <!-- desplegable con las opciones indicadas -->
103              <!-- por las etiquetas <option> -->
104              <select name = "calificacion">
105                  <option selected = "seleccion">Sorprendente</option>
106                  <option>10</option>
107                  <option>9</option>
108                  <option>8</option>
109                  <option>7</option>
110                  <option>6</option>
111                  <option>5</option>
112                  <option>4</option>
113                  <option>3</option>
114                  <option>2</option>
115                  <option>1</option>
116                  <option>Horrendo</option>
117              </select>
118
119          </label>
120      </p>
121
122      <p>
123          <input type = "submit" value =
124              "Enviar sus datos" />
125          <input type = "reset" value = "Borrar sus entradas" />
126      </p>
127

```

Figura G.5 | Formulario que incluye botones de opción y una lista desplegable. (Parte 3 de 4).

128 </form>
129
130 </body>
131 </html>

The figure consists of two screenshots of a Microsoft Internet Explorer browser window displaying a web page titled "Formulario de retroalimentacion". The browser's address bar shows the file path "C:\MisProyectos\Apndice2\formulario3.html".

The top screenshot shows the form in its initial state. It includes a title "Formulario de retroalimentacion", a subtitle "Llene este formulario para ayudarnos a mejorar nuestro sitio.", and several input fields: "Nombre:" (text), "Comentarios:" (text area), and "Direccion de e-mail:" (text). Below these are checkboxes for "Cosas que le gustaron:" (Diseño del sitio, Vinculos, Facilidad de uso, Imagenes, Código fuente) and radio buttons for "Como llego a nuestro sitio?:" (Motor de búsqueda, Vinculos de otro sitio, Sitio Web Destel.com, Referencia en un libro, Otro). There is also a dropdown menu for "Califique nuestro sitio:" set to "Sorprendente". At the bottom are two buttons: "Enviar sus datos" and "Borrar sus entradas".

The bottom screenshot shows the form after it has been filled out. The "Nombre:" field contains "Juan Perez", the "Comentarios:" field contains "Excelente sitio!", and the "Direccion de e-mail:" field is filled with asterisks. The "Facilidad de uso" checkbox is now checked. The "Motor de búsqueda" radio button is selected. The "Califique nuestro sitio:" dropdown menu is open, showing a list of numbers from 1 to 10, with "10" selected. The "Enviar sus datos" button is visible at the bottom left.

Figura G.5 | Formulario que incluye botones de opción y una lista desplegable. (Parte 4 de 4).



Error común de programación G.3

Si no se establecen los atributos `name` de los botones de opción con el mismo nombre en un formulario, se produce un error lógico, ya que el usuario podría seleccionarlos todos al mismo tiempo.

El elemento **`select`** (líneas 104-117) proporciona una lista desplegable de elementos, de los cuales el usuario puede seleccionar uno. El atributo `name` identifica la lista desplegable. El elemento **`option`** (líneas 105-116) agrega elementos a la lista desplegable. El atributo **`selected`** del elemento `option` especifica cuál elemento se muestra al principio como el elemento seleccionado.

G.6 Vínculos internos

En el apéndice F hablamos sobre cómo crear un hipervínculo entre dos páginas Web. La figura G.6 introduce los ***vínculos internos***: un mecanismo que permite al usuario saltar entre varias ubicaciones dentro del mismo documento. Los vínculos internos son útiles para documentos extensos que contienen muchas secciones. Al hacer clic en un vínculo interno, los usuarios pueden encontrar una sección sin tener que desplazarse por todo el documento completo.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. G.6: vinculos.html -->
6  <!-- Vínculos internos      -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10     <title>Como programar en Internet y WWW - Lista</title>
11    </head>
12
13    <body>
14
15     <!-- el atributo id crea un destino de hipervínculo interno -->
16     <h1 id = "features">Las mejores características de Internet</h1>
17
18     <!-- la dirección de un vínculo interno es "#id" -->
19     <p><a href = "#insectos">Ir a <em>Insectos favoritos</em></a></p>
20
21     <ul>
22       <li>Podemos conocer personas de otros países
23         alrededor del mundo.</li>
24
25       <li>Tenemos acceso a los nuevos medios, al momento en que se hacen públicos:
26         <ul>
27           <li>Nuevos juegos</li>
28           <li>Nuevas aplicaciones
29             <ul>
30               <li>De negocios</li>
31               <li>De entretenimiento</li>
32             </ul>
33           </li>
34
35           <li>Noticias a toda hora</li>
36           <li>Motores de búsqueda</li>
37           <li>Compras</li>
38           <li>Programación

```

Figura G.6 | Los hipervínculos internos facilitan la navegación dentro de las páginas. (Parte 1 de 3).

```

39         <ul>
40             <li>XHTML</li>
41             <li>Java</li>
42             <li>HTML dinamico</li>
43             <li>Secuencias de comandos</li>
44             <li>Nuevos lenguajes</li>
45         </ul>
46     </li>
47 </ul>
48 </li>
49
50 <li>Vinculos</li>
51 <li>Mantenerse en contacto con viejos amigos</li>
52 <li>Es la tecnologia del futuro!</li>
53 </ul>
54
55 <!-- el atributo id crea un destino de hipervínculo interno -->
56 <h1 id = "insectos">Mis 3 insectos favoritos</h1>
57
58 <p>
59
60     <!-- hipervínculo interno a las características -->
61     <a href = "#caracteristicas">Ir a <em>Características favoritas</em>
62 </a></p>
63
64 <ol>
65     <li>Libelula</li>
66     <li>Hormiga</li>
67     <li>Tarantula</li>
68 </ol>
69
70 </body>
71 </html>

```

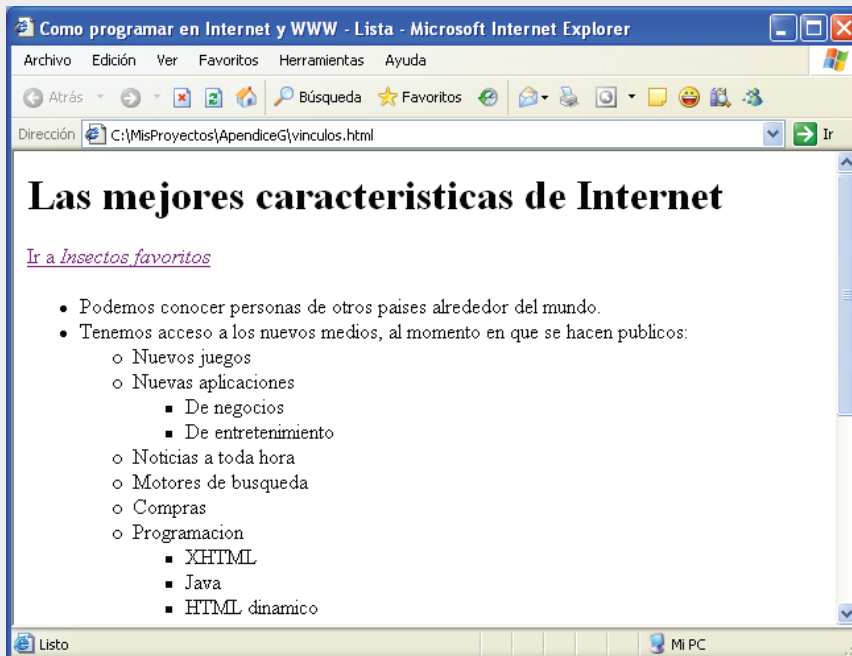


Figura G.6 | Los hipervínculos internos facilitan la navegación dentro de las páginas. (Parte 2 de 3).

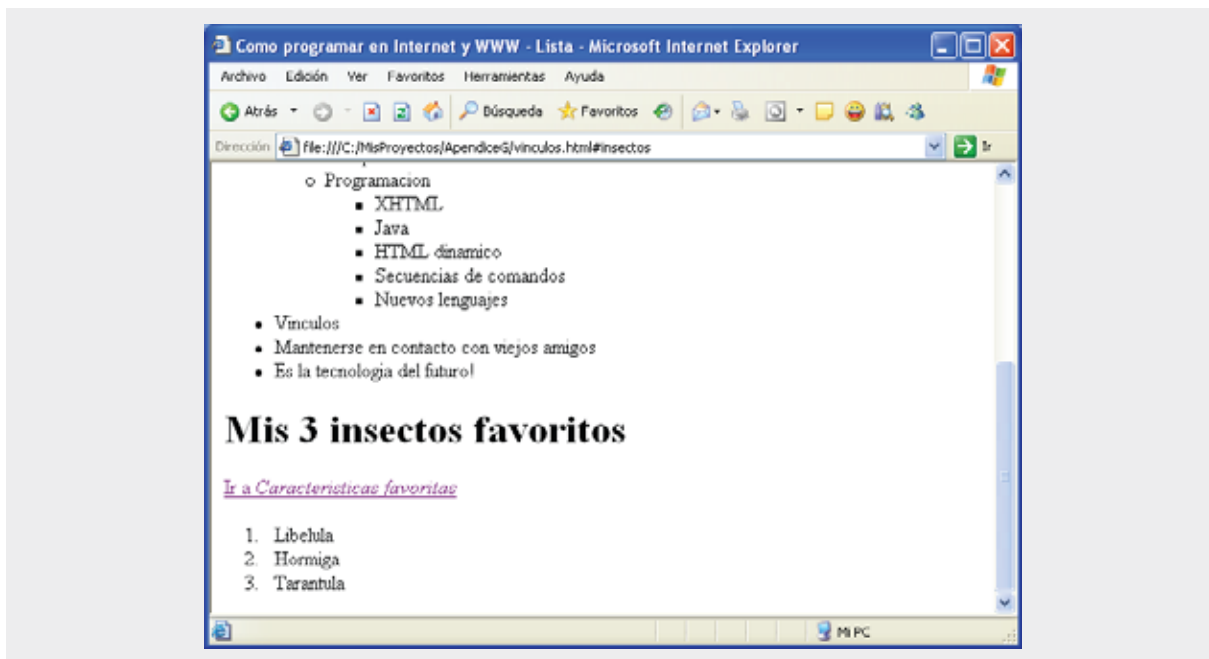


Figura G.6 | Los hipervínculos internos facilitan la navegación dentro de las páginas. (Parte 3 de 3).

La línea 16 contiene una etiqueta con el atributo `id` (llamado "características") para un hipervínculo interno. Para crear un vínculo a una etiqueta dentro de la misma página Web, el atributo `href` de un elemento ancla incluye el valor del atributo `id` precedido por un signo `#` (por ejemplo, `#características`). Las líneas 61-62 contienen un hipervínculo con el `id` `características` como destino. Al seleccionar este hipervínculo en un explorador Web, la ventana del explorador se desplaza hasta la etiqueta `h1` en la línea 16.



Observación de apariencia visual G.2

Los hipervínculos internos son útiles en documentos en XHTML que contienen grandes cantidades de información. Los vínculos internos a distintas partes de la página facilitan a los usuarios navegar por la misma. No tienen que desplazarse por la pantalla para encontrar la sección que desean.

Aunque no se demostró en este ejemplo, un hipervínculo puede especificar un vínculo interno en otro documento; para ello se especifica el nombre del documento, seguido de un signo `#` y del valor `id`, como en:

```
href = "nombrearchivo.html#id"
```

Por ejemplo, para crear un vínculo a una etiqueta con el atributo `id` llamado `listalibros` en `libros.html`, a `href` se le asigna `"libros.html#listalibros"`.

G.7 Creación y uso de mapas de imágenes

En el apéndice F, demostramos cómo pueden usarse las imágenes como hipervínculos, para crear vínculos a otros recursos en Internet. En esta sección presentaremos otra técnica para vincular imágenes, conocida como *mapas de imágenes*, en donde se designan ciertas áreas de una imagen (conocidas como *zonas activas*) como vínculos.¹ La figura G.7 introduce los mapas de imágenes y las zonas activas.

1. Los exploradores Web actuales no soportan los mapas de imágenes de XHTML 1.1. Por esta razón, utilizamos XHTML 1.0 Transicional, una versión anterior de XHTML del W3C. Para poder validar el código en la figura G.7 como XHTML 1.1, hay que eliminar el `#` del atributo `usemap` de la etiqueta `img` (línea 53).

Las líneas 20-48 definen un mapa de imagen mediante el uso de un elemento **map**. El atributo **id** (línea 20) identifica al mapa de imagen. Si se omite **id**, no se puede hacer referencia al mapa mediante una imagen (lo cual veremos en un momento). Las zonas activas se definen mediante elementos **area** (como se muestra en las líneas 25-27). El atributo **href** (línea 25) especifica el destino del vínculo (es decir, el recurso al cual se va a vincular). Los atributos **shape** (línea 25) y **coords** (línea 26) especifican la forma y las coordenadas de la zona activa, respectivamente. El atributo **alt** (línea 27) proporciona el texto alternativo para el vínculo.



Error común de programación G.4

*Si no se especifica un atributo **id** para un elemento **map**, el elemento **img** no podrá utilizar los elementos **area** de ese elemento **map** para definir zonas activas.*

```

1  <?xml version = "1.0" ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5  <!-- Fig. G.7: imagen.html -->
6  <!-- Creación y uso de Mapas de imágenes -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10      <title>
11        Como programar en Internet y WWW - Mapa de imagenes
12      </title>
13    </head>
14
15    <body>
16
17      <p>
18
19        <!-- la etiqueta <map> define un mapa de imagen -->
20        <map id = "imagen">
21
22          <!-- shape = "rect" indica un área rectangular, -->
23          <!-- con coordenadas para las esquinas superior -->
24          <!-- izquierda y superior derecha -->
25          <area href = "formulario.html" shape = "rect"
26            coords = "2,123,100,143"
27            alt = "Ir al formulario de retroalimentacion" />
28          <area href = "contacto.html" shape = "rect"
29            coords = "126,122,198,143"
30            alt = "Ir a la pagina de contacto" />
31          <area href = "principal.html" shape = "rect"
32            coords = "3,7,61,25" alt = "Ir a la pagina principal" />
33          <area href = "vinculos.html" shape = "rect"
34            coords = "150,5,213,25"
35            alt = "Ir a la pagina de vinculos" />
36
37          <!-- el valor "poly" crea una zona activa en forma -->
38          <!-- de un poligono, definido por las coordenadas -->
39          <area shape = "poly" alt = "Enviar correo electronico a los Deitel"
40            coords = "162,25,154,39,158,54,169,51,183,39,161,26"
41            href = "mailto:deitel@deitel.com" />
42
43          <!-- shape = "circle" indica un área circular -->
44          <!-- con el centro y el radio dados -->

```

Figura G.7 | Imagen con vínculos anclados a un mapa de imagen. (Parte I de 2).

```

45     <area href = "mailto:deitel@deitel.com"
46           shape = "circle" coords = "100,36,33"
47           alt = "Enviar correo electronico a los Deitel" />
48 </map>
49
50 <!-- <img src =... usemap = "id"> indica que con esta imagen -->
51 <!-- se va a usar el mapa de imagen especificado -->
52 
54 </p>
55 </body>
56 </html>

```

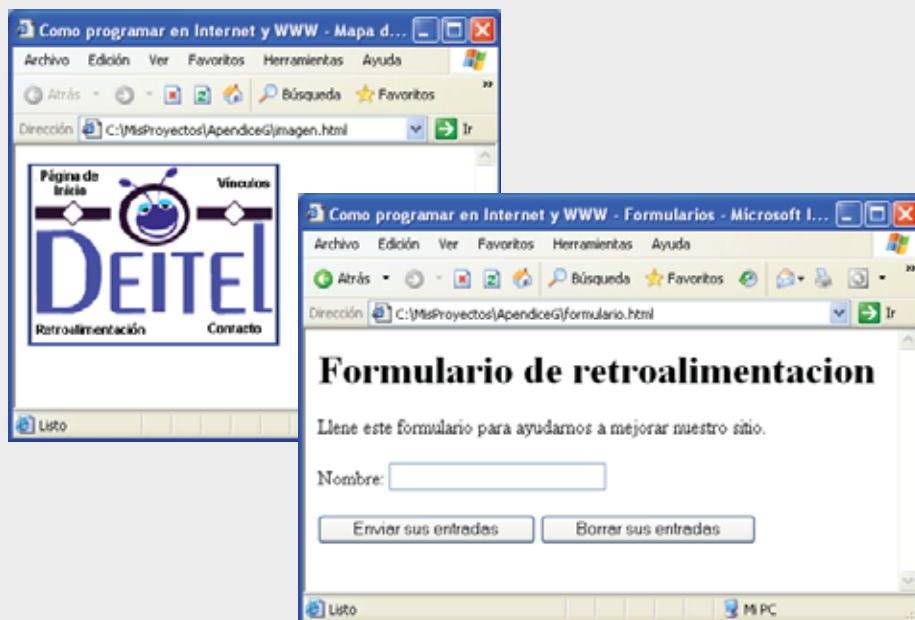


Figura G.7 | Imagen con vínculos anclados a un mapa de imagen. (Parte 2 de 2).

El marcado en las líneas 25-27 crea una **zona activa rectangular** (`shape = "rect"`) para las **coordenadas** especificadas en el atributo `coord`. Un par de coordenadas consiste en dos números que representan las ubicaciones de un punto en el eje x y el eje y , respectivamente. El eje x se extiende en forma horizontal y el eje y se extiende en forma vertical, a partir de la esquina superior izquierda de la imagen. Cada punto en una imagen tiene una coordenada x - y única. Para las zonas activas rectangulares, las coordenadas requeridas son las de las esquinas superior izquierda e inferior derecha del rectángulo. En este caso, la esquina superior izquierda del rectángulo se encuentra en 2 en el eje x , y en 123 en el eje y , lo cual se escribe como (2, 123). La esquina inferior derecha del rectángulo está en (54, 143). Las coordenadas se miden en píxeles.



Error común de programación G.5

Si se traslapan las coordenadas de un mapa de imagen, el explorador podría representar la primera zona activa que encuentre para esa área.

El elemento `area` del mapa en las líneas 39-41 asigna el atributo `shape = "poly"` para crear una zona activa con forma de polígono, usando las coordenadas en el atributo `coords`. Estas coordenadas representan cada **vértice**, o esquina, del polígono. El explorador conecta estos puntos con líneas para formar el área de la zona activa.

El elemento `area` del mapa en las líneas 45-47 asigna el atributo `shape = "circle"` para crear una **zona activa circular**. En este caso, el atributo `coords` especifica las coordenadas del centro del círculo y su radio, en píxeles.

Para utilizar un mapa de imagen con un elemento `img`, debemos asignar el atributo **usemap** del elemento `img` al atributo `id` de un elemento `map`. Las líneas 52-53 hacen referencia al mapa de imagen "#imagen". El mapa de imagen se encuentra dentro del mismo documento, por lo que se utiliza un vínculo interno.

G.8 Elementos meta

Los motores de búsqueda se utilizan para buscar sitios Web. Por lo general, para clasificar los sitios van siguiendo vínculos de página en página (lo que se conoce como "spidering" o "crawling") y guardan la información de identificación y clasificación para cada página. Una forma en la que los motores de búsqueda clasifican las páginas es leyendo el contenido en los elementos **meta** de cada página, el cual especifica información acerca de un documento.

Dos atributos importantes del elemento meta son **name**, que identifica el tipo de elemento meta, y **content**, que proporciona la información que utilizan los motores de búsqueda para clasificar páginas. La figura G.8 presenta el elemento meta.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. G.8: principal.html -->
6  <!-- La etiqueta <meta> -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10      <title>Como programar en Internet and WWW - Bienvenido</title>
11
12      <!-- las etiquetas <meta> proveen información a los -->
13      <!-- motores de búsqueda para clasificar un sitio -->
14      <meta name = "keywords" content = "pagina Web, diseño,
15        XHTML, tutorial, personal, ayuda, indice, formulario,
16        contacto, retroalimentacion, lista, vinculos, marco, deitel" />
17
18      <meta name = "description" content = "Este sitio Web le ayudara
19        a conocer los fundamentos de XHTML y el diseño de paginas Web
20        mediante el uso de ejemplos e instrucciones
21        interactivas." />
22
23    </head>
24
25    <body>
26
27      <h1>Bienvenido a nuestro sitio Web!</h1>
28
29      <p>Hemos disenado este sitio para enseniarle las
30      maravillas de <strong><em>XHTML</em></strong>. <em>XHTML</em>
31      esta mejor equipado que <em>HTML</em> para representar datos
32      complejos en Internet. <em>XHTML</em> aprovecha la estricta
33      sintaxis de XML para asegurar que los documentos esten bien
34      formados. Pronto conocera muchas de las estupendas nuevas
35      características de <em>XHTML.</em></p>
36
37      <p>Diviertase con el sitio!</p>
38
39    </body>
40  </html>

```

Figura G.8 | Las etiquetas meta proporcionan palabras clave y una descripción de una página. (Parte 1 de 2).

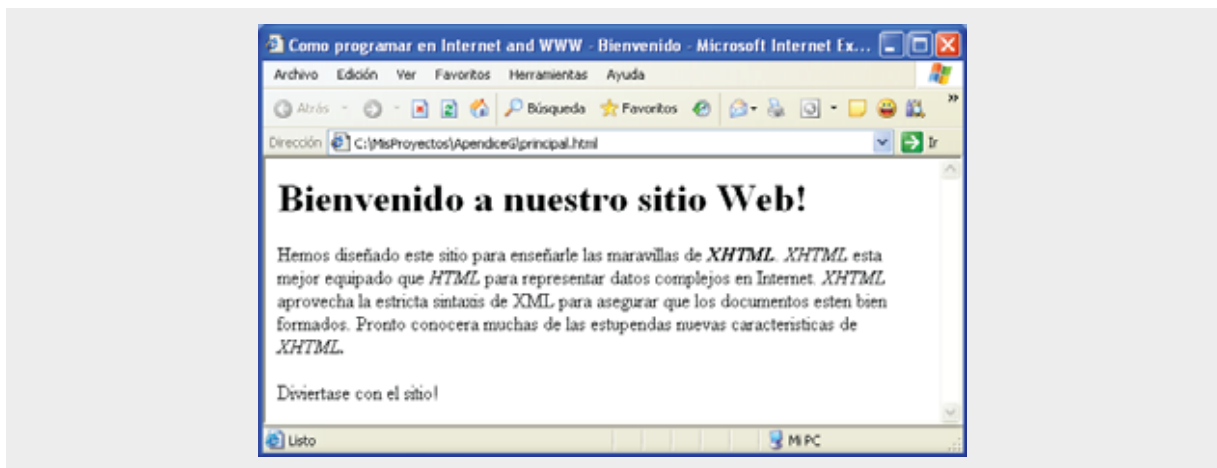


Figura G.8 | Las etiquetas meta proporcionan palabras clave y una descripción de una página. (Parte 2 de 2).

Las líneas 14-16 demuestran un elemento meta "keywords". El atributo `content` de dicho elemento meta proporciona a los motores de búsqueda una lista de palabras que describen a una página. Estas palabras se comparan con las palabras en las solicitudes de búsqueda. Por lo tanto, si incluye elementos meta y su información en `content`, podrá atraer más visitantes a su sitio Web.

Las líneas 18-21 demuestran un elemento meta "description". El atributo `content` de dicho elemento meta proporciona una descripción de un sitio en tres o cuatro líneas, escrita en forma de oraciones. Los motores de búsqueda también utilizan esta descripción para clasificar los sitios y, algunas veces, muestran esta información como parte de los resultados de una búsqueda.



Observación de ingeniería de software G.1

Los elementos meta no son visibles para los usuarios, y deben colocarse dentro de la sección `head` de un documento en XHTML. Si no se colocan en esta sección, los motores de búsqueda no los leerán.

G.9 Elemento `frameset`

Todas las páginas Web que presentamos en este libro tienen la habilidad de crear vínculos a otras páginas, pero sólo pueden mostrar una página a la vez. Los *Marcos* permiten a un desarrollador Web mostrar más de un documento de XHTML en el explorador al mismo tiempo. La figura G.9 utiliza marcos para mostrar los documentos en las figuras G.8 y G.10.

La mayoría de nuestros ejemplos anteriores se adhieren al tipo de documentos de XHTML 1.1, mientras que estos utilizan los tipos de documentos de XHTML 1.0.¹ Estos tipos de documentos se especifican en las líneas 2-3 y se requieren para los documentos que definen conjuntos de marcos, o que utilizan el atributo `target` para trabajar con conjuntos de marcos.

Por lo general, un documento que define un conjunto de marcos consiste en un elemento `html` que contiene un encabezado `head` y un elemento **`frameset`** (líneas 23-40). En la figura G.9, la etiqueta **`<frameset>`** (línea 23) informa al explorador que la página contiene marcos. El atributo **`cols`** especifica la distribución de las columnas del conjunto de marcos. El valor de `cols` proporciona la anchura de cada marco, ya sea en píxeles o como un porcentaje de la anchura del explorador. En este caso, el atributo `cols = "110, *"` informa al explorador que hay dos marcos verticales. El primer marco se extiende 110 píxeles a partir del borde izquierdo de la ventana del explorador,

1. XHTML 1.1 ya no soporta el uso de marcos. El W3C recomienda usar Hojas de estilo en cascada para lograr el mismo efecto. Sin embargo, los marcos aún se utilizan ampliamente en Internet y la mayoría de los exploradores los soportan. El elemento `frameset` y el atributo `target` cuentan todavía con soporte en las definiciones de tipo de documentos XHTML 1.0 Frameset y XHTML 1.0 Transicional, respectivamente. Para obtener más información, consulte la página www.w3.org/TR/xhtml1/#tds.

y el segundo marco rellena el resto de la anchura del explorador (según lo indica el asterisco). De manera similar, el atributo *rows* de frameset puede utilizarse para especificar el número de filas y el tamaño de cada fila en un conjunto de marcos.

Los documentos que se van a cargar en el elemento frameset se especifican mediante elementos **frame** (líneas 27-28 en este ejemplo). El atributo *src* especifica el URL de la página que se va a mostrar en el marco. Cada marco tiene atributos *name* y *src*. El primer marco (que abarca 110 píxeles en el lado izquierdo del frameset), llamado marcoizq, muestra la página nav.html (figura G.10). El segundo marco, llamado principal, muestra la página principal.html (figura G.8).

El atributo *name* identifica a un marco y permite que los hipervínculos en un conjunto de marcos especifiquen el elemento frame de destino (*target*) en el que se debe mostrar un documento vinculado, cuando el usuario haga clic en el vínculo. Por ejemplo:

```
<a href = "vinculos.html" target = "principal">
```

carga la página vinculos.html en el marco cuyo nombre es "principal".

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
4
5  <!-- Fig. G.9: indice.html -->
6  <!-- Marcos en XHTML I -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head>
10      <title>Como programar en Internet y WWW - Principal</title>
11      <meta name = "keywords" content = "Wpagina Web, diseño,
12        XHTML, tutorial, personal, ayuda, indice, formulario,
13        contacto, retroalimentacion, lista, vinculos, marco, deitel" />
14
15      <meta name = "description" content = "Este sitio Web le ayudara
16        a conocer los fundamentos de XHTML y el diseño de paginas Web
17        mediante el uso de ejemplos e instrucciones
18        interactivas." />
19
20    </head>
21
22    <!-- la etiqueta <frameset> establece las dimensiones del marco -->
23    <frameset cols = "110,*">
24
25      <!-- los elementos frame especifican las páginas que se -->
26      <!-- cargan en un marco dado -->
27      <frame name = "marcoizq" src = "nav.html" />
28      <frame name = "principal" src = "principal.html" />
29
30      <noframes>
31        <body>
32          <p>Esta pagina utiliza marcos, pero su explorador no
33            soporta su uso.</p>
34
35          <p>Por favor <a href = "nav.html">siga este vinculo para
36            explorar nuestro sitio sin marcos</a>.</p>
37        </body>
38      </noframes>
39
40    </frameset>
41  </html>

```

Figura G.9 | Documento con marcos en XHTML, con navegación y contenido. (Parte I de 2).

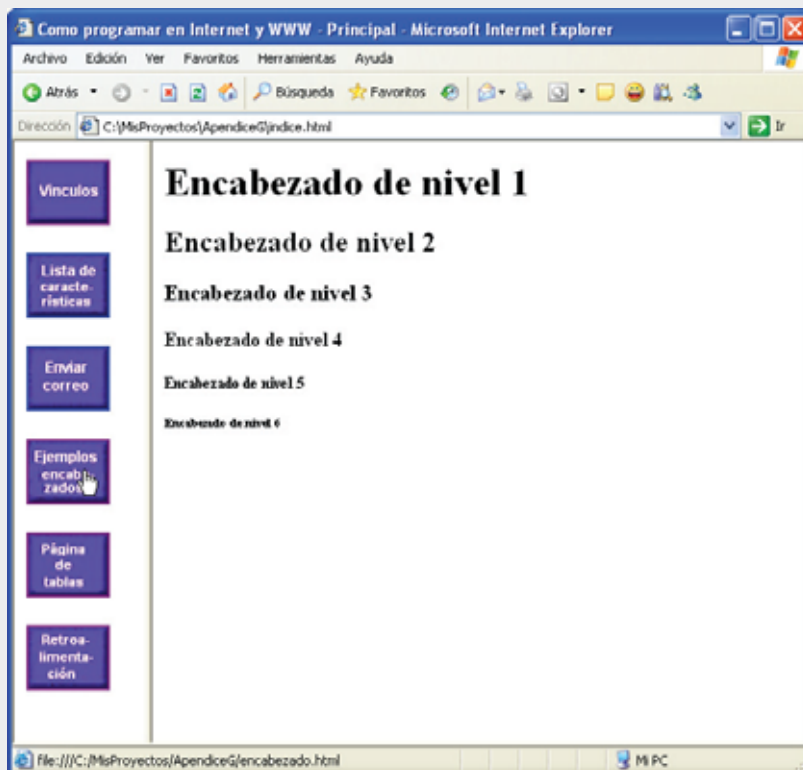
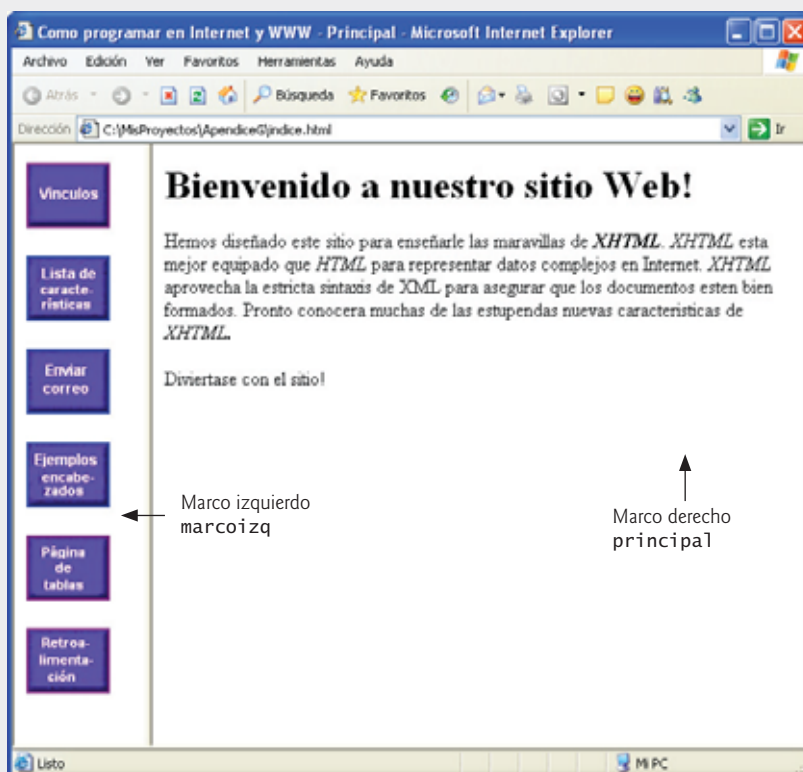


Figura G.9 | Documento con marcos en XHTML, con navegación y contenido. (Parte 2 de 2).

No todos los exploradores soportan el uso de marcos. XHTML cuenta con el elemento **noframes** (líneas 30-38) para permitir que los diseñadores de documentos en XHTML especifiquen un contenido alternativo para los exploradores que no soportan marcos.



Tip de portabilidad G.1

*Algunos exploradores no soportan el uso de marcos. Use el elemento **noframes** dentro de un elemento **frameset** para dirigir a los usuarios a una versión de su sitio sin marcos.*

La figura G.10 es la página Web que se muestra en el marco izquierdo de la figura G.9. Este documento en XHTML proporciona los botones de navegación que, cuando se hace clic en ellos, determinan el documento que se va a mostrar en el marco derecho.

La línea 27 (figura G.9) muestra la página de XHTML en la figura G.10. El atributo **target** de la ancla (línea 18 en la figura G.10) especifica que los documentos vinculados se van a cargar en el marco principal (línea 28 en la figura G.9). Puede establecerse un atributo **target** con un número de valores preestablecidos: **"_blank"** carga la página en una nueva ventana del explorador, **"_self"** carga la página en el marco en el que aparece el elemento ancla, y **"_top"** carga la página en la ventana completa del explorador (es decir, elimina el frameset).

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5  <!-- Fig.G.10: nav.html -->
6  <!-- Uso de imágenes como anclas de vínculos -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9
10   <head>
11     <title>Como programar en Internet y WWW - Barra de navegacion
12     </title>
13   </head>
14
15   <body>
16
17     <p>
18       <a href = "vinculos.html" target = "principal">
19         <img src = "botones/vinculos.jpg" width = "65"
20           height = "50" alt = "Pagina de vinculos" />
21       </a><br />
22
23       <a href = "lista.html" target = "principal">
24         <img src = "botones/lista.jpg" width = "65"
25           height = "50" alt = "Pagina de ejemplo de listas" />
26       </a><br />
27
28       <a href = "contacto.html" target = "principal">
29         <img src = "botones/contacto.jpg" width = "65"
30           height = "50" alt = "Pagina de contacto" />
31       </a><br />
32
33       <a href = "encabezado.html" target = "principal">
34         <img src = "botones/encabezado.jpg" width = "65"
35           height = "50" alt = "Pagina de encabezado" />
36       </a><br />

```

Figura G.10 | Documento en XHTML que se muestra en el marco izquierdo de la figura G.9. (Parte 1 de 2).

```

37
38     <a href = "tabla1.html" target = "principal">
39         <img src = "botones/tabla.jpg" width = "65"
40             height = "50" alt = "Pagina de tablas" />
41     </a><br />
42
43     <a href = "formulario.html" target = "principal">
44         <img src = "botones/formulario.jpg" width = "65"
45             height = "50" alt = "Formulario de retroalimentacion" />
46     </a><br />
47 </p>
48
49 </body>
50 </html>

```

Figura G.10 | Documento en XHTML que se muestra en el marco izquierdo de la figura G.9. (Parte 2 de 2).

G.10 Elementos frameset anidados

Podemos utilizar el elemento frameset para crear esquemas más complejos en una página Web; para ello hay que anidar elementos frameset, como en la figura G.11. El elemento frameset anidado en este ejemplo muestra los documentos de XHTML de las figuras G.7, G.8 y G.10.

```

1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
4
5  <!-- Fig. G.11: indice2.html -->
6  <!-- Marcos en XHTML II -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9      <head>
10         <title>Como programar en Internet y WWW - Principal</title>
11
12         <meta name = "keywords" content = "pagina Web, diseño,
13             XHTML, tutorial, personal, ayuda, indice, formulario,
14             contacto, retroalimentacion, lista, vinculos, marco, deitel" />
15
16         <meta name = "description" content = "Este sitio Web le ayudara
17             a conocer los fundamentos de XHTML y el diseño de paginas Web
18             mediante el uso de ejemplos e instrucciones
19             interactivas." />
20
21     </head>
22
23     <frameset cols = "110,*">
24         <frame name = "marcoizq" src = "nav.html" />
25
26         <!-- los elementos frameset anidados se usan para cambiar -->
27         <!-- el formato y la distribución del conjunto de marcos -->
28         <frameset rows = "175,*">
29             <frame name = "imagen" src = "imagen.html" />
30             <frame name = "principal" src = "principal.html" />
31         </frameset>
32
33     <noframes>

```

Figura G.11 | Sitio Web con marcos mediante el uso de elementos frameset anidados. (Parte 1 de 2).

```

34      <body>
35      <p>Esta pagina utiliza marcos, pero su explorador no
36      soporta su uso.</p>
37
38      <p>Por favor <a href = "nav.html">siga este vinculo para
39      explorar nuestro sitio sin marcos</a>.</p>
40      </body>
41  </noframes>
42
43  </frameset>
44  </html>

```



Figura G.11 | Sitio Web con marcos mediante el uso de elementos frameset anidados. (Parte 2 de 2).

El elemento frameset exterior (líneas 23-43) define dos columnas. El marco izquierdo se extiende sobre los primeros 110 píxeles, a partir del borde izquierdo del explorador, y el marco derecho ocupa el resto de la anchura de la ventana. El elemento frame en la línea 24 especifica que se va a mostrar el documento `nav.html` (figura G.10) en la columna izquierda.

Las líneas 28-31 definen un elemento frameset anidado para la segunda columna del elemento frameset exterior. Este elemento frameset define dos filas. La primera fila se extiende 175 píxeles a partir de la parte superior de la ventana del explorador, según lo indica el atributo `rows = "175, *"`. La segunda fila ocupa el resto de la altura de la ventana del explorador. El elemento frame en la línea 29 especifica que la primera fila del elemento frameset anidado va a mostrar la página `imagen.html` (figura G.7). El elemento frame en la línea 30 especifica que la segunda fila del elemento frameset anidado va a mostrar la página `principal.html` (figura G.8).



Tip de prevención de errores G.1

Al utilizar elementos `frameset` anidados, aplique sangría a cada elemento de la etiqueta `<frame>`. Esta práctica mejora la legibilidad de la página y facilita su depuración.

G.11 Recursos Web

www.vbxml.com/xhtml/articles/xhtml_tables

El sitio Web *VBXML.com* contiene un tutorial acerca de la creación de tablas de XHTML.

www.webreference.com/xml/reference/xhtml.html

Esta página Web contiene una lista de etiquetas de XHTML de uso común, como las etiquetas de encabezado, de tablas, de marcos y de formularios. También proporciona una descripción de cada etiqueta.



Caracteres especiales de HTML/XHTML

La tabla de la figura H.1 muestra muchos caracteres especiales de uso común en HTML/XHTML; el Consorcio World Wide Web llama a estos caracteres *referencias a entidades de caracteres*. Para obtener un listado completo de las referencias a entidades de caracteres, visite el sitio

www.w3.org/TR/REC-html40/sgml/entities.html

Carácter	Codificación XHTML	Carácter	Codificación XHTML
espacio sin interrupción	 	ê	ê
§	§	ì	ì
©	©	í	í
®	®	î	î
π	¼	ñ	ñ
∫	½	ò	ò
Ω	¾	ó	ó
à	à	ô	ô
á	á	õ	õ
â	â	÷	÷
ã	ã	ù	ù
ä	å	ú	ú
ç	ç	û	û
è	è	•	•
é	é	™	™

Figura H.1 | Caracteres especiales de XHTML.



Colores en HTML/XHTML

I.1 Colores

Los colores pueden especificarse mediante el uso de un nombre estándar (como `aqua`) o un valor RGB hexadecimal (como `#00FFFF` para el color aguamarina). De los seis dígitos hexadecimales en un valor RGB, los primeros dos representan la cantidad de rojo en el color, los dos de en medio la cantidad de verde, y los últimos dos representan la cantidad de azul. Por ejemplo, el color negro (`black`) es la ausencia de color y se define mediante el valor `#000000`, mientras que el blanco (`white`) es la cantidad máxima de rojo, verde y azul, y se define por el valor `#FFFFFF`. El rojo puro es `#FF0000`, el verde puro (que se conoce como `lime`) es `#00FF00` y el azul puro es `#0000FF`. En el estándar, el color verde (`green`) se define como `#008000`. La figura I.1 contiene el conjunto de colores estándar de HTML/XHTML. La figura I.2 contiene el conjunto de colores extendido de HTML/XHTML.

Nombre del color	Valor	Nombre del color	Valor
aqua	#00FFFF	navy	#000080
black	#000000	olive	#808000
blue	#0000FF	purple	#800080
fuchsia	#FF00FF	red	#FF0000
gray	#808080	silver	#C0C0C0
green	#008000	teal	#008080
lime	#00FF00	yellow	#FFFF00
maroon	#800000	white	#FFFFFF

Figura I.1 | Colores estándar de HTML/XHTML y sus valores RGB hexadecimales.

Nombre del color	Valor	Nombre del color	Valor
aliceblue	#F0F8FF	ghostwhite	#F8F8FF
antiquewhite	#FAEBD7	gold	#FFD700
aquamarine	#7FFFD4	goldenrod	#DAA520
azure	#F0FFFF	greenyellow	#ADFF2F
beige	#F5F5DC	honeydew	#F0FFF0
bisque	#FFE4C4	hotpink	#FF69B4
blanchedalmond	#FFEBCD	indianred	#CD5C5C
blueviolet	#8A2BE2	indigo	#4B0082
brown	#A52A2A	ivory	#FFFFFF
burlywood	#DEB887	khaki	#F0E68C
cadetblue	#5F9EA0	lavender	#E6E6FA
chartreuse	#7FFF00	lavenderblush	#FFF0F5
chocolate	#D2691E	lawngreen	#7CFC00
coral	#FF7F50	lemonchiffon	#FFFACD
cornflowerblue	#6495ED	lightblue	#ADD8E6
cornsilk	#FFF8DC	lightcoral	#F08080
crimson	#DC1436	lightcyan	#E0FFFF
cyan	#00FFFF	lightgoldenrodyellow	#FAFAD2
darkblue	#00008B	lightgreen	#90EE90
darkcyan	#008B8B	lightgrey	#D3D3D3
darkgoldenrod	#8B860B	lightpink	#FFB6C1
darkgray	#A9A9A9	lightsalmon	#FFA07A
darkgreen	#006400	lightseagreen	#20B2AA
darkkhaki	#BDB76B	lightskyblue	#87CEFA
darkmagenta	#8B008B	lightslategray	#778899
darkolivegreen	#556B2F	lightsteelblue	#B0C4DE
darkorange	#FF8C00	lightyellow	#FFFFE0
darkorchid	#9932CC	limegreen	#32CD32
darkred	#8B0000	mediumaquamarine	#66CDAA
darksalmon	#E9967A	mediumblue	#0000CD
darkseagreen	#8FBC8F	mediumorchid	#BA55D3
darkslateblue	#483D8B	mediumpurple	#9370DB
darkslategray	#2F4F4F	mediumseagreen	#3CB371
darkturquoise	#00CED1	mediumslateblue	#7B68EE
darkviolet	#9400D3	mediumspringgreen	#00FA9A
deeppink	#DD1493	mediumturquoise	#48D1CC
deepskyblue	#00BFFF	mediumvioletred	#C71585
dimgray	#696969	midnightblue	#191970
dodgerblue	#1E90FF	mintcream	#F5FFFA
firebrick	#B22222	mistyrose	#FFE4E1
floralwhite	#FFFAF0	moccasin	#FFE4V5
forestgreen	#228B22	navajowhite	#FFDEAD
gainsboro	#DCDCDC	oldlace	#FDF5E6

Figura I.2 | Colores extendidos de HTML/XHTML y sus valores RGB hexadecimales. (Parte I de 2).

Nombre del color	Valor	Nombre del color	Valor
olivedrab	#6B8E23	sandybrown	#F4A460
orange	#FFA500	seagreen	#2E8B57
orangered	#FF4500	seashell	#FFF5EE
orchid	#DA70D6	sienna	#A0522D
palegoldenrod	#EEE8AA	skyblue	#87CEEB
palegreen	#98FB98	slateblue	#6A5ACD
paleturquoise	#AFEEEE	slategray	#708090
palevioletred	#DB7093	snow	#FFFAFA
papayawhip	#FFEFD5	springgreen	#00FF7F
peachpuff	#FFDAB9	steelblue	#4682B4
peru	#CD853F	tan	#D2B48C
pink	#FFC0CB	thistle	#D8BFD8
plum	#DDA0DD	tomato	#FF6347
powderblue	#B0E0E6	turquoise	#40E0D0
rosybrown	#BC8F8F	violet	#EE82EE
royalblue	#4169E1	wheat	#F5DEB3
saddlebrown	#8B4513	whitesmoke	#F5F5F5
salmon	#FA8072	yellowgreen	#9ACD32

Figura I.2 | Colores extendidos de HTML/XHTML y sus valores RGB hexadecimales. (Parte 2 de 2).



Código del caso de estudio del ATM

J.1 Implementación del caso de estudio del ATM

Este apéndice contiene la implementación funcional completa del sistema ATM que diseñamos en las nueve secciones tituladas “Caso de estudio de ingeniería de software” en los capítulos 1, 3-9 y 11. La implementación comprende 655 líneas de código en C#. Consideramos las 11 clases en el orden en el que las identificamos en la sección 4.11 (con la excepción de *Transaccion*, que se introdujo en el capítulo 11 como la clase base de las clases *SolicitudSaldo*, *Retiro* y *Deposito*):

- *ATM*
- *Pantalla*
- *Teclado*
- *DispensadorEfectivo*
- *RanuraDeposito*
- *Cuenta*
- *BaseDatosBanco*
- *Transaccion*
- *SolicitudSaldo*
- *Retiro*
- *Deposito*

Aplicamos los lineamientos que vimos en las secciones 9.17 y 11.9 para codificar estas clases, con base en la manera en que las modelamos en los diagramas de clases UML de las figuras 11.21 y 11.22. Para desarrollar los cuerpos de los métodos de las clases, nos referimos a los diagramas de actividad presentados en la sección 6.9 y los diagramas de comunicaciones y secuencia presentados en la sección 8.14. Observe que nuestro diseño del ATM no especifica toda la lógica de programación, por lo que tal vez no especifique todos los atributos y operaciones requeridos para completar la implementación del ATM. Ésta es una parte normal del proceso de diseño orientado a objetos. A medida que implementamos el sistema, completamos la lógica del programa, agregando atributos y comportamientos según sea necesario para construir el sistema ATM especificado por el documento de requerimientos de la sección 3.10.

Concluimos la discusión presentando un arnés de prueba (*CasoEstudioATM* en la sección J.13) que crea un objeto de la clase *ATM* y lo inicia llamando a su método *Run*. Recuerde que estamos desarrollando la primera versión

del sistema ATM que se ejecuta en una computadora personal, y utiliza el teclado y el monitor para lograr la mayor semejanza posible con el teclado y la pantalla de un ATM. Además, simulamos las acciones del dispensador de efectivo y la ranura de depósito del ATM. Tratamos de implementar el sistema de manera tal que las versiones reales de hardware de esos dispositivos pudieran integrarse sin necesidad de cambios considerables en el código. [Nota: para los fines de esta simulación, hemos incluido dos cuentas predefinidas en la clase BaseDatosBanco. La primera cuenta tiene el número de cuenta 12345 y el NIP 54321. La segunda cuenta tiene el número de cuenta 98765 y el NIP 56789. Cuando pruebe el ATM, utilice estas cuentas.]

J.2 La clase ATM

La clase ATM (figura J.1) representa al ATM como un todo. Las líneas 5-11 implementan los atributos de la clase. Determinamos todos estos atributos (excepto uno) de los diagramas de clase de las figuras 11.21 y 11.22. La línea 5 declara el atributo `bool` `usuarioAutenticado` de la figura 11.22. La línea 6 declara un atributo que no se incluye en nuestro diseño UML: el atributo `int` `numeroCuentaActual`, que lleva el registro del número de cuenta del usuario autenticado actual. Las líneas 7-11 declaran variables de instancia de tipo de referencia, correspondientes a las asociaciones de la clase ATM modeladas en el diagrama de clases de la figura 11.21. Estos atributos permiten al ATM acceder a sus partes (es decir, su `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`) e interactuar con la base de datos de información de cuentas bancarias (es decir, un objeto `BaseDatosBanco`).

```

1  // ATM.cs
2  // Representa a un cajero automático.
3  public class ATM
4  {
5      private bool usuarioAutenticado; // verdadero si el usuario es autenticado
6      private int numeroCuentaActual; // número de cuenta del usuario
7      private Pantalla pantalla; // referencia a la pantalla del ATM
8      private Teclado teclado; // referencia al teclado del ATM
9      private DispensadorEfectivo dispensadorEfectivo; // referencia al dispensador de
      efectivo del ATM
10     private RanuraDeposito ranuraDeposito; // referencia a la ranura de depósito del ATM
11     private BaseDatosBanco baseDatosBanco; // referencia a la información de la base de
      datos de cuentas
12
13     // enumeración que representa las opciones del menú principal
14     private enum OpcionMenu
15     {
16         SOLICITUD_SALDO = 1,
17         RETIRO = 2,
18         DEPOSITO = 3,
19         SALIR_ATM = 4
20     } // fin de la enumeración OpcionMenu
21
22     // el constructor sin parámetro inicializa las variables de instancia
23     public ATM()
24     {
25         usuarioAutenticado = false; // al principio el usuario no está autenticado
26         numeroCuentaActual = 0; // al principio no hay número de cuenta actual
27         pantalla = new Pantalla(); // crea la pantalla
28         teclado = new Teclado(); // crea el teclado
29         dispensadorEfectivo = new DispensadorEfectivo(); // crea el dispensador de
            efectivo
30         ranuraDeposito = new RanuraDeposito(); // crea la ranura de depósito
31         baseDatosBanco = new BaseDatosBanco(); // crea la base de datos de información
            de las cuentas
32     } // fin del constructor
33

```

Figura J.1 | La clase ATM representa al ATM. (Parte 1 de 3).

```

34 // inicia ATM
35 public void Ejecutar()
36 {
37     // bienvenida y autenticación de usuarios; realiza transacciones
38     while ( true ) // ciclo infinito
39     {
40         // itera mientras el usuario no sea autenticado
41         while ( !usuarioAutenticado )
42         {
43             pantalla.MostrarLineaMensaje( "\n¡Bienvenido!" );
44             AutenticarUsuario(); // autentica el usuario
45         } // fin de while
46
47         RealizarTransacciones(); // para el usuario autenticado
48         usuarioAutenticado = false; // se restablece antes de la siguiente sesión con
            el ATM
49         numeroCuentaActual = 0; // se restablece antes de la siguiente sesión con el ATM
50         pantalla.MostrarLineaMensaje( "\n¡Gracias! ¡Adiós!" );
51     } // fin de while
52 } // fin del método Ejecutar
53
54 // trata de autenticar al usuario con la base de datos
55 private void AutenticarUsuario()
56 {
57     // pide el número de cuenta y lo recibe como entrada del usuario
58     pantalla.MostrarMensaje( "\nIntroduzca su número de cuenta: " );
59     int numeroCuenta = teclado.ObtenerEntrada();
60
61     // pide el NIP y lo recibe como entrada del usuario
62     pantalla.MostrarMensaje( "\nIntroduzca su NIP: " );
63     int pin = teclado.ObtenerEntrada();
64
65     // establece usuarioAutenticado al valor booleano devuelto por la base de datos
66     usuarioAutenticado =
67         baseDatosBanco.AutenticarUsuario( numeroCuenta, pin );
68
69     // verifica si se realizó la autenticación con éxito
70     if ( usuarioAutenticado )
71         numeroCuentaActual = numeroCuenta; // guarda el # de cuenta del usuario
72     else
73         pantalla.MostrarLineaMensaje(
74             "Número de cuenta o NIP inválido. Intente de nuevo." );
75 } // fin del método AutenticarUsuario
76
77 // muestra el menú principal y realiza las transacciones
78 private void RealizarTransacciones()
79 {
80     Transaccion transaccionActual; // la transacción que se está procesando
81     bool usuarioSalio = false; // el usuario no ha elegido salir
82
83     // itera mientras el usuario no elija la opción para salir
84     while ( !usuarioSalio )
85     {
86         // muestra el menú principal y obtiene la selección del usuario
87         int seleccionMenuPrincipal = MostrarMenuPrincipal();
88
89         // decide cómo proceder, con base en la selección del menú del usuario
90         switch ( ( OpcionMenu ) seleccionMenuPrincipal )
91         {

```

Figura J.1 | La clase ATM representa al ATM. (Parte 2 de 3).

```

92         // el usuario elije realizar uno de tres tipos de transacciones
93         case OpcionMenu.SOLICITUD_SALDO:
94         case OpcionMenu.RETIRO:
95         case OpcionMenu.DEPOSITO:
96             // se inicializa como nuevo objeto del tipo elegido
97             transaccionActual =
98                 CrearTransaccion( seleccionMenuPrincipal );
99             transaccionActual.Ejecutar(); // ejecuta la transacción
100             break;
101         case OpcionMenu.SALIR_ATM: // el usuario eligió terminar la sesión
102             pantalla.MostrarLineaMensaje( "\nSaliendo del sistema..." );
103             usuarioSalio = true; // esta sesión con el ATM debe terminar
104             break;
105         default: // el usuario no introdujo un entero del 1 al 4
106             pantalla.MostrarLineaMensaje(
107                 "\nNo introdujo una selección válida. Intente de nuevo." );
108             break;
109     } // fin de switch
110 } // fin de while
111 } // fin del método RealizarTransacciones
112
113 // muestra el menú principal y devuelve una selección de entrada
114 private int MostrarMenuPrincipal()
115 {
116     pantalla.MostrarLineaMensaje( "\nMenú principal:" );
117     pantalla.MostrarLineaMensaje( "1 - Ver mi saldo" );
118     pantalla.MostrarLineaMensaje( "2 - Retirar efectivo" );
119     pantalla.MostrarLineaMensaje( "3 - Depositar fondos" );
120     pantalla.MostrarLineaMensaje( "4 - Salir\n" );
121     pantalla.MostrarMensaje( "Introduzca una opción: " );
122     return teclado.ObtenerEntrada(); // devuelve la selección del usuario
123 } // fin del método MostrarMenuPrincipal
124
125 // devuelve un objeto de la clase especificada, derivada de Transaccion
126 private Transaccion CrearTransaccion( int tipo )
127 {
128     Transaccion temp = null; // referencia Transaccion nula
129
130     // determina el tipo de Transaccion que va a crear
131     switch ( ( OpcionMenu ) tipo )
132     {
133         // crea nueva transacción SolicitudSaldo
134         case OpcionMenu.SOLICITUD_SALDO:
135             temp = new SolicitudSaldo( numeroCuentaActual,
136                 pantalla, baseDatosBanco );
137             break;
138         case OpcionMenu.RETIRO: // crea nueva transacción Retiro
139             temp = new Retiro( numeroCuentaActual, pantalla,
140                 baseDatosBanco, teclado, dispensadorEfectivo );
141             break;
142         case OpcionMenu.DEPOSITO: // crea nueva transacción Deposito
143             temp = new Deposito( numeroCuentaActual, pantalla,
144                 baseDatosBanco, teclado, ranuraDeposito );
145             break;
146     } // fin de switch
147
148     return temp;
149 } // fin del método CrearTransaccion
150 } // fin de la clase ATM

```

Figura J.1 | La clase ATM representa al ATM. (Parte 3 de 3).

Las líneas 14-20 definen una enumeración que corresponde a las cuatro opciones en el menú principal del ATM (es decir, solicitud de saldo, retiro, depósito y salir). Las líneas 23-32 declaran el constructor de la clase ATM, el cual inicializa los atributos de la clase. Cuando se crea un objeto ATM por primera vez, ningún usuario está autenticado, por lo que la línea 25 inicializa `usuarioAutenticado` con `false`. La línea 26 inicializa `numeroCuentaActual` con 0, ya que no hay un usuario actual todavía. Las líneas 27-30 crean instancias de nuevos objetos para representar las partes del ATM. Recuerde que la clase ATM tiene relaciones de composición con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`, por lo que la clase ATM es responsable de su creación. La línea 31 crea un nuevo objeto `BaseDatosBanco`. Como veremos pronto, el objeto `BaseDatosBanco` crea dos objetos `Cuenta` que pueden usarse para probar el ATM. [Nota: si éste fuera un sistema ATM real, la clase ATM recibiría una referencia a un objeto de base de datos existente, creado por el banco. No obstante, en esta implementación sólo estamos simulando la base de datos del banco, por lo que la clase ATM crea el objeto `BaseDatosBanco` con el que interactúa.]

Implementación de la operación

El diagrama de clases de la figura 11.22 no lista operaciones para la clase ATM. Ahora vamos a implementar una operación (es decir, un método `public`) en la clase ATM que permite que un cliente externo a la clase (es decir, la clase `CasoEstudioATM`; sección J.13) indique al ATM que se ejecute. El método `Ejecutar` de ATM (líneas 35-52) utiliza un ciclo infinito (líneas 38-51) para dar la bienvenida a un usuario repetidas veces, tratar de autenticar al usuario y, si la autenticación tiene éxito, permitir al usuario que realice transacciones. Una vez que un usuario autenticado realiza las transacciones deseadas y sale, el ATM se reinicia a sí mismo, muestra un mensaje de despedida y reinicia el proceso para el siguiente usuario. Aquí utilizamos un ciclo infinito para simular el hecho de que un ATM parece ejecutarse en forma continua hasta que el banco lo desconecta (una acción que está más allá del control del usuario). Un usuario del ATM puede salir del sistema, pero no puede apagar el ATM por completo.

Dentro del ciclo infinito del método `Ejecutar`, las líneas 41-45 hacen que el ATM dé la bienvenida repetidas veces y trate de autenticar al usuario, siempre y cuando éste no se haya autenticado ya (es decir, que la condición `!usuarioAutenticado` sea `true`). La línea 43 invoca al método `MostrarLineaMensaje` de la `pantalla` del ATM para mostrar un mensaje de bienvenida. Al igual que el método `MostrarMensaje` de `Pantalla` diseñado en el caso de estudio, el método `MostrarLineaMensaje` (declarado en las líneas 14-17 de la figura J.2) muestra un mensaje al usuario, pero este método también imprime en pantalla una nueva línea después de mostrar el mensaje. Agregamos este método durante la implementación para dar a los clientes de la clase `Pantalla` más control sobre la colocación de los mensajes a mostrar en pantalla. La línea 44 invoca al método utilitario `private AutenticarUsuario` de la clase ATM (declarado en las líneas 55-75) para tratar de autenticar al usuario.

Autenticación del usuario

Consultamos el documento de requerimientos para determinar los pasos necesarios para autenticar al usuario antes de permitir que ocurran transacciones. La línea 58 del método `AutenticarUsuario` invoca al método `MostrarMensaje` de la `pantalla` del ATM para pedir al usuario que introduzca un número de cuenta. La línea 59 invoca al método `ObtenerEntrada` del `teclado` del ATM para obtener la entrada del usuario, y después almacena este entero en la variable local `numeroCuenta`. A continuación, el método `AutenticarUsuario` pide al usuario que introduzca un NIP (línea 62) y lo almacena en la variable `nip` (línea 63). Después, las líneas 66-67 tratan de autenticar al usuario enviando el `numeroCuenta` y `nip` introducidos por el usuario al método `AutenticarUsuario` de `baseDatosBanco`. La clase ATM establece su atributo `usuarioAutenticado` con el valor `bool` devuelto por este método; `usuarioAutenticado` se vuelve `true` si la autenticación tiene éxito (es decir, que el `nombreUsuario` y el `nip` coincidan con los de una `Cuenta` existente en la `baseDatosBanco`) y permanece `false` en caso contrario. Si `usuarioAutenticado` es `true`, la línea 71 guarda el número de cuenta introducido por el usuario (es decir, `numeroCuenta`) en el atributo `numeroCuentaActual` de ATM. Los demás métodos de la clase ATM utilizan esta variable cada vez que una sesión con el ATM requiere acceso al número de cuenta del usuario. Si `usuarioAutenticado` es `false`, las líneas 73-74 llaman al método `MostrarLineaMensaje` de `pantalla` para indicar que se introdujo un número de cuenta y/o NIP inválidos, por lo que el usuario debe intentar otra vez. Observe que establecemos `numeroCuentaUsuario` sólo después de autenticar el número de cuenta del usuario y su NIP asociado; si la base de datos no puede autenticar al usuario, `numeroCuentaUsuario` permanece en 0.

Una vez que el método `Ejecutar` trata de autenticar al usuario (línea 44), si `usuarioAutenticado` sigue siendo `false` (línea 41), el cuerpo del ciclo `while` (líneas 41-45) se ejecuta otra vez. Si `usuarioAutenticado`

es ahora true, el ciclo termina y el control continúa con la línea 47, en donde se hace una llamada al método utilitario `private RealizarTransacciones` de la clase `ATM`.

Realización de transacciones

El método `RealizarTransacciones` (líneas 78-111) lleva a cabo una sesión con el ATM para un usuario autenticado. La línea 80 declara la variable local `Transaccion`, a la cual asignamos un objeto `SolicitudSaldo`, `Retiro` o `Deposito` que representa la transacción con el ATM que se está procesando en esos momentos. Observe que aquí utilizamos una variable `Transaccion` para poder aprovechar el polimorfismo. Además, nombramos a esta variable con base en el nombre del rol incluido en el diagrama de clases de la figura 4.21: `transaccionActual`. La línea 81 declara otra variable local: una variable `bool` llamada `usuarioSalio`, la cual contiene información acerca de si el usuario ha elegido salir de su sesión con el ATM. Esta variable controla un ciclo `while` (líneas 84-110) que permite al usuario ejecutar un número ilimitado de transacciones, antes de que elija la opción para salir. Dentro de este ciclo, la línea 87 muestra el menú principal y obtiene la selección del menú que hizo el usuario, llamando al método utilitario `Mostrar MenuPrincipal` de `ATM` (declarado en las líneas 114-123). Este método muestra el menú principal invocando a los métodos de la `pantalla` del ATM, y devuelve una selección del menú que obtiene del usuario a través del teclado del ATM. La línea 87 almacena la selección del usuario devuelta por `MostrarMenuPrincipal`, en la variable local `seleccionMenuPrincipal`.

Después de obtener una selección del menú principal el método `RealizarTransacciones` utiliza una instrucción `switch` (líneas 90-109) para responder a la selección en forma apropiada. Si `seleccionMenuPrincipal` es igual al valor subyacente de cualquiera de los tres miembros de `enum` que representan tipos de transacciones (es decir, si el usuario eligió realizar una transacción), las líneas 97-98 llaman al método utilitario `CrearTransaccion` (declarado en las líneas 126-149) para devolver un objeto recién instanciado del tipo que corresponde a la transacción seleccionada. A la variable `transaccionActual` se le asigna la referencia devuelta por el método `CrearTransaccion`, y después la línea 99 invoca al método `Ejecutar` de esta transacción para ejecutarla. En breve hablaremos sobre el método `Ejecutar` de `Transaccion` y las tres clases derivadas de `Transaccion`. Observe que asignamos a la variable `transaccionActual` de la clase `Transaccion` un objeto de una de las tres clases derivadas de `Transaccion`, para poder ejecutar transacciones. Por ejemplo, si el usuario elige realizar una solicitud de saldo, (`OpcionMenu`) `seleccionMenuPrincipal` (línea 90) coincide con la etiqueta `OpcionMenu.SOLICITUD_SALDO` de la instrucción `case`, y `CrearTransaccion` devuelve un objeto `SolicitudSaldo` (líneas 97-98). Por ende, `transaccionActual` hace referencia a un objeto `SolicitudSaldo` y la invocación de `transaccionActual.Ejecutar()` (línea 99) resulta en una llamada mediante el polimorfismo a la versión de `Ejecutar` de `SolicitudSaldo`.

Creación de transacciones

El método `CrearTransaccion` (líneas 126-149) utiliza una instrucción `switch` (líneas 131-146) para instanciar un nuevo objeto de la clase derivada `Transaccion`, del tipo indicado por el parámetro `tipo`. Recuerde que el método `RealizarTransacciones` pasa el valor de `seleccionMenuPrincipal` al método `CrearTransaccion` sólo cuando `seleccionMenuPrincipal` contiene un valor que corresponde a uno de los tres tipos de transacciones. Entonces, el parámetro `tipo` (línea 126) recibe uno de los siguientes valores: `OpcionMenu.SOLICITUD_SALDO`, `OpcionMenu.RETIRO` u `OpcionMenu.DEPOSITO`. Cada `case` en la instrucción `switch` crea una instancia de un nuevo objeto, llamando al constructor apropiado de la clase derivada de `Transaccion`. Observe que cada constructor tiene una lista de parámetros única, basada en los datos especificados que se requieren para inicializar el objeto de la clase derivada. Un objeto `SolicitudSaldo` (líneas 135-136) requiere sólo el número de cuenta del usuario actual, y referencias a la `pantalla` y la `baseDatosBanco` del ATM. Además de estos parámetros, un objeto `Retiro` (líneas 139-140) requiere referencias al teclado y `dispensadorEfectivo` del ATM, y un objeto `Deposito` (líneas 143-144) requiere referencias al teclado y la `ranuraDeposito` del ATM. En las secciones J.9-J.12 hablaremos sobre las clases de transacciones con detalle.

Después de ejecutar una transacción (línea 99 en el método `RealizarTransacciones`), `usuarioSalio` sigue siendo `false` y se repite el ciclo `while` en las líneas 84-110, con lo cual el usuario regresa al menú principal. No obstante, si un usuario no realiza una transacción y selecciona la opción del menú principal para salir, la línea 103 establece `usuarioSalio` a `true`, haciendo que la condición en la línea 84 del ciclo `while` (`!usuarioSalio`) sea `false`. Esta instrucción `while` es la instrucción final del método `RealizarTransacciones`, por lo que el control regresa a la línea 47 del método `Ejecutar` que hizo la llamada. Si el usuario introduce una selección inválida del menú principal (por decir, un número que no sea entero en el rango de 1 a 4), las líneas 106-107 muestran

un mensaje de error apropiado, `usuarioSalio` sigue siendo `false` (como se estableció en la línea 81) y el usuario regresa al menú principal para intentar de nuevo.

Cuando el método `RealizarTransacciones` devuelve el control al método `Ejecutar`, el usuario ha elegido salir del sistema, por lo que las líneas 48-49 restablecen los atributos `usuarioAutenticado` y `numeroCuentaActual` del ATM a `false` y 0, respectivamente, como parte del proceso de preparación para el siguiente usuario del ATM. La línea 50 muestra un mensaje de despedida para el usuario actual, antes de que el ATM dé la bienvenida al siguiente usuario.

J.3 La clase Pantalla

La clase `Pantalla` (figura J.2) representa la pantalla del ATM y encapsula todos los aspectos relacionados con el proceso de mostrar los resultados al usuario. La clase `Pantalla` simula a la pantalla de un ATM real con el monitor de la computadora, e imprime en pantalla mensajes de texto mediante el uso de los métodos de salida de consola estándar `Console.Write` y `Console.WriteLine`. En la parte de diseño de este caso de estudio, equipamos a la clase `Pantalla` con una operación: `MostrarMensaje`. Para obtener una mayor flexibilidad al mostrar mensajes en la `Pantalla`, ahora declaramos tres métodos de `Pantalla`: `MostrarMensaje`, `MostrarLineaMensaje` y `MostrarMontoEnDolares`.

El método `MostrarMensaje` (líneas 8-11) recibe un objeto `string` como argumento y lo imprime en la pantalla mediante `Console.Write`. El cursor permanece en la misma línea, por lo cual este método es apropiado para mostrar indicadores al usuario. El método `MostrarLineaMensaje` (líneas 14-17) hace lo mismo usando `Console.WriteLine`, que imprime una nueva línea para desplazar el cursor a la siguiente línea. Por último, el método `MostrarMontoEnDolares` (líneas 20-23) imprime en pantalla un monto en dólares con el formato apropiado (por ejemplo, \$1,234.56). La línea 22 utiliza el método `Console.Write` para imprimir en pantalla un valor `decimal`, con formato de moneda y signo de dólares, dos lugares decimales y comas para mejorar la legibilidad de los montos grandes en dólares.

J.4 La clase Teclado

La clase `Teclado` (figura J.3) representa el teclado del ATM, y es responsable de recibir toda la entrada por parte del usuario. Recuerde que estamos simulando este hardware, por lo que utilizamos el teclado de la computadora

```

1  // Pantalla.cs
2  // Representa la pantalla del ATM.
3  using System;
4
5  public class Pantalla
6  {
7      // muestra un mensaje sin un retorno de carro al final
8      public void MostrarMensaje( string mensaje )
9      {
10         Console.Write( mensaje );
11     } // fin del método MostrarMensaje
12
13     // muestra un mensaje con un retorno de carro al final
14     public void MostrarLineaMensaje( string mensaje )
15     {
16         Console.WriteLine( mensaje );
17     } // fin del método MostrarLineaMensaje
18
19     // muestra un monto en dólares
20     public void MostrarMontoEnDolares( decimal monto )
21     {
22         Console.Write( "{0:C}", monto );
23     } // fin del método MostrarMontoEnDolares
24 } // fin de la clase Pantalla

```

Figura J.2 | La clase `Pantalla` representa la pantalla del ATM.

```

1 // Teclado.cs
2 // Representa el teclado del ATM.
3 using System;
4
5 public class Teclado
6 {
7     // devuelve un valor entero introducido por el usuario
8     public int ObtenerEntrada()
9     {
10         return Convert.ToInt32( Console.ReadLine() );
11     } // fin del método ObtenerEntrada
12 } // fin de la clase Teclado

```

Figura J.3 | La clase `Teclado` representa el teclado del ATM.

para simular el teclado del ATM. Utilizamos el método `Console.ReadLine` para obtener la entrada del usuario a través del teclado. Un teclado de computadora contiene muchas teclas que no se encuentran en el teclado del ATM. Aquí vamos a suponer que el usuario sólo oprime las teclas en el teclado de computadora que aparecen también en el teclado del ATM: los números del 0 al 9 y la tecla *Intro*.

El método `ObtenerEntrada` (líneas 8-11) invoca al método `ToInt32` de `Convert` para convertir la entrada devuelta por `Console.ReadLine` (línea 10) en un valor `int`. [*Nota:* el método `ToInt32` puede lanzar una excepción `FormatException` si el usuario introduce datos que no sean enteros. Como el teclado real del ATM sólo permite introducir enteros, vamos a suponer que no ocurrirán excepciones. En el capítulo 12, Manejo de excepciones, podrá consultar información acerca de cómo atrapar y procesar las excepciones.] Recuerde que `ReadLine` obtiene toda la entrada que utiliza el ATM. El método `ObtenerEntrada` de la clase `Teclado` sólo devuelve los datos enteros introducidos por el usuario. Si un cliente de la clase `Teclado` requiere entrada que cumpla con ciertos criterios específicos (por decir, un número que corresponda a una opción válida del menú), el cliente debe realizar la comprobación de errores apropiada.

J.5 La clase `DispensadorEfectivo`

La clase `DispensadorEfectivo` (figura J.4) representa el dispensador de efectivo del ATM. La línea 6 declara la constante `CUENTA_INICIAL`, la cual indica el número de billetes de \$20 que contiene el dispensador de efectivo cuando el ATM inicia sus operaciones (es decir, 500). La línea 7 implementa el atributo `cuentaBilletes` (modelado en la figura 11.22), que lleva la cuenta del número de billetes restantes en el `DispensadorEfectivo`, en cualquier momento dado. El constructor (líneas 10-13) establece `cuentaBilletes` con la cuenta inicial.

```

1 // DispensadorEfectivo.cs
2 // Representa el dispensador de efectivo del ATM.
3 public class DispensadorEfectivo
4 {
5     // el número inicial predeterminado de billetes en el dispensador de efectivo
6     private const int CUENTA_INICIAL = 500;
7     private int cuentaBilletes; // número restante de billetes de $20
8
9     // constructor sin parámetros que inicializa cuentaBilletes con CUENTA_INICIAL
10    public DispensadorEfectivo()
11    {
12        cuentaBilletes = CUENTA_INICIAL; // establece cuentaBilletes a CUENTA_INICIAL
13    } // fin del constructor
14
15    // simula dispensar el monto especificado de efectivo
16    public void DispensarEfectivo( decimal monto )

```

Figura J.4 | La clase `DispensadorEfectivo` representa el dispensador de efectivo del ATM. (Parte 1 de 2).

```

17  {
18      // número requerido de billetes de $20
19      int billetesRequeridos = ( ( int ) monto ) / 20;
20      cuentaBilletes -= billetesRequeridos;
21  } // fin del método DispensarEfectivo
22
23      // indica si el dispensador de efectivo puede dispensar el monto deseado
24  public bool HaySuficienteEfectivoDisponible( decimal monto )
25  {
26      // número requerido de billetes de $20
27      int billetesRequeridos = ( ( int ) monto ) / 20;
28
29      // devuelve si hay suficientes billetes disponibles
30      return ( cuentaBilletes >= billetesRequeridos );
31  } // fin del método HaySuficienteEfectivoDisponible
32  } // fin de la clase DispensadorEfectivo

```

Figura J.4 | La clase `DispensadorEfectivo` representa el dispensador de efectivo del ATM. (Parte 2 de 2).

[Nota: vamos a suponer que el proceso de agregar más billetes al `DispensadorEfectivo` y actualizar el valor de `cuentaBilletes` ocurre fuera del sistema ATM.] La clase `DispensadorEfectivo` tiene dos métodos `public`: `DispensarEfectivo` (líneas 16-21) y `HaySuficienteEfectivoDisponible` (líneas 24-31). La clase confía en que un cliente (por decir, `Retiro`) llama al método `DispensarEfectivo` sólo después de establecer que hay suficiente efectivo disponible, mediante una llamada al método `HaySuficienteEfectivoDisponible`. Por lo tanto, `DispensarEfectivo` simula el proceso de dispensar el monto solicitado de efectivo sin comprobar que haya suficiente efectivo disponible.

El método `HaySuficienteEfectivoDisponible` (líneas 24-31) tiene un parámetro llamado `monto` que especifica el monto de efectivo en cuestión. La línea 27 calcula el número de billetes de \$20 requeridos para dispensar el monto especificado. El ATM permite al usuario elegir sólo montos de retiro que sean múltiplos de \$20, por lo que convertimos a `monto` en un valor entero y lo dividimos entre 20 para obtener el número de `billetesRequeridos`. La línea 30 devuelve `true` si la `cuentaBilletes` del `DispensadorEfectivo` es mayor o igual que `billetesRequeridos` (es decir, que haya suficientes billetes disponibles) y devuelve `false` en caso contrario (es decir, que no haya suficientes billetes). Por ejemplo, si un usuario desea retirar \$80 (es decir, `billetesRequeridos` es 4) pero sólo quedan tres billetes (`cuentaBilletes` = 3), el método devuelve `false`.

El método `DispensarEfectivo` (líneas 16-21) simula el proceso de dispensar el efectivo. Si nuestro sistema estuviera conectado a un dispensador de efectivo de hardware real, este método interactuaría con el dispositivo de hardware para dispensar físicamente el efectivo. Nuestra versión simulada del método simplemente resta a la cuenta de billetes restantes (`cuentaBilletes`) el número requerido para dispensar el monto especificado (línea 20). Hay que tener en cuenta que es responsabilidad del cliente de la clase (es decir, `Retiro`) informar al usuario que se ha dispensado el efectivo; `DispensadorEfectivo` no interactúa directamente con la pantalla.

J.6 La clase `RanuraDeposito`

La clase `RanuraDeposito` (figura J.5) representa la ranura de depósito del ATM. Esta clase simula la funcionalidad de una ranura de depósito de hardware real. `RanuraDeposito` no tiene atributos y sólo tiene un método: `SeRecibioSobreDeposito` (líneas 7-10), el cual indica si se recibió o no un sobre de depósito.

En el documento de requerimientos vimos que el ATM permite al usuario hasta dos minutos para insertar un sobre. La versión actual del método `SeRecibioSobreDeposito` simplemente devuelve `true` de inmediato (línea 9), ya que ésta es sólo una simulación de software, por lo que se asume que el usuario inserta un sobre dentro del rango de tiempo requerido. Si se conectara una ranura de depósito real en nuestro sistema, el método `SeRecibioSobreDeposito` se implementaría de manera que esperara un máximo de dos minutos para recibir una señal de la ranura de depósito de hardware, indicando que el usuario verdaderamente ha insertado un sobre de depósito. Si `SeRecibioSobreDeposito` fuera a recibir dicha señal dentro de un plazo no mayor a dos minutos, el método devolvería `true`. Si pasaran dos minutos y el método no hubiera recibido una señal, entonces devolvería `false`.

```

1  // RanuraDeposito.cs
2  // Representa la ranura de depósito del ATM.
3  public class RanuraDeposito
4  {
5      // indica si se recibió el sobre (siempre devuelve verdadero, ya que
6      // ésta es sólo una simulación de software de una ranura de depósito real)
7      public bool SeRecibioSobreDeposito()
8      {
9          return true; // El sobre fue recibido
10     } // fin del método SeRecibioSobreDeposito
11 } // fin de la clase RanuraDeposito

```

Figura J.5 | La clase `RanuraDeposito` representa la ranura de depósito del ATM.

J.7 La clase Cuenta

La clase `Cuenta` (figura J.6) representa una cuenta bancaria. Cada `Cuenta` tiene cuatro atributos (que se modelan en la figura 11.22): `numeroCuenta`, `nip`, `saldoDisponible` y `saldoTotal`. Las líneas 5-8 implementan estos atributos como variables de instancia `private`. Para cada una de las variables de instancia `numeroCuenta`, `saldoDisponible` y `saldoTotal`, proporcionamos una propiedad con el mismo nombre que el atributo, pero empieza con mayúscula. Por ejemplo, la propiedad `NumeroCuenta` corresponde al atributo `numeroCuenta` modelado en la figura 11.22. Los clientes de esa clase no necesitan modificar la variable de instancia `numeroCuenta`, por lo cual `NumeroCuenta` se declara como una propiedad de sólo lectura (es decir, sólo proporciona un descriptor de acceso `get`).

```

1  // Cuenta.cs
2  // La clase Cuenta representa una cuenta bancaria.
3  public class Cuenta
4  {
5      private int numeroCuenta; // número de cuenta
6      private int nip; // NIP para autenticación
7      private decimal saldoDisponible; // monto disponible para retiro
8      private decimal saldoTotal; // fondos disponibles + depósito pendiente
9
10     // el constructor de cuatro parámetros inicializa los atributos
11     public Cuenta( int elNumeroCuenta, int elNIP,
12         decimal elSaldoDisponible, decimal elSaldoTotal )
13     {
14         numeroCuenta = elNumeroCuenta;
15         nip = elNIP;
16         saldoDisponible = elSaldoDisponible;
17         saldoTotal = elSaldoTotal;
18     } // fin del constructor
19
20     // propiedad de sólo lectura que obtiene el número de cuenta
21     public int NumeroCuenta
22     {
23         get
24         {
25             return numeroCuenta;
26         } // fin de get
27     } // fin de la propiedad NumeroCuenta
28
29     // propiedad de sólo lectura que obtiene el saldo disponible
30     public decimal SaldoDisponible

```

Figura J.6 | La clase `Cuenta` representa una cuenta bancaria. (Parte I de 2).

```

31     {
32         get
33         {
34             return saldoDisponible;
35         } // fin de get
36     } // fin de la propiedad SaldoDisponible
37
38     // propiedad de sólo lectura que obtiene el saldo total
39     public decimal SaldoTotal
40     {
41         get
42         {
43             return saldoTotal;
44         } // fin de get
45     } // fin de la propiedad SaldoTotal
46
47     // determina si un NIP especificado por el usuario coincide con un NIP en Cuenta
48     public bool ValidarNIP( int NIPUsuario )
49     {
50         return ( NIPUsuario == nip );
51     } // fin del método ValidarNIP
52
53     // abona a la cuenta (los fondos no se han verificado todavía)
54     public void Abonar( decimal monto )
55     {
56         saldoTotal += monto; // lo suma al saldo total
57     } // fin del método Abonar
58
59     // carga a la cuenta
60     public void Cargar( decimal monto )
61     {
62         saldoDisponible -= monto; // lo resta del saldo disponible
63         saldoTotal -= monto; // lo resta del saldo total
64     } // fin del método Cargar
65 } // fin de la clase Cuenta

```

Figura J.6 | La clase Cuenta representa una cuenta bancaria. (Parte 2 de 2).

La clase Cuenta tiene un constructor (líneas 11-18) que recibe como argumentos un número de cuenta, el NIP establecido para la cuenta, el saldo inicial disponible y el saldo total inicial. Las líneas 14-17 asignan estos valores a los atributos de la clase (es decir, las variables de instancia). Observe que, por lo general, los objetos Cuenta se crean de manera externa al sistema ATM. No obstante, en esta simulación, los objetos Cuenta se crean dentro de la clase BaseDatosBanco (figura J.7).

Propiedades public de sólo lectura de la clase Cuenta

La propiedad de sólo lectura NumeroCuenta (líneas 21-27) provee el acceso a la variable de instancia numeroCuenta de una Cuenta. Incluimos esta propiedad en nuestra implementación para que un cliente de la clase (por ejemplo, BaseDatosBanco) pueda identificar una Cuenta específica. Por ejemplo, BaseDatosBanco contiene muchos objetos Cuenta y puede acceder a esta propiedad en cada uno de sus objetos Cuenta, para localizar el que tiene un número de cuenta específico.

Las propiedades de sólo lectura SaldoDisponible (líneas 30-36) y SaldoTotal (líneas 39-45) permiten a los clientes obtener los valores de las variables de instancia `private decimal` llamadas `saldoDisponible` y `saldoTotal`, respectivamente. La propiedad SaldoDisponible representa el monto de los fondos disponibles para retiro. La propiedad SaldoTotal representa el monto de los fondos disponibles, más el monto de los fondos depositados cuya confirmación está pendiente, tanto de efectivo como de cheques en los sobres de depósito.

Métodos public de la clase Cuenta

El método `ValidarNIP` (líneas 48-51) determina si un NIP especificado por el usuario (es decir, el parámetro `NIPUsuario`) coincide con el NIP asociado con la cuenta (es decir, el atributo `nip`). Recuerde que modelamos el parámetro `NIPUsuario` de este método en el diagrama de clases UML de la figura 7.23. Si los dos NIPs coinciden, el método devuelve `true`; en caso contrario, devuelve `false`.

El método `Abonar` (líneas 54-57) suma un monto de dinero (es decir, el parámetro `monto`) a una `Cuenta` como parte de una transacción de depósito. Observe que este método suma el `monto` sólo a la variable de instancia `saldoTotal` (línea 56). El dinero abonado a una cuenta durante un depósito no se hace disponible de inmediato, por lo que sólo modificamos el saldo total. Estamos suponiendo que el banco actualiza el saldo disponible en forma apropiada una vez transcurrido el tiempo necesario para verificar el monto de efectivo en el sobre de depósito, junto con los cheques. Nuestra implementación de la clase `Cuenta` sólo incluye los métodos requeridos para llevar a cabo transacciones con el ATM. Por lo tanto, omitimos los métodos que invocaría algún otro sistema bancario para sumar un monto a la variable de instancia `saldoDisponible` para confirmar un depósito, o restar un monto al atributo `saldoTotal` para rechazar un depósito.

El método `Cargar` (líneas 60-64) resta un monto de dinero (es decir, el parámetro `monto`) de una `Cuenta`, como parte de una transacción de retiro. Este método resta el `monto` tanto de la variable de instancia `saldoDisponible` (línea 62), como de la variable de instancia `saldoTotal` (línea 63), ya que un retiro afecta en ambos saldos.

J.8 La clase BaseDatosBanco

La clase `BaseDatosBanco` (figura J.7) modela la base de datos del banco con la que el ATM interactúa para acceder a la información de la cuenta de un usuario y modificarla. Determinamos un atributo de tipo de referencia para la clase `BaseDatosBanco`, con base en su relación de composición con la clase `Cuenta`. En la figura 11.21 vimos que una `BaseDatosBanco` está compuesta por cero o más objetos de la clase `Cuenta`. La línea 5 declara el atributo `cuentas` (un arreglo que almacena objetos `Cuenta`) para implementar esta relación de composición. La clase `BaseDatosBanco` tiene un constructor sin parámetros (líneas 8-15) que inicializa `cuentas` con nuevos objetos `Cuenta` (líneas 13-14). Observe que el constructor de `Cuenta` (figura J.6, líneas 11-18) tiene cuatro parámetros: el número de cuenta, el NIP asignado a esa cuenta, el saldo disponible inicial y el saldo total inicial.

```

1  // BaseDatosBanco.cs
2  // Representa la base de datos de información de las cuentas bancarias.
3  public class BaseDatosBanco
4  {
5      private Cuenta[] cuentas; // arreglo de las cuentas bancarias
6
7      // el constructor sin parámetros inicializa las cuentas
8      public BaseDatosBanco()
9      {
10         // crea dos objetos Cuenta para prueba y
11         // los coloca en el arreglo cuentas
12         cuentas = new Cuenta[ 2 ]; // crea el arreglo cuentas
13         cuentas[ 0 ] = new Cuenta( 12345, 54321, 1000.00M, 1200.00M );
14         cuentas[ 1 ] = new Cuenta( 98765, 56789, 200.00M, 200.00M );
15     } // fin del constructor
16
17     // obtiene un objeto Cuenta que contiene el número de cuenta especificado
18     private Cuenta ObtenerCuenta( int numeroCuenta )
19     {
20         // itera a través de cuentas, buscando un número de cuenta que coincida
21         foreach ( Cuenta cuentaActual in cuentas )

```

Figura J.7 | La clase `BaseDatosBanco` representa la base de datos de información de cuentas bancarias. (Parte 1 de 2).

```

22     {
23         if ( cuentaActual.NumeroCuenta == numeroCuenta )
24             return cuentaActual;
25     } // fin de foreach
26
27     // no se encontró la cuenta
28     return null;
29 } // fin del método ObtenerCuenta
30
31 // determina si el número de cuenta y el NIP especificados por el usuario
32 // coinciden con los de una cuenta en la base de datos
33 public bool AutenticarUsuario( int numeroCuentaUsuario, int NIPUsuario)
34 {
35     // trata de obtener la cuenta con el número de cuenta
36     Cuenta cuentaUsuario = ObtenerCuenta( numeroCuentaUsuario );
37
38     // si la cuenta existe, devuelve el resultado de la función ValidarNIP de Cuenta
39     if ( cuentaUsuario != null )
40         return cuentaUsuario.ValidarNIP( NIPUsuario ); // verdadero si coincide
41     else
42         return false; // no se encontró el número de cuenta, por lo que devuelve falso
43 } // fin del método AutenticarUsuario
44
45 // devuelve el saldo disponible de la Cuenta con el número de cuenta especificado
46 public decimal ObtenerSaldoDisponible( int numeroCuentaUsuario )
47 {
48     Cuenta cuentaUsuario = ObtenerCuenta( numeroCuentaUsuario );
49     return cuentaUsuario.SaldoDisponible;
50 } // fin del método ObtenerSaldoDisponible
51
52 // devuelve el saldo total de la Cuenta con el número de cuenta especificado
53 public decimal ObtenerSaldoTotal( int numeroCuentaUsuario )
54 {
55     Cuenta cuentaUsuario = ObtenerCuenta(numeroCuentaUsuario);
56     return cuentaUsuario.SaldoTotal;
57 } // fin del método ObtenerSaldoTotal
58
59 // abona a la Cuenta con el número de cuenta especificado
60 public void Abonar( int numeroCuentaUsuario, decimal monto )
61 {
62     Cuenta cuentaUsuario = ObtenerCuenta( numeroCuentaUsuario );
63     cuentaUsuario.Abonar( monto );
64 } // fin del método Abonar
65
66 // carga a la Cuenta con el número de cuenta especificado
67 public void Cargar( int numeroCuentaUsuario, decimal monto )
68 {
69     Cuenta cuentaUsuario = ObtenerCuenta( numeroCuentaUsuario );
70     cuentaUsuario.Cargar( monto );
71 } // fin del método Cargar
72 } // fin de la clase BaseDatosBanco

```

Figura J.7 | La clase BaseDatosBanco representa la base de datos de información de cuentas bancarias. (Parte 2 de 2).

Recuerde que la clase BaseDatosBanco sirve como intermediario entre la clase ATM y los objetos Cuenta que contienen información de las cuentas de usuarios. Por ende, los métodos de la clase BaseDatosBanco invocan a los métodos y propiedades correspondientes del objeto Cuenta que pertenece al usuario actual del ATM.

Método utilitario private ObtenerCuenta

Incluimos el método utilitario `private ObtenerCuenta` (líneas 18-29) para permitir que el objeto `BaseDatosBanco` obtenga una referencia a una Cuenta específica dentro del arreglo cuentas. Para localizar la Cuenta del usuario, `BaseDatosBanco` compara el valor devuelto por la propiedad `NumeroCuenta` para cada elemento de cuentas con un número de cuenta especificado, hasta que encuentra una coincidencia. Las líneas 21-25 recorren el arreglo cuentas. Si el número de cuenta de `cuentaActual` es igual al valor del parámetro `numeroCuenta`, el método devuelve `cuentaActual`. Si no hay una cuenta con el número especificado, entonces la línea 28 devuelve `null`.

Métodos public

El método `AutenticarUsuario` (líneas 33-43) prueba o desaprueba la identidad de un usuario del ATM. Este método recibe como argumentos un número de cuenta y un NIP especificados por el usuario, y determina si coinciden con el número de cuenta y NIP de una Cuenta en la base de datos. La línea 36 llama al método `ObtenerCuenta`, el cual devuelve una Cuenta con `numeroCuentaUsuario` como su número de cuenta, o `null` para indicar que el `numeroCuentaUsuario` es inválido. Si `ObtenerCuenta` devuelve un objeto Cuenta, la línea 40 devuelve el valor `bool` devuelto por el método `ValidarNIP` de ese objeto. Observe que el método `AutenticarUsuario` de `BaseDatosBanco` no realiza la comparación de NIPs, sino que envía el `NIPUsuario` al método `ValidarNIP` del objeto Cuenta para hacerlo. El valor devuelto por el método `ValidarNIP` de Cuenta (línea 40) indica si el NIP especificado por el usuario coincide con el NIP de la Cuenta del usuario, por lo que el método `AutenticarUsuario` sólo devuelve este valor (línea 40) al cliente de la clase (es decir, ATM).

La `BaseDatosBanco` confía en que el ATM invoque al método `AutenticarUsuario` y reciba un valor de retorno `true` para permitir que el usuario realice transacciones. `BaseDatosBanco` también confía en que cada objeto `Transaccion` creado por el ATM contiene el número de cuenta válido del usuario actual autenticado, y que este número de cuenta se pasa al resto de los métodos de `BaseDatosBanco` como el argumento `numeroCuentaUsuario`. Por lo tanto, los métodos `ObtenerSaldoDisponible` (líneas 46-50), `ObtenerSaldoTotal` (líneas 53-57), `Abonar` (líneas 60-64) y `Cargar` (líneas 67-71) simplemente obtienen el objeto Cuenta del usuario mediante el método utilitario `ObtenerCuenta` y después invocan al método de Cuenta apropiado en ese objeto. Sabemos que las llamadas a `ObtenerCuenta` dentro de estos métodos nunca devolverán `null`, ya que `numeroCuentaUsuario` debe hacer referencia a una Cuenta existente. Observe que `ObtenerSaldoDisponible` y `ObtenerSaldoTotal` devuelven los valores devueltos por las correspondientes propiedades de Cuenta. Observe además que los métodos `Abonar` y `Cargar` simplemente redirigen el parámetro `monto` hacia los métodos de Cuenta que invocan.

J.9 La clase Transaccion

La clase `Transaccion` (figura J.8) es una clase base abstract que representa la noción de una transacción con el ATM. Contiene las características comunes de las clases derivadas `SolicitudSaldo`, `Retiro` y `Deposito`. Esta clase parte del “esqueleto” de código que se desarrolló por primera vez en la sección 11.9. La línea 3 declara esta clase como abstract. Las líneas 5-7 declaran las variables de instancia `private` de la clase. En el diagrama de clases de la figura 11.22 vimos que la clase `Transaccion` contiene la propiedad `NumeroCuenta` que indica la cuenta involucrada en la Transacción. La línea 5 implementa la variable de instancia `numeroCuenta` para mantener los datos de la propiedad `NumeroCuenta`. Derivamos los atributos `pantalla` (que se implementa como la variable de instancia `pantallaUsuario` en la línea 6) y `baseDatosBanco` (que se implementa como la variable de instancia `baseDatos` en la línea 7) de las asociaciones de la clase `Transaccion` que modelamos en la figura 11.21. Todas las transacciones requieren acceso a la pantalla del ATM y a la base de datos del banco.

```

1  // Transaccion.cs
2  // La clase base abstracta Transaccion representa una transacción en el ATM.
3  public abstract class Transaccion
4  {
5      private int numeroCuenta; // cuenta involucrada en la transacción
6      private Pantalla pantallaUsuario; // referencia a la pantalla del ATM
7      private BaseDatosBanco baseDatos; // referencia a la base de datos de información
        de cuentas

```

Figura J.8 | La clase base abstract `Transaccion` representa una transacción con el ATM. (Parte 1 de 2).

```

8
9 // constructor de tres parámetros invocado por las clases derivadas
10 public Transaccion( int cuentaUsuario, Pantalla laPantalla,
11     BaseDatosBanco laBaseDatos )
12 {
13     numeroCuenta = cuentaUsuario;
14     pantallaUsuario = laPantalla;
15     baseDatos = laBaseDatos;
16 } // fin del constructor
17
18 // propiedad de sólo lectura que obtiene el número de cuenta
19 public int NumeroCuenta
20 {
21     get
22     {
23         return numeroCuenta;
24     } // fin de get
25 } // fin de la propiedad NumeroCuenta
26
27 // propiedad de sólo lectura que obtiene la referencia a la pantalla
28 public Pantalla PantallaUsuario
29 {
30     get
31     {
32         return pantallaUsuario;
33     } // fin de get
34 } // fin de la propiedad PantallaUsuario
35
36 // propiedad de sólo lectura que obtiene la referencia a la baseDatos del banco
37 public BaseDatosBanco BaseDatos
38 {
39     get
40     {
41         return baseDatos;
42     } // fin de get
43 } // fin de la propiedad BaseDatos
44
45 // realiza la transacción (cada clase derivada lo redefine)
46 public abstract void Ejecutar(); // no hay implementación aquí
47 } // fin de la clase Transaccion

```

Figura J.8 | La clase base abstract Transaccion representa una transacción con el ATM. (Parte 2 de 2).

La clase Transaccion tiene un constructor (líneas 10-16) que recibe como argumentos el número de cuenta del usuario actual y las referencias a la pantalla del ATM y la base de datos del banco. Como Transaccion es una clase abstract (línea 3), este constructor nunca se llama directamente para instanciar objetos Transaccion. En vez de ello, los constructores de las clases derivadas de Transaccion invocan a este constructor, a través de sus inicializadores.

La clase Transaccion tiene tres propiedades public de sólo lectura: NumeroCuenta (líneas 19-25), PantallaUsuario (líneas 28-34) y BaseDatos (líneas 37-43). Las clases derivadas de Transaccion heredan estas propiedades y las utilizan para obtener accesos a las variables de instancia private de la clase Transaccion. Tenga en cuenta que elegimos los nombres de las propiedades PantallaUsuario y BaseDatos por cuestión de legibilidad; quisimos evitar nombres de propiedades que sean iguales que los nombres de las clases Pantalla y BaseDatosBanco, lo cual puede ser confuso.

La clase Transaccion también declara el método abstract Ejecutar (línea 46). No tiene sentido proveer una implementación para este método en la clase Transaccion, ya que no puede ejecutarse una transacción genérica. Por ende, declaramos este método como abstract, con lo cual obligamos a que cada clase derivada concreta de Transaccion proporcione su propia implementación que ejecute el tipo específico de transacción.

J.10 La clase SolicitudSaldo

La clase `SolicitudSaldo` (figura J.9) hereda de `Transaccion` y representa una transacción de solicitud de saldo con el ATM (línea 3). `SolicitudSaldo` no tiene atributos propios, pero hereda los atributos `numeroCuenta`, `pantalla` y `baseDatosBanco` de `Transaccion`, a los que puede acceder mediante las propiedades `public` de sólo lectura de `Transaccion`. El constructor de `SolicitudSaldo` (líneas 6-8) recibe los argumentos que corresponden a estos atributos y los envía al constructor de `Transaccion` mediante una invocación del inicializador del constructor con la palabra clave `base` (línea 8). El cuerpo del constructor está vacío.

La clase `SolicitudSaldo` redefine el método abstracto `Ejecutar` de `Transaccion` y proporciona una implementación concreta (líneas 11-27) que realiza los pasos involucrados en una solicitud de saldo. Las líneas 14-15 obtienen el saldo disponible de la Cuenta especificada mediante una invocación al método `ObtenerSaldoDisponible` de la propiedad `BaseDatos` heredada. Observe que la línea 15 utiliza la propiedad `NumeroCuenta` heredada para obtener el número de cuenta del usuario actual. La línea 18 obtiene el saldo total de la Cuenta especificada. Las líneas 21-26 muestran la información del saldo en la pantalla del ATM, usando la propiedad `PantallaUsuario` heredada. Recuerde que `MostrarMontoEnDolares` recibe un argumento `decimal` y lo imprime en la pantalla con el formato de monto en dólares, con el signo de dólar. Por ejemplo, si el saldo disponible de un usuario es 1000.50M, la línea 23 imprime \$1,000.50. Observe que la línea 26 inserta una línea en blanco como salida, para separar la información del saldo de la salida subsiguiente (es decir, el menú principal repetido por la clase ATM, después de ejecutar la transacción `SolicitudSaldo`).

J.11 La clase Retiro

La clase `Retiro` (figura J.10) extiende a `Transaccion` y representa una transacción de retiro con el ATM. Esta clase parte del “esqueleto” de código que se desarrolló en la figura 11.24. En el diagrama de clases de la figura 11.22 vimos que la clase `Retiro` tiene un atributo llamado `monto`, el cual se declara en la línea 5 como una variable de instancia `decimal`. La figura 11.21 modela las asociaciones entre la clase `Retiro` y las clases `TecLado`

```

1  // SolicitudSaldo.cs
2  // Representa una transacción de solicitud de saldo en el ATM
3  public class SolicitudSaldo : Transaccion
4  {
5      // el constructor de cinco parámetros inicializa las variables de la clase base
6      public SolicitudSaldo( int numeroCuentaUsuario,
7          Pantalla pantallaATM, BaseDatosBanco baseDatosBancoATM )
8          : base( numeroCuentaUsuario, pantallaATM, baseDatosBancoATM ) {}
9
10     // realiza una transacción; redefine el método abstracto de Transaccion
11     public override void Ejecutar()
12     {
13         // obtiene el saldo disponible para la Cuenta del usuario actual
14         decimal saldoDisponible =
15             BaseDatos.ObtenerSaldoDisponible( NumeroCuenta );
16
17         // obtiene el saldo total para la Cuenta del usuario actual
18         decimal saldoTotal = BaseDatos.ObtenerSaldoTotal( NumeroCuenta );
19
20         // muestra la información del saldo en la pantalla
21         PantallaUsuario.MostrarLineaMensaje( "\nInformación del saldo:" );
22         PantallaUsuario.MostrarMensaje( " - Saldo disponible: " );
23         PantallaUsuario.MostrarMontoEnDolares( saldoDisponible );
24         PantallaUsuario.MostrarMensaje( "\n - Saldo total: " );
25         PantallaUsuario.MostrarMontoEnDolares( saldoTotal );
26         PantallaUsuario.MostrarLineaMensaje( "" );
27     } // fin del método Ejecutar
28 } // fin de la clase SolicitudSaldo

```

Figura J.9 | La clase `SolicitudSaldo` representa una transacción de solicitud de saldo con el ATM.

y `DispensadorEfectivo`, para las cuales se implementan en las líneas 6-7 los atributos de referencia `teclado` y `dispensadorEfectivo`, respectivamente. La línea 10 declara una constante que corresponde a la opción para cancelar del menú.

El constructor de la clase `Retiro` (líneas 13-21) tiene cinco parámetros. Utiliza el inicializador del constructor para pasar los parámetros `numeroCuentaUsuario`, `pantallaATM` y `baseDatosBancoATM` al constructor de la clase base `Transaccion` y establecer los atributos que `Retiro` hereda de `Transaccion`. El constructor también recibe las referencias `tecladoATM` y `dispensadorEfectivoATM` como parámetros, y los asigna a los atributos de tipo de referencia `teclado` y `dispensadorEfectivo`, respectivamente.

```

1  // Retiro.cs
2  // La clase Retiro representa una transacción de retiro del ATM.
3  public class Retiro : Transaccion
4  {
5      private decimal monto; // monto a retirar
6      private Teclado teclado; // referencia a Teclado
7      private DispensadorEfectivo dispensadorEfectivo; // referencia al dispensador de
        efectivo
8
9      // constante que corresponde a la opción de menú para cancelar
10     private const int CANCELO = 6;
11
12     // constructor de cinco parámetros
13     public Retiro( int numeroCuentaUsuario, Pantalla pantallaATM,
14         BaseDatosBanco baseDatosBancoATM, Teclado tecladoATM,
15         DispensadorEfectivo dispensadorEfectivoATM )
16         : base( numeroCuentaUsuario, pantallaATM, baseDatosBancoATM )
17     {
18         // inicializa las referencias a teclado y al dispensador de efectivo
19         teclado = tecladoATM;
20         dispensadorEfectivo = dispensadorEfectivoATM;
21     } // fin del constructor
22
23     // realiza una transacción, redefine el método abstracto de Transaccion
24     public override void Ejecutar()
25     {
26         bool efectivoDispensado = false; // el efectivo no se ha dispensado aún
27
28         // la transacción no se ha cancelado aún
29         bool transaccionCancelada = false;
30
31         // itera hasta que se dispense efectivo o el usuario cancele
32         do
33         {
34             // obtiene el monto de retiro elegido por el usuario
35             int seleccion = MostrarMenuDeMontos();
36
37             // comprueba si el usuario eligió un monto de retiro o canceló
38             if ( seleccion != CANCELO )
39             {
40                 // establece monto al monto seleccionado en dólares
41                 monto = seleccion;
42
43                 // obtiene el saldo disponible de la cuenta involucrada
44                 decimal saldoDisponible =
45                     BaseDatos.ObtenerSaldoDisponible( NumeroCuenta );
46

```

Figura J.10 | La clase `Retiro` representa una transacción de retiro en el ATM. (Parte I de 3).

```

47      // comprueba si el usuario tiene suficiente dinero en la cuenta
48      if ( monto <= saldoDisponible )
49      {
50          // comprueba si el dispensador de efectivo tiene suficiente dinero
51          if ( dispensadorEfectivo.HaySuficienteEfectivoDisponible( monto ) )
52          {
53              // carga a la cuenta para reflejar el retiro
54              BaseDatos.Cargar( NumeroCuenta, monto );
55
56              dispensadorEfectivo.DispensarEfectivo( monto ); // dispensa efectivo
57              efectivoDispensado = true; // se dispensó el efectivo
58
59              // instruye al usuario para que tome el efectivo
60              PantallaUsuario.MostrarLineaMensaje(
61                  "\nPor favor tome su efectivo del dispensador." );
62          } // fin del if más interno
63          else // el dispensador no tiene suficiente efectivo
64              PantallaUsuario.MostrarLineaMensaje(
65                  "\nNo hay suficiente efectivo disponible en el ATM." +
66                  "\n\nPor favor elija un monto más pequeño." );
67      } // fin del if intermedio
68      else // no hay suficiente dinero disponible en la cuenta del usuario
69          PantallaUsuario.MostrarLineaMensaje(
70              "\nNo hay suficiente efectivo disponible en su cuenta." +
71              "\n\nPor favor elija un monto más pequeño." );
72  } // fin del if más externo
73  else
74  {
75      PantallaUsuario.MostrarLineaMensaje( "\nCancelando la transacción..." );
76      transaccionCancelada = true; // el usuario canceló la transacción
77  } // fin del else
78  } while ( ( !efectivoDispensado ) && ( !transaccionCancelada ) );
79  } // fin del método Ejecutar
80
81  // muestra un menú de montos para retirar y la opción para cancelar;
82  // devuelve el monto elegido o 6 si el usuario elije cancelar
83  private int MostrarMenuDeMontos()
84  {
85      int eleccionUsuario = 0; // variable para almacenar el valor devuelto
86
87      // arreglo de montos que corresponden a los números del menú
88      int[] montos = { 0, 20, 40, 60, 100, 200 };
89
90      // itera mientras no se haya realizado una selección válida
91      while ( eleccionUsuario == 0 )
92      {
93          // muestra el menú
94          PantallaUsuario.MostrarLineaMensaje( "\nOpciones de retiro:" );
95          PantallaUsuario.MostrarLineaMensaje( "1 - $20" );
96          PantallaUsuario.MostrarLineaMensaje( "2 - $40" );
97          PantallaUsuario.MostrarLineaMensaje( "3 - $60" );
98          PantallaUsuario.MostrarLineaMensaje( "4 - $100" );
99          PantallaUsuario.MostrarLineaMensaje( "5 - $200" );
100         PantallaUsuario.MostrarLineaMensaje( "6 - Cancelar transacción" );
101         PantallaUsuario.MostrarMensaje(
102             "\nElija una opción de retiro (1-6): " );
103
104         // obtiene la entrada del usuario a través del teclado
105         int entrada = teclado.ObtenerEntrada();

```

Figura J.10 | La clase Retiro representa una transacción de retiro en el ATM. (Parte 2 de 3).

```

106
107 // determina cómo proceder con base en el valor de entrada
108 switch ( entrada )
109 {
110     // si el usuario eligió un monto de retiro (es decir, la opción
111     // 1, 2, 3, 4 o 5), devuelve el monto correspondiente
112     // del arreglo montos
113     case 1: case 2: case 3: case 4: case 5:
114         eleccionUsuario = montos[ entrada ]; // guarda la elección del usuario
115         break;
116     case CANCELLO: // el usuario eligió cancelar
117         eleccionUsuario = CANCELLO; // guarda la elección del usuario
118         break;
119     default:
120         PantallaUsuario.MostrarLineaMensaje(
121             "\nSelección inválida. Intente de nuevo." );
122         break;
123     } // fin de switch
124 } // fin de while
125
126 return eleccionUsuario;
127 } // fin del método MostrarMenuDeMontos
128 } // fin de la clase Retiro

```

Figura J.10 | La clase Retiro representa una transacción de retiro en el ATM. (Parte 3 de 3).

Redefinición del método abstract Ejecutar

La clase Retiro redefine el método abstract Ejecutar de Transaccion con una implementación concreta (líneas 24-79) que realiza los pasos involucrados en un retiro. La línea 26 declara e inicializa una variable local `bool` llamada `efectivoDispensado`. Esta variable indica si se ha dispensado efectivo (es decir, si la transacción se completó con éxito); al principio su valor es `false`. La línea 29 declara e inicializa en `false` una variable `bool` llamada `transaccionCancelada`, para indicar que la transacción aún no ha sido cancelada por el usuario.

Las líneas 32-78 contienen una instrucción `do...while` que ejecuta su cuerpo hasta que se dispensa efectivo (es decir, hasta que `efectivoDispensado` se vuelve `true`) o hasta que el usuario elige cancelar (es decir, hasta que `transaccionCancelada` se vuelva `true`). Utilizamos este ciclo para regresar al usuario en forma continua hasta el inicio de la transacción, si ocurre un error (es decir, si el monto de retiro solicitado es mayor que el saldo disponible del usuario, o mayor que el monto de efectivo en el dispensador). La línea 35 muestra un menú de montos de retiro y obtiene la selección del usuario mediante una llamada al método utilitario `private` `MostrarMenuDeMontos` (el cual se declara en las líneas 83-127). Este método muestra el menú de montos y devuelve un monto de retiro tipo `int` o una constante `int` llamada `CANCELLO`, para indicar que el usuario ha elegido cancelar la transacción.

Mostrar opciones con el método utilitario private MostrarMenuDeMontos

El método `MostrarMenuDeMontos` (líneas 83-127) declara primero la variable local `eleccionUsuario` (al principio es 0) para almacenar el valor que devolverá el método (línea 85). La línea 88 declara un arreglo entero de montos de retiro, que corresponden a los montos que se muestran en el menú de retiro. Ignoramos el primer elemento en el arreglo (índice 0), ya que el menú no tiene una opción 0. La instrucción `while` en las líneas 91-124 se repite hasta que `eleccionUsuario` recibe un valor distinto de 0. En breve veremos que esto ocurre cuando el usuario selecciona una opción válida del menú. Las líneas 94-102 muestran el menú de retiro en la pantalla y piden al usuario que introduzca una opción. La línea 105 obtiene el entero `entrada` desde el teclado. La instrucción `switch` en las líneas 108-123 determina cómo proceder, con base en la entrada del usuario. Si el usuario selecciona 1, 2, 3, 4 o 5, la línea 114 establece `eleccionUsuario` al valor del elemento en el arreglo `montos` con el índice `entrada`. Por ejemplo, si el usuario introduce 3 para retirar \$60, la línea 114 establece `eleccionUsuario` al valor de `montos[ 3 ]`; es decir, 60. La variable `eleccionUsuario` ya no es igual a 0, por lo que la instrucción `while` de las líneas 91-124 termina, y la línea 126 devuelve `eleccionUsuario`. Si el usuario selecciona la opción del menú para cancelar se

ejecuta la línea 117, con lo cual `eleccionUsuario` se establece a `CANCELLO` y el método devuelve este valor. Si el usuario no introduce una selección válida del menú, las líneas 120-121 muestran un mensaje de error y el usuario es devuelto al menú de retiro.

La instrucción `if` en la línea 38 del método `Ejecutar` determina si el usuario seleccionó un monto de retiro, o si optó por cancelar. Si el usuario cancela, la línea 75 muestra un mensaje apropiado al usuario, antes de regresar el control al método que hizo la llamada, el método `RealizarTransacciones` de `ATM`. Si el usuario eligió un monto de retiro, la línea 41 asigna la variable local `seleccion` a la variable de instancia `monto`. Las líneas 44-45 obtienen el saldo disponible de la Cuenta del usuario actual y la almacenan en una variable local `decimal` llamada `saldoDisponible`. A continuación, la instrucción `if` en la línea 48 determina si el monto seleccionado es menor o igual que el saldo disponible del usuario. Si no lo es, las líneas 69-71 muestran un mensaje de error. Después, el control continúa hasta el final de la instrucción `do...while` y el ciclo se repite, ya que tanto `efectivoDispensado` como `transaccionCancelada` siguen siendo `false`. Si el saldo del usuario es lo suficientemente alto, la instrucción `if` en la línea 51 determina si el dispensador de efectivo tiene suficiente dinero para satisfacer la solicitud de retiro, mediante una invocación al método `HaySuficienteEfectivoDisponible` de `dispensadorEfectivo`. Si este método devuelve `false`, las líneas 64-66 muestran un mensaje de error y se repite la instrucción `do...while`. Si hay suficiente efectivo disponible, se cumplen los requerimientos para el retiro y la línea 54 carga el monto a la cuenta del usuario en la base de datos. Después, las líneas 56-57 instruyen al dispensador de efectivo para que entregue el efectivo al usuario y establecen `efectivoDispensado` a `true`. Por último, las líneas 60-61 muestran un mensaje al usuario, para que tome el efectivo dispensado. Como ahora `efectivoDispensado` es `true`, el control continúa después de la instrucción `do...while`. Debido a que no aparecen instrucciones adicionales debajo del ciclo, el método devuelve el control a la clase `ATM`.

J.12 La clase Depósito

La clase `Deposito` (figura J.11) hereda de `Transaccion` y representa a una transacción de depósito con el ATM. En el diagrama de clases de la figura 11.22 vimos que la clase `Deposito` tiene un atributo llamado `monto`, que en la línea 5 se declara como variable de instancia `decimal`. Las líneas 6-7 crean los atributos de referencia `teclado` y `ranuraDeposito`, que implementan las asociaciones entre la clase `Deposito` y las clases `Tecclado` y `RanuraDeposito`, modeladas en la figura 11.21. La línea 10 declara una constante llamada `CANCELLO`, que corresponde al valor que introduce un usuario para cancelar una transacción de depósito.

La clase `Deposito` contiene un constructor (líneas 13-21) que pasa tres parámetros al constructor de la clase base `Transaccion`, mediante un inicializador de constructor. Este constructor también tiene los parámetros `tecladoATM` y `ranuraDepositoATM`, que asigna a las correspondientes variables de instancia de referencia (líneas 19-20).

```

1  // Deposito.cs
2  // Representa una transacción de depósito en el ATM.
3  public class Deposito : Transaccion
4  {
5      private decimal monto; // monto a depositar
6      private Teclado teclado; // referencia al Teclado
7      private RanuraDeposito ranuraDeposito; // referencia a la ranura de depósito
8
9      // constante que representa la opción de cancelar
10     private const int CANCELLO = 0;
11
12     // constructor de cinco parámetros que inicializa las variables de instancia de la
13     // clase
14     public Deposito( int numeroCuentaUsuario, Pantalla pantallaATM,
15                     BaseDatosBanco baseDatosBancoATM, Teclado tecladoATM,
16                     RanuraDeposito ranuraDepositoATM )
17     {
18         : base( numeroCuentaUsuario, pantallaATM, baseDatosBancoATM )
19         teclado = tecladoATM;
20         ranuraDeposito = ranuraDepositoATM;
21     }
22 }

```

Figura J.11 | La clase `Deposito` representa a una transacción de depósito con el ATM. (Parte 1 de 2).

```

18      // inicializa las referencias al teclado y a la ranura de depósito
19      teclado = tecladoATM;
20      ranuraDeposito = ranuraDepositoATM;
21  } // fin del constructor de cinco parámetros
22
23  // realiza una transacción; redefine el método abstracto de Transaccion
24  public override void Ejecutar()
25  {
26      monto = PedirMontoADepositar(); // obtiene el monto a depositar del usuario
27
28      // comprueba si el usuario introdujo un monto a depositar, o si canceló
29      if ( monto != CANCELO )
30      {
31          // solicita un sobre de depósito que contenga el monto especificado
32          PantallaUsuario.MostrarMensaje(
33              "\nIntroduzca un sobre de depósito que contenga " );
34          PantallaUsuario.MostrarMontoEnDolares( monto );
35          PantallaUsuario.MostrarLineaMensaje( " en la ranura para depósitos." );
36
37          // obtiene el sobre de depósito
38          bool sobreRecibido = ranuraDeposito.SeRecibioSobreDeposito();
39
40          // comprueba si se recibió el sobre de depósito
41          if ( sobreRecibido )
42          {
43              PantallaUsuario.MostrarLineaMensaje(
44                  "\nSe recibió su sobre.\n" +
45                  "El dinero que acaba de depositar no estará disponible " +
46                  "sino hasta que \nverifiquemos el monto del efectivo " +
47                  "dentro del sobre, y que se haya verificado cualquier cheque incluido." );
48
49              // abona a la cuenta para reflejar el depósito
50              BaseDatos.Abonar( NumeroCuenta, monto );
51          } // fin de if interno
52          else
53              PantallaUsuario.MostrarLineaMensaje(
54                  "\nNo insertó un sobre, por lo que el ATM ha " +
55                  "cancelado su transacción." );
56      } // fin de if externo
57      else
58          PantallaUsuario.MostrarLineaMensaje( "\nCancelando la transacción..." );
59  } // fin del método Ejecutar
60
61  // pide al usuario que introduzca un monto de depósito para abonarlo a la cuenta
62  private decimal PedirMontoADepositar()
63  {
64      // muestra el indicador y recibe la entrada
65      PantallaUsuario.MostrarMensaje(
66          "\nIntroduzca un monto a depositar en CENTAVOS (o 0 para cancelar): " );
67      int entrada = teclado.ObtenerEntrada();
68
69      // comprueba si el usuario canceló o introdujo un un monto válido
70      if ( entrada == CANCELO )
71          return CANCELO;
72      else
73          return entrada / 100.00M;
74  } // fin del método PedirMontoADepositar
75  } // fin de la clase Deposito

```

Figura J.11 | La clase Deposito representa a una transacción de depósito con el ATM. (Parte 2 de 2).

Redefinición del método abstract Ejecutar

El método Ejecutar (líneas 24-59) redefine el método abstract Ejecutar en la clase base Transaccion, con una implementación concreta que realiza los pasos requeridos en una transacción de depósito. La línea 26 pide al usuario que introduzca un monto a depositar, invocando al método utilitario private PedirMontoADepositar (declarado en las líneas 62-74), y establece el atributo monto con el valor devuelto. El método PedirMontoADepositar pide al usuario que introduzca un monto de depósito como un número entero de centavos (ya que el teclado del ATM no contiene un punto decimal; esto es consistente con muchos ATMs reales) y devuelve el valor decimal que representa el monto en dólares a depositar.

Obtener el monto a depositar con el método utilitario private PedirMontoADepositar

Las líneas 65-66 en el método PedirMontoADepositar muestran un mensaje pidiendo al usuario que introduzca un monto de depósito como un número de centavos, o “0” para cancelar la transacción. La línea 67 recibe la entrada del usuario desde el teclado. La instrucción if en las líneas 70-73 determina si el usuario introdujo un monto a depositar, o si eligió cancelar. Si el usuario elige cancelar, la línea 71 devuelve la constante CANCELLO. En caso contrario, la línea 73 devuelve el monto a depositar después de convertir el número int de centavos a un monto en dólares y centavos, dividiéndolo entre la literal decimal 100.00M. Por ejemplo, si el usuario introduce 125 como el número de centavos, la línea 73 devuelve 125 dividido entre 100.00M, o 1.25; 125 centavos son \$1.25.

La instrucción if en las líneas 29-58 del método Ejecutar determina si el usuario eligió cancelar la transacción, en vez de introducir un monto de depósito. Si el usuario cancela, la línea 58 muestra un mensaje apropiado y el método regresa. Si el usuario introduce un monto a depositar, las líneas 32-35 instruyen al usuario para que inserte un sobre de depósito con el monto correcto. Recuerde que el método MostrarMontoEnDolares de Pantalla imprime un valor decimal con formato de dólares (incluyendo el signo).

La línea 38 establece una variable local bool con el valor devuelto por el método SeRecibioSobreDeposito de ranuraDeposito, indicando si se recibió un sobre de depósito. Recuerde que codificamos el método SeRecibioSobreDeposito (líneas 7-10 de la figura J.5) para que siempre devuelva true, ya que estamos simulando la funcionalidad de la ranura de depósito y suponemos que el usuario siempre inserta un sobre a tiempo (es decir, dentro del límite de tiempo de dos minutos). No obstante, codificamos el método Ejecutar de la clase Deposito para que evalúe la posibilidad de que el usuario no inserte un sobre; la buena ingeniería de software demanda que los programas tomen en cuenta todos los posibles valores de retorno. Por ende, la clase Deposito está preparada para futuras versiones de SeRecibioSobreDeposito que pudieran devolver false. Las líneas 43-50 se ejecutan si la ranura de depósito recibe un sobre. Las líneas 43-47 muestran un mensaje apropiado al usuario. La línea 50 acredita el monto del depósito a la cuenta del usuario en la base de datos. Las líneas 53-55 se ejecutan si la ranura de depósito no recibe un sobre. En este caso, mostramos un mensaje indicando que el ATM ha cancelado la transacción. Después, el método regresa sin acreditar el monto a la cuenta del usuario.

J.13 La clase CasoEstudioATM

La clase CasoEstudioATM (figura J.12) simplemente nos permite empezar, o “encender” el ATM y evaluar la implementación de nuestro modelo del sistema ATM. El método Main de la clase CasoEstudioATM (líneas 6-10) simplemente crea una instancia de un nuevo objeto ATM llamado e1ATM (línea 8) e invoca a su método Ejecutar (línea 9) para iniciar el ATM.

```

1 // CasoEstudioATM.cs
2 // Aplicación para probar el caso de estudio del ATM.
3 public class CasoEstudioATM
4 {
5     // el método Main es el punto de entrada de la aplicación
6     public static void Main( string[] args )
7     {
8         ATM e1ATM = new ATM();
9         e1ATM.Ejecutar();
10    } // fin del método Main
11 } // fin de la clase CasoEstudioATM

```

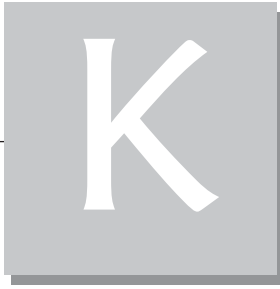
Figura J.12 | La clase CasoEstudioATM inicia el ATM.

J.14 Conclusión

Ahora que hemos presentado la implementación completa del ATM, podemos ver que surgieron muchas cuestiones durante la implementación para las que no proporcionamos diagramas de UML detallados. Esto no es poco común en una experiencia de diseño e implementación orientados a objetos. Por ejemplo, muchos atributos listados en los diagramas de clases se implementaron como propiedades de C#, para que los clientes de las clases pudieran obtener el acceso controlado a las variables de instancia `private` subyacentes. No creamos todas estas propiedades durante nuestro proceso de diseño, ya que no había nada en el documento de requerimientos o en nuestro proceso de diseño que indicara que, en un momento dado, habría que acceder a ciertos atributos fuera de sus clases.

También nos encontramos con varios problemas al simular el hardware. Un ATM real es un dispositivo de hardware que no posee un teclado de computadora completo. Un problema con el uso de un teclado de computadora para simular el teclado numérico es que el usuario puede introducir caracteres que no sean dígitos. No invertimos mucho tiempo lidiando con esas cuestiones, ya que este problema no es posible en un ATM real, que sólo tiene un teclado numérico. No obstante, es bueno tener que pensar acerca de cuestiones como ésta. Es muy típico que los diseños de software para sistemas completos impliquen la simulación de dispositivos de hardware como el dispensador de efectivo y el teclado. A pesar del hecho de que algunos aspectos de nuestro ATM pueden parecer inventados, en los sistemas reales el diseño y la implementación del hardware ocurre con frecuencia en paralelo con el diseño y la implementación del software. En tales casos, el software no puede implementarse en su forma final, ya que el hardware todavía no está listo. Por lo tanto, los desarrolladores de software deben simular el hardware, como hicimos en este caso de estudio.

¡Felicidades por haber completado todo el caso de estudio de ingeniería de software acerca del ATM! Esperamos que esta experiencia le haya parecido valiosa y que haya reforzado muchos de los conceptos que aprendió en los capítulos del 1 al 11. Apreciamos sinceramente que envíe sus comentarios, críticas y sugerencias. Puede contactarnos en deitel@deitel.com. Le responderemos lo más pronto posible.



UML 2: tipos de diagramas adicionales

K.1 Introducción

Si leyó las secciones opcionales del Caso de estudio de ingeniería de software en los capítulos del 3 al 9 y 11, ahora debe tener una idea de los tipos de diagramas de UML que utilizamos para modelar nuestro sistema ATM. El caso de estudio está pensado para usarse en cursos de primer o segundo semestre, por lo que limitamos nuestra discusión a un subconjunto conciso del UML. UML 2 cuenta con un total de 13 tipos de diagramas. El final de la sección 3.10 resume los seis tipos de diagramas que utilizamos en el caso de estudio. Este apéndice lista y define brevemente los siete tipos de diagramas restantes.

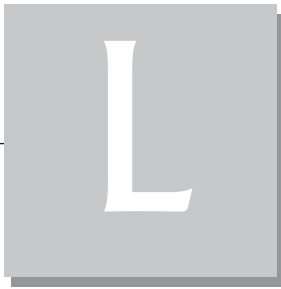
K.2 Tipos de diagramas adicionales

A continuación se describen los siete tipos de diagramas que optamos por no utilizar en nuestro Caso de estudio de ingeniería de software.

- *Diagramas de objetos*: modelan una “instantánea” del sistema; para ello modelan los objetos de un sistema y sus relaciones en un punto específico en el tiempo. Cada objeto representa una instancia de una clase de un diagrama de clases, y pueden crearse varios objetos a partir de una clase. Para nuestro sistema ATM, un diagrama de objetos podría mostrar varios objetos Cuenta distintos, uno al lado del otro, ilustrando que todos forman parte de la base de datos de cuentas del banco.
- *Diagramas de componentes*: modelan los *artefactos* y *componentes*: recursos (incluyen archivos de código fuente) que conforman el sistema.
- *Diagramas de despliegue*: modelan los requerimientos del sistema en tiempo de ejecución (como la computadora o computadoras en donde residirá el sistema), los requerimientos de memoria, o los demás dispositivos que requiera el sistema durante la ejecución.
- *Diagramas de paquetes*: modelan la estructura jerárquica de los *paquetes* (que son grupos de clases) en el sistema en tiempo de compilación, así como las relaciones que existen entre los paquetes.
- *Diagramas de estructuras compuestas*: modelan la estructura interna de un objeto complejo en tiempo de ejecución. Estos diagramas son nuevos en UML 2, y permiten a los diseñadores de sistemas descomponer en forma jerárquica un objeto complejo en partes más pequeñas. Los diagramas de estructuras compuestas están más allá del alcance de nuestro caso de estudio. Son más apropiados para aplicaciones industriales más grandes, las cuales exhiben agrupamientos complejos de objetos en tiempo de ejecución.

- *Diagramas de vista de interacción*: estos diagramas son nuevos en UML 2 y proporcionan un resumen del flujo de control en el sistema, mediante una combinación de elementos de varios tipos de diagramas de comportamiento (por ejemplo, diagramas de actividad, diagramas de secuencia).
- *Diagramas de sincronización*: también nuevos en UML 2, modelan las restricciones de sincronización impuestas en los cambios de etapas y las interacciones entre los objetos en un sistema.

Para aprender más acerca de estos diagramas y de los temas avanzados de UML, visite el sitio Web www.uml.org los recursos Web listados al final de las secciones 1.9 y 3.10.



Los tipos simples

Tipo	Tamaño en bits	Rango de valores	Estándar
bool	8	true o false	
byte	8	0 a 255, inclusive	
sbyte	8	−128 a 127, inclusive	
char	16	'\u0000' a '\uFFFF' (0 a 65535), inclusive	Unicode
short	16	−32768 a 32767, inclusive	
ushort	16	0 a 65535, inclusive	
int	32	−2,147,483,648 a 2,147,483,647, inclusive	
uint	32	0 a 4,294,967,295, inclusive	
float	32	<i>Rango negativo aproximado:</i> −3.4028234663852886E+38 a −1.40129846432481707E-45 <i>Rango aproximado positivo:</i> 1.40129846432481707E-45 a 3.4028234663852886E+38 <i>Otros valores soportados:</i> cero positivo y negativo infinito positivo y negativo no es un número (NaN)	IEEE 754 IEC 60559
long	64	−9,223,372,036,854,775,808 a 9,223,372,036,854,775,807, inclusive	
ulong	64	0 a 18,446,744,073,709,551,615, inclusive	

Figura L.1 | Los tipos simples. (Parte I de 2).

Tipo	Tamaño en bits	Rango de valores	Estándar
double	64	<i>Rango negativo aproximado:</i> $-1.7976931348623157E+308$ a $-4.94065645841246544E-324$ <i>Rango aproximado positivo:</i> $4.94065645841246544E-324$ a $1.7976931348623157E+308$ <i>Otros valores soportados:</i> cero positivo y negativo infinito positivo y negativo no es un número (NaN)	IEEE 754 IEC 60559
decimal	128	<i>Rango negativo:</i> $-79,228,162,514,264,337,593,543,950,335$ $(-7.9E+28)$ a $-1.0E-28$ <i>Rango positivo:</i> $1.0E-28$ a $79,228,162,514,264,337,593,543,950,335$ $(7.9E+28)$	

Figura L.1 | Los tipos simples. (Parte 2 de 2).

Información adicional sobre los tipos simples

- Este apéndice se basa en información de las secciones 4.1.4-4.2.8 de la versión de la *Especificación del lenguaje C#* de Microsoft, y en las secciones 11.1.4-11.1.8 de la ECMA-334 (la versión de ECMA de la *Especificación del lenguaje C#*). Estos documentos están disponibles en los siguientes sitios Web:

msdn.microsoft.com/vcsharp/programming/language
www.ecma-international.org/publications/standards/Ecma-334.htm

- Los valores de tipo float tienen siete dígitos de precisión.
- Los valores de tipo double tienen de 15 a 16 dígitos de precisión.
- Los valores de tipo decimal se representan como valores enteros que se escalan mediante una potencia de 10. Los valores entre -1.0 y 1.0 se representan exactamente hasta 28 dígitos.
- Para más información acerca de IEEE 754, visite el sitio Web grouper.ieee.org/groups/754/. Para más información acerca de Unicode, vea el apéndice E, Unicode®.

Índice

Símbolos

! (negación lógica), 166
!=, 69
#FIXED, 689
#IMPLIED, 689
#REQUIRED, 689
%>, 777
--%>, 777
%>, 832
&& (AND condicional), 164
* (Todas las columnas), 853
{, 51
{0:D2}, 223
|| (OR condicional), 164
+ y el - unarios, 130
<, 69
<%-->, 777
<%\$, 832
<%@, 777
<=, 69
<Ctrl> F5, 57
<Ctrl> z, 158
==, 69
>, 69
>=, 69

A

a prueba de subprocesos, 539
AbortRetryIgnore, 434
AboveNormal, 513
abstracción de datos, 291
acceso interno, 295
Aceptar, 53
acoplamiento, 422
activación, 259
actividad, 80
actor, 79
ADO.NET, 726
Adobe® Photoshop™ Elements
(www.adobe.com), 43
adquirir el bloqueo, 517
agregación, 111
Agregar
 elemento existente..., 792
 nuevo elemento, 87, 812
 página de contenido, 845
 referencia Web..., 870
alcance, 148
 de una declaración, 196
alias, 852
análisis y diseño orientados a
 objetos (A/DOO), 12
analizador
 de XML (o procesador de
 XML), 678
 DOM, 705

analizadores
 de validación, 678
 no validadores, 678
analizar un mensaje SOAP, 871
anclaje, 422
anchura de campo, 151
AND lógico booleano (&), 165
ángulo
 del arco, 597
 inicial, 597
anidamiento de instrucciones
 de control, 118
apellidoPaterno, 349
apilamiento de estructuras
 de control, 118
Aplicación
 de consola, 53
 de chat cliente/servidor,
 922
 para Windows, 34
aplicaciones
 de consola, 48
 de misión crítica, 234
 de n niveles, 775
 multinivel, 775
Application.Exit, 471
ApplicationException, 394
apuntador de posición
 de archivo, 658
árbol, 479, 984
 de origen, 697
 de resultados, 697
 del Modelo de objetos de
 documento (DOM), 705
árboles binarios, 984
archivo
 ASMX, 866
 de código subyacente
 (code-behind), 772
 de código, 87
 DISCO, 866
 .dll, 293
 secuencial, 636
Archivo, menú
 Cerrar solución, 34
 Guardar todo, 43, 56
 Nuevo proyecto, 34, 53
 Nuevo sitio Web..., 781
archivos
 ASPX, 772
 Properties, 28
 References, 28
 Form1.cs, 28
 Program.cs, 28
área de texto, 1103
args, 253
argumento, 52
 de tipo, 1004

EventArgs, 447
KeyPressEventArgs, 446
argumentos
 de línea de comandos, 179, 253
 de tipo explícito, 1005
arnés de prueba, 242
ArrangeIcons, 497
Array, ArrayList,
 Stack, Hashtable,
 SortedDictionary
 genérica y LinkedList
 genérica, 1019
arreglo
 ArrayList, 349
 de m por n, 242
arreglos, 216
 bidimensionales, 242
 dentados, 242
 multidimensionales, 242
 rectangulares, 242
ASC, 734
asignación dinámica de
 memoria, 966
asistente
 para configurar origen
 de datos, 825
 para la configuración
 de consultas de
 TableAdapter, 751, 902
 para la configuración de
 orígenes de datos, 740,
 854
 para la configuración de
 TableAdapter, 852, 853
asociación, 109
asociatividad, 68
ASP.NET, 5
asterisco (*), 67, 222, 689
Asterisk, 434
ATM, 109
atributo
 action, 781, 1103
 align, 1098
 alt, 1088
 AutoEventWireup, 777
 base, 696
 cols, 1116
 data-type, 704
 EnableViewState, 777
 href, 1084
 Language, 777
 match, 700
 maxOccurs, 695
 method, 1101
 method del elemento form,
 781
 minOccurs, 695
 opcional maxLength, 1103

order, 704
reservado xmlns, 685
rows, 1117
runat, 777
select, 700
selected, 1110
[Serializable], 663, 952
size, 1103
span, 1098
src, 1087
summary, 1096
type "hidden", 1103
usemap del elemento img, 1115
valign, 1100
value, 1103
version, 679, 699
WebMethod, 866
WebService, 876
WebServiceBinding, 876
atributos, 85, 1080
 rowspan y colspan, 1100
 shape, 1113
 WebService y
 WebServiceBinding, 878
autenticación de formularios, 834
autenticado, 838
autodocumente, 65
AVI, video, 619
ayuda
 dinámica, 31
 características
 Cómo, 31
 Buscar, 31
 Índice, 31
 Contenido, 31
 sensible al contexto, 32
Ayuda, menú
 Ayuda dinámica, 31
 Índice, 420
Ayudante de excepciones, 392

B

bandeja de componentes, 440, 745
barra
 de estado, 43
 de navegación, 843
 de título, 36, 1080
 diagonal inversa (\), 61
 indicadora de margen, 1054
barras separadoras, 455
base
 de datos, 636, 726
 relacional, 727
 del diseño, 1070
base(), 320
BaseDatosBanco, 109
baseline, 1101

bases de datos, 1078
relacionales, 726

BelowNormal, 513

biblioteca de clases, 3, 307
del .NET Framework (FCL),
49, 176

biblioteca de vínculos dinámicos,
293

bit, 634

BitArray, 1019

Bitmap, 605

Bloc de notas, 1078

bloque, 121

catch, 392

finally, 392

try, 392

bloquea, 920

booleana, 119

botón, 39

Agregar a los colores
personalizados, 591

Anidar archivos relacionados,
782

Configuración, 879

de opción, 1106

Personalizada, 789

Plantilla, 844

Ejecutar consulta, 755

Generador de consultas...,
754

Importar, 41

Propiedades de celda..., 789

botones

de estado, 430

de opción, 426

Propiedades, 783

Actualizar, 783

bottom, 1100

Brush, 582

buffer, 636

BufferedStream, 636

bugs, 1052

búsqueda en el DNS, 773

bytes, 634

C

cadena, 52

de caracteres, 52

de conexión, 742

de formato, 62

cálculo de referencias
(marshaling), 951

por referencia, 952

por valor, 952

campo, 634

Dirección URL, 881

campos, 11, 92

de una clase, 179

canal

HTTP, 951

TCP, 951

canales, 951

cantidad, 365

Capacity, 558

carácter

de barra diagonal (/), 1088

de escape, 61

de nueva línea, 61

de palabra, 571

Bo1d, 591

Italic, 591

Regular, 591

Strikeout, 591

Underline, 591

caracteres, 634

de espacio en blanco, 50

especiales, 547

Cascade, 497

casilla **Resultados**, 853

casillas de verificación, 426

celdas de datos, 1098

Cerrar sesión, 851

**CGI (Interfaz de compuerta
común)**, 1101

ciclo

de vida del software, 78

infinito, 122

ciclos, 118

círculo relleno, 117, 1054

rodeado por un círculo vacío,
117

círculos, 1093

clase, 11, 85

abstracta **Stream**, 636

Application, 457

autorreferenciada, 965

base, 306

directa, 306

indirecta, 306

Transaccion, 376

BinaryFormatter, 663

BindingSource, 748

BuscadorMaximo, 180

ButtonBase, 426

Console, 49, 52

Control, 420, 778

derivada, 306

Directory, 484

DirectoryInfo, 489

Exception, 394

Factura, 364

File, 634

FileInfo, 489

Font, 582

FontFamily, 582

GC (en el espacio de nombres
System), 283

genérica

LinkedList, 1036

LinkedListNode, 1036

SortedDictionary, 1034

Graphics, 582

GraphicsPath, 606

Hashtable, 1030

HttpSessionState, 816

IPAddress, 920

MarshalByRefObject, 952

MatchCollection, 571

object, 306

Page, 778

Process, 468

proxy, 871

PruebaBuscadorMaximo, 180

Random, 187

Regex, 570

remota, 952

SaveFileDialog, 648

Service, 878

Socket, 919

SolidBrush, 446, 588

SqlConnection, 739

StringBuilder, 558

System.Data.DataSet, 739

System.Object, 298

TcpClient, 921

TcpListener, 920

Timer, 414

TreeNode, 480, 708

Type, 361

UserControl, 503

ValueType, 565, 965

XmlReader, 706

XPathNavigator, 711

clases

abstractas, 347

bases abstractas, 347

concretas, 347

de caracteres, 571

de colección, 1018

de excepciones definidas por
el usuario, 406

genéricas, 1000

clases definidas por el usuario,
50

cláusula

catch general, 392

ORDER BY, 733

where, 732, 1006

clave

compuesta, 727

de registro, 636

primaria, 727

Clear, 1024

CML (Lenguaje de marcado
químico), 677

CodeFile, 777

codificaciones, 1070

código

de carácter, 547

de procesamiento de errores,
387

de tecla, 447

fuerza abierto, 3

códigos hash, 338

coincidencias de patrones, 733

cola, 291

de prioridades multinivel,
513

colaboración, 254

colección, 1018

ArrayList, 1024

Cookies, 814

Nodes, 480

SelectedItem, 470

colisión de nombres, 293

ColorDialog, 589

columna

autoincremental, 728

de identidad, 728

Filtro, 853

Reglas de acceso, 839

Usuarios, 839

columnas, 242

comando

Continuar, 1055

Insertar tabla, menú Diseño,
789

Paso a paso para salir, 1061

Paso a paso por
instrucciones, 1059

comas (,), 61

comentarios, 48

de una sola línea, 48

de ASP.NET, 777

de XHTML, 1079

delimitados, 48

Common Language Runtime, 5
comparación léxico-gráfica, 551
compilador justo a tiempo
(JIT), 6

CompleteWizardStep, 847

componente, 413

ToolTip, 440

componentes, 9, 11, 412

reutilizables estándar, 307

comportamiento del sistema,
79

composición, 279

comprobación

de límites, 226

de validez, 103

comunicaciones, 918

concesión entre espacio/tiempo,
1031

condición, 69

de continuación de ciclo, 118,
144

condiciones

de guardia, 118

simples, 164

Conexión

activa, 739

de área local, 879

Configuración, 789

de ASP.NET, 783

de conexión de red, 879

regional y de idioma, 1074

conflictos, 1030

de nombres, 293, 684

conjunto

de caracteres, 634

de doble byte (DBCS),
1072

multibyte (MBCS), 1072
Unicode®, 547, 634

de nodos, 700

Console.Out, **Console.In** y
Console.Error, 636

Consortio World Wide Web
(W3C), 4, 676

constante(s), 162

con nombre, 220

de cadena, 547

de enumeración, 195

FileAccess.Write, 653

FileMode.OpenOrCreate, 653

tipo carácter, 162, 547

constructor, 99

predeterminado, 99

sin parámetros, 276

constructores sobrecargados, 274

- Consulta
 - de prueba, 826
 - de SQL, 731
 - SELECT**, 731
- consultas, 726
- consumidor, 519
- consumir, 866
- contador, 122
- contenedor, 414, 1040
- contenido
 - complejo, 695
 - simple, 695
- content, 1115
- contexto de gráficos, 584
- control
 - activo, 422
 - BindingNavigator**, 745
 - ComboBox**, 474
 - Content**, 846
 - ContentPlaceHolder**, 844
 - CreateUserWizard, 834, 846
 - CheckedListBox**, 469
 - DataGridView**, 745
 - de validación, 798
 - DropDownList**, 793
 - HyperLink**, 793
 - Image**, 792
 - Label**, 38, 39
 - LinkLabel**, 465
 - ListBox**, 469
 - Login**, 849
 - LoginName**, 851
 - LoginStatus**, 837, 851
 - MenuStrip**, 453
 - MonthCalendar**, 460
 - NumericUpDown**, 442
 - ObjectDataSource**, 852
 - PictureBox**, 41
 - RadioButtonList**, 793
 - SqlDataSource**, 825
 - TabControl**, 489
 - TextBox**, 793
 - TreeView**, 479, 708
 - Web **AdRotator**, 793
 - Web **Label**, 781
 - Windows Media Player**, 619
 - XmlDataSource**, 795
- controles, 7
 - ASP.NET de inicio de sesión**, 834
 - constituyentes, 503
 - de servidor **ASP.NET**, 772
 - RegularExpressionValidator**, 802
 - RequiredFieldValidator**, 801
 - Timer**, 504
 - Web, 772
- Controles comunes (grupo), 31, 38
- conversión
 - boxing, 965
 - descendente, 345
 - explícita, 130
 - implícita, 130
 - unboxing, 965
- Convert.ToInt32 (Método) (System), 395
- cookie *cifrada*, 850
- cookies, 809
- coordenada
 - horizontal, 583
 - vertical, 583
- x, 583
- y, 583
- coords, 1113
- copia superficial, 338
- Copiar sitio Web, 783
- corchetes ([]), 216
- correo electrónico, 4
- Crear directorio para la solución
 - (opción), 56, 57
 - casilla de verificación, 35
- crear un objeto, 85
- Create-Instance**, 1024
- CreateUserWizardStep, 847
- criterios de selección, 732
- cuadro cerrar, X, 8
- cuadro de diálogo
 - Elegir elementos del cuadro de herramientas, 619
 - Agregar conexión, 740
 - Agregar referencia..., 294
 - Agregar referencia Web, 881
 - Campos, 828
 - Configuración avanzada, 789
 - Fuente**, 39
 - Guardar proyecto**, 35
 - Insertar tabla, 789, 844
 - Nuevo proyecto, 34
 - Nuevo sitio Web, 781
 - Opciones, 32, 786
 - Propiedades de celda, 790
 - Seleccionar imagen, 793
 - Seleccionar recurso**, 41
 - Seleccionar una página principal**, 851
 - Selector de color, 790
 - Ubicación del proyecto**, 35
- cuadro de grupo Diseño, 789
- Cuadro de herramientas, 29, 414
- cuadro de información sobre herramientas (tool tip), 24, 55, 1055
- cuadro de lista desplegable de selección de componente, 31
- Objeto inicial, 179
- cuadro de texto, 1103
- de contraseña, 425
- cuadros combinados Filas y Columnas, 789
- cuanto, 513
- Cuenta, 109
- cuerpo
 - de la declaración del método, 52
 - de la tabla, 1098
- cursor de la pantalla, 52
- Custom**, 462
- CheckBox**, 430
- D**
- Dar formato al documento (menú Edición), 50
- DashCap.Round**, 606
- DashStyle.Dash**, 606
- datagramas, 919
- DataRow**, 739
- DataSet**
 - fuertemente tipificado, 748
 - sin tipo, 748
- DateChanged**, 460
- DateTimePicker**, 461
- datos
 - de tabla, 1098
 - persistentes, 634
 - tipo carácter analizados, 690
- decisión, 69
- declaración, 64
 - de la clase, 86
 - de la propiedad, 94
 - de lista de atributos **ATTLIST**, 689
 - de tipo de elemento **ELEMENT**, 689
 - de una interfaz, 363
 - namespace, 292
 - XML**, 679
- decremento, 144
- DefaultValue**, 856
- Definición de tipo de documento (**DTD**), 678
- Definir métodos de datos, 854
- delegado, 511
- delegados, 419
- Delegate**, 419
- DeleteCommand**, 832
- Deposito, 109
- Depuración Just-In-Time, 389
- depurador, 1052
- Depurar
 - Detener depuración, 788, 1059
 - Iniciar depuración, 788, 1055
 - Iniciar sin depurar, 57, 389, 788
 - Variables locales (menú Ventanas), 1057
- dequeue (retirar de la cola), 980
- desarrollo de aplicaciones Web, 772
- desbordamiento
 - aritmético, 291
 - de pila, 184
- DESC**, 734
- descenso recursivo, 704
- descripción del servicio, 867
- Descripción universal, descubrimiento e integración (**UDDI**), 883
- descripcionPieza, 365
- descriptores de acceso, 94
 - get**, 85
- Descubrimiento de servicios Web (**DISCO**), 866
- deseleccionado, 433
- deserializarse, 663, 952
- deshabilitar un punto de interrupción, 1055
- desplazar, 188
- destino de la PI, 699
- destructor, 282
- detalles de implementación de la clase, 268
- detección de colisiones, 609
- diagrama(s)
 - con elementos omitidos (elided diagram), 109
 - de actividad, 80, 117
 - de caso-uso, 79, 80
 - de clases, 80, 109
 - de colaboración, 257
 - de comunicación, 80, 257
 - de entidad relación (**ER**), 730
 - de interacción, 257
 - de máquina de estado, 80, 169
 - de secuencia, 80, 257
- diamantes sólidos, 110
- DictionaryEntry**, 1033
- dígito binario, 634
- dígitos decimales, 634
- Dirección
 - de correo electrónico de Internet, 802
 - de protocolo Internet (**Dirección IP**), 920
 - de red, 920
 - de retorno de bucle (loopback) **IP**, 926
 - IP**, 773
- direccionamiento
 - absoluto, 700
 - relativo, 700
- directiva, 777
 - Master**, 843
 - Page**, 777
 - using**, 49
- directorio
 - App_Code**, 877
 - debug de la aplicación, 388
 - virtual, 773
- disco, 1091
 - vsdisco, 866
 - discomap, 866
 - map, 866
- Diseñador de DataSet**, 751
- diseño, 12
 - orientado a objetos, 10
- DispensadorEfectivo, 109
- Display**, 591
- DisplayDelegate**, 926
- dispositivos de almacenamiento secundario, 634
- división
 - de enteros, 67
 - entera, 127
 - entre cero, 389
- DOCUMENT**, 591, 785
- documentación de .NET, 49
- documento(s)
 - de instancia **XML**, 695
 - de requerimientos, 74
 - de **XML** bien formado, 678
 - XHTML**, 1078
- DocumentTitle**, 951
- DocumentTitleChanged**, 951
- Double.NaN**, 389
- Double.NegativeInfinity**, 389
- Double.PositiveInfinity**, 389
- DrawArc**, **DrawPie** y **FillPie**, 598
- DropDownList**, 477
- DTD** externa, 683
- Dynamic**, 802

E

- Editar DataSet con el Diseñador, 751
- Editar y continuar*, característica, 1062
- Editor
 - de expresiones regulares, 802
 - de la colección Cadena, 470
 - de la colección Imágenes, 485
 - de la colección ListItem, 793
 - de parámetros y comandos, 828
 - de texto, 54
 - TreeNode, 480
 - WYSIWYG (*Lo que se ve es lo que se obtiene*), 786
- ejecución secuencial*, 116
- Elegir
 - origen de datos, 795, 854
 - un objeto comercial, 854
- elemento*
 - a (ancla), 1084
 - AlternateText, 798
 - all, 696
 - anidado, 1080
 - attribute, 696
 - authentication, 842
 - authorization, 842
 - body, 1081
 - br, 1088
 - caption, 1096
 - cero, 216
 - col, 1098
 - colgroup, 1098
 - Columns, 860
 - complexType, 695
 - contenedor, 680
 - ControlParameter, 860
 - checkbox, 1104
 - de entrada "*password*", 1103
 - de lista, 1091
 - de probabilidad, 187
 - de1, 1091
 - deny, 842
 - element, 695
 - extension, 696
 - form, 1101
 - frameset, 1116
 - head, 1080
 - html, 1080
 - ImageUrl, 796
 - img, 1087
 - Impressions, 798
 - input
 - "checkbox", 1104
 - "reset", 1103
 - "submit", 1103
 - "text", 1103
 - Label, 1103
 - li, 1091
 - ListItem, 793
 - map, 1113
 - minInclusive, 696
 - NavigateUrl, 796
 - noframes, 1119
 - ol, 1092
 - option, 1110
 - Puntero, 414
 - raíz, 677
 - Advertisements, 796
 - schema, 693
 - script, 843
 - select, 1110
 - sequence, 695
 - simpleType, 696
 - span de XHTML, 778
 - strong, 1084
 - sup o sub, 1091
 - table, 1096
 - tbody, 1098
 - textarea, 1103
 - tfoot, 1098
 - thead, 1098
 - tr, 1098
 - ul, 1091
 - vacío bandera, 684
 - value-of, 700
- elementos, 216, 676, 1080
 - Ad, 796
 - area, 1113
 - BoundField, 860
 - de encabezado, 1083
 - de formato, 62
 - de menú, 452
 - de sesión, 819
 - frame, 1117
 - hijos, 680
 - meta, 1115
 - padres, 680
 - td, 1098
 - th, 1098
 - vacíos, 1088
 - xsl:for-each, 700
- eliminación de duplicados*, 991
- emacs, 1078
- emisor del evento, 419
- Empleado, 343
- EmpleadoAsalariado, 343
- EmpleadoBaseMasComision, 343
- EmpleadoBaseMasComision4, 343
- EmpleadoPorComision, 343
- EmpleadoPorComision2
 - ToString(), 347
- EmpleadoPorComision3, 345
- EmpleadoPorHoras, 343
- encabezado,
 - cuerpo y pie, 1098
 - del método, 87
- encabezados, 1083
- HTTP, 774
- encapsular, 11
- enlace, 965
- enlaza, 920
- enqueue (agregar a la cola), 980
- enteros, 63
- Entorno, 54
- entorno de desarrollo integrado, 3
- entrada/salida estándar, 52
- entradas ocultas, 780, 1101
- enumeración, 195
 - ComboBoxStyle, 476
 - DashCap, 606
 - DashStyle, 606
 - DateTimePickerFormat, 462
 - DayOfWeek, 463
- DialogResult, 438
- FileAccess, 653
- FontStyle, 431, 591
- HatchStyle, 586
- Keys, 447
- LinearGradientMode, 603
- MdiLayout, 497
- SelectionMode, 469
- ThreadPriority, 513
- XmlNodeType, 708
- enumeradores, 1018
- envoltura SOAP, 871
- equipo remoto, 865
- error, 434
 - de sintaxis, 58
- escalar, 188
- escritura, 1074
- espacios de nombres, 49
 - de .NET Remoting System
 - Runtime.Remoting,
 - System.Runtime.
 - Remoting.Channels
 - y System.Runtime.
 - Remoting.Channels.Http,
 - 957
 - global, 92
 - predeterminado, 687
 - System, 49
 - System.Collections, 814
 - System.Collections
 - Generic, 1018
 - System.Collections
 - Specialized, 1018
 - System.Data, 739
 - System.Diagnostics, 468
 - System.IO, 484
 - System.Net, 920
 - System.Net.Sockets, 920
 - System.Runtime
 - Serialization.
 - Formatters.Binary, 663
 - System.Text
 - RegularExpressions, 570
 - System.Web.UI, 778
 - System.Web
 - UI.WebControls, 779
 - System.Windows.Forms, 414
 - System.Xml.XPath, 711
 - xsl, 699
- espacio en blanco, 50
- especializaciones, 377
- especificación de diseño, 79
- especificador de, 104
 - formato D, 223
 - formato D2, 266
- esquema, 678
 - de color Profesional, 847
- estado, 80
 - AbortRequested, 511, 512
 - Background, 513
 - final, 117
 - inicial, 117, 169
 - Suspended, 513
 - SuspendRequested, 513
 - Unstarted, 511
 - WaitSleepJoin, 512
- Estándar Unicode, 1070
- estereotipos, 97
- estilo
 - de la fuente, 591
 - de mayúsculas/minúsculas
 - Pascal, 50
 - DropDown, 476
- estrechamente empaquetados, 991
- estructura, 1078
 - Color, 582
 - DateTime, 463, 504
 - de datos, 184, 216
 - lineales, 984
 - "último en entrar, primero en salir" (LIFO), 184
 - de repetición, 116
 - de secuencia, 116
 - de selección, 116
 - del sistema, 79
 - Point, 584
 - Rectangle, 584
 - Size, 584
- etiqueta
 - default, 159
 - final, 1080
 - final </title>, 773
 - <frameset>, 1116
 - <hr />, 1091
 - inicial, 1080
 - stylesheet, 699
 - <title>, 773
- Label, 778
- etiquetas, 773
 - case, 159
 - de párrafo (<p> y </p>), 1081
 - iniciales y etiquetas finales, 676
 - <textarea> y </textarea>, 1103
- evaluación en corto circuito, 165
- EventHandler, 419
- evento
 - Click, 455
 - Click del control Button, 416
 - DocumentCompleted, 951
 - Init, 779
 - ItemCheck, 473
 - KeyPress, 446
 - LinkClicked, 465
 - Navigating, 951
 - Paint, 503, 584
 - predeterminado, 419
 - ProgressChanged, 951
 - SelectedIndexChanged, 469, 477
 - StatusTextChanged, 951
- eventos
 - de tecla, 446
 - del ratón, 444
- Examinador de objetos, 297
- excepción
 - ArgumentOutOfRangeException, 548
 - ArgumentException, 991, 1033
 - DivideByZeroException, 389
 - FormatException, 389

- InvalidCastException*, 361
 - InvalidOperationException**, 1023
 - IOException**, 653
 - KeyNotFoundException*, 1036
 - no atrapada*, 392
 - OutOfMemoryException*, 966
 - SerializationException**, 663
 - Exclamation, 434
 - exclusión mutua*, 517
 - exit, 58
 - Expat XML Parser*, 678
 - expat.sourceforge.net, 678
 - explorador basado en texto*, 1088
 - Explorador de soluciones, 19
 - Explorar con..., 788
 - exponer, 866
 - expresión*
 - ASP.NET*, 829
 - expresión condicional*
 - constante de cadena*, 155
 - constante integral*, 155
 - de acceso a un arreglo*, 216
 - de acción*, 117
 - de creación de arreglos*, 217
 - de creación de objeto*, 88
 - del switch*, 159
 - expresiones*, 66
 - extensión*
 - .dll* (biblioteca de vínculos dinámicos), 57
 - .exe* (ejecutable), 57
 - de archivo
 - .asmx*, 867
 - .aspx*, 772
 - .cs*, 364
 - .html* o *htm*, 1078
 - .master*, 843
 - .xml*, 677
 - .xsl*, 699
 - de nombre de archivo .cs*, 28
 - Extensiones multipropósito de correo Internet (MIME)*, 774
 - extiende*, 306
- F**
- factor*
 - de carga*, 1030
 - de escala*, 188
 - falsa*, 69
 - FCL*, 176
 - FCL de .NET*, 29
 - fecha de expiración*, 809
 - ficha
 - Colores con nombre, 790
 - Componentes COM, 619, 621
 - Examinar, 294
 - Opciones avanzadas, 789, 879
 - Personalizado**, 37
 - Personalizado, Web y Sistema, 37
 - Seguridad, 839
 - Servicios, 789
 - fila de la tabla*, 1098
 - filas*, 242
 - y *columns*, 727
 - FileAccess.Read*, 653
 - FileAccess.ReadWrite*, 653
 - FileStream*, 636
 - Firewall de Windows, 789, 879
 - firewalls*, 871
 - firma*, 200
 - flecha amarilla*, 1055
 - flechas*
 - de navegabilidad*, 299
 - de transición*, 117
 - flujo*, 636
 - de trabajo*, 117
 - “Forms” del atributo *mode*, 842
 - Formato**
 - automático, 825
 - Centrar en el formulario, Horizontalmente, 39
 - de código fuente*, 1078
 - formatos multimedia*, 619
 - formulario*, 1096, 1101
 - formularios*
 - Web Forms*, 772
 - Windows Forms*, 413
 - Fuentes y colores, 54
 - fuga de memoria*, 283, 395
 - fuga de recursos*, 282, 395
 - función*, 11
 - de establecimiento de nodo name*, 703
 - de establecimiento de nodo text*, 704
- G**
- GDI+*, 582
 - “Generación satisfactoria”, 43
 - Generador de consultas**, 754, 853
 - generalización*, 377
 - Generar solución (menú Generar), 43, 57, 1055
 - Generar instrucciones
 - Insert, 826
 - Update, 826
 - Delete, 826
 - Generar página**, 788
 - Generar sitio Web**, 788
 - genéricos*, 1000
 - GET*, 774
 - GIF (Formato de intercambio de gráficos), 41
 - glifos*, 1072
 - gramática EBNF (Forma de Backus-Naur extendida), 688
 - GraphicsUnit*, 591
 - GridView*, 824
 - GroupBox*, 428
 - Guardar, 57
 - Guardar proyecto, 56
 - guión bajo* (`_`), 733
- H**
- Habilitar ordenación**, 856
 - hace referencia a un objeto*, 98
 - hacer clic*, 444
 - Hand, 434
 - handshaking*, 918
 - hashing*, 1030
 - Hashtable, 814
 - HatchBrush*, 603
 - height, 424, 1087
 - HelpLink*, 403
 - herencia*, 10
 - de implementación*, 349
 - de interfaz*, 349
 - simple*, 306
 - visual*, 500, 838
 - hermanos*, 984
 - Herramienta Administración de sitios Web**, 839
 - Herramientas > Elegir elementos del cuadro de herramientas..., 619
 - Herramientas > Opciones..., 53
 - Herramientas > Opciones de Internet..., 951
 - hexadecimales (hex)*, 1090
 - Highest**, 513
 - hijo*, 984
 - derecho*, 984
 - izquierdo*, 984
 - hipertexto*, 773
 - hipervínculo de imagen*, 1088
 - hipervínculos*, 773
 - hoja de estilo*, 680
 - en cascada (CSS)*, 1078
 - XSL*, 697
 - Horas, 353
 - host*, 773
 - http://localhost*, 782
 - http://www.w3.org/1999/XSL/Transform*, 699
 - http://www.w3.org/2001/XMLSchema*, 693
 - HttpCookie*, 814
 - HttpCookieCollection*, 815
- I**
- icono
 - Alfabético**, 30
 - Eventos, 420
 - Por categorías**, 30
 - Propiedades, 420
 - ID de sesión único*, 821
 - identificador*, 50
 - de recursos uniforme (URI)*, 685
 - get*, 95
 - set*, 95
 - IDictionary**, 1033
 - implementa*, 364
 - Increment**, 442
 - incremento*, 144
 - Inch*, 591
 - independencia*
 - de datos*, 4
 - de la plataforma*, 6
 - independiente del lenguaje*, 6
 - index.html*, 1084
 - indexador de string, 549
 - indexadores*, 271
 - IndexOf**, 1024
 - IndexOfAny*, 553
 - IndexOutOfRangeException**, 226
 - IndexOutOfRangeException**, 394
 - indicador*, 65
 - de fin de archivo*, 158
 - de ocurrencia de signo positivo (+)*, 689
 - índice, 395
 - cero*, 216
 - del elemento*, 216
 - inferencia de tipos*, 1004
 - información a nivel de clase*, 283
 - Información de parámetros*, 55
 - Information, 434
 - inicializador*
 - de arreglos*, 219
 - de constructor*, 276
 - inicializadores de arreglos anidados*, 242
 - Iniciar sesión, 851
 - iniciativa .NET*, 5
 - Inicio > Todos los programas > Accesorios > Símbolo del sistema, 57, 388
 - InnerException**, 402
 - inorden, preorden y postorden*, 984
 - Insertar Separator, 455
 - InsertCommand**, 832
 - instrucción*, 52
 - break*, 159, 162
 - continue*, 162
 - de asignación*, 65
 - de declaración de variable*, 64
 - de procesamiento (PI)*, 698
 - de repetición*, 144
 - do...while*, 153
 - for*, 146
 - while*, 122
 - de retorno*, 95
 - de selección múltiple*
 - switch*, 155
 - doble*, 118
 - múltiple*, 118
 - simple*, 118
 - DELETE**, 738
 - foreach*, 230
 - goto*, 116
 - if*, 117
 - INSERT**, 737
 - throw*, 400
 - try*, 393
 - UPDATE**, 737
 - using*, 401
 - vacía*, 73
- instrucciones*
 - de control de una entrada/una salida*, 118
 - de iteración*, 118
 - de repetición*, 118
 - de selección*, 117
 - if...else anidadas*, 120
- IntelliSense*, 55
- interbloqueo*, 518
- intercambio*, 918
- «interface», 365, 788
- interfaces*, 11
 - genéricas*, 1000

interfaz

- de base de datos*, 726
- de múltiples documentos (MDI)*, 494
- de un solo documento (SDI)*, 494
- IComparable**, 991
- IComparable<T>**, 1005
- IComponent**, 413
- IEnumerable**, 1023
- ISerializable**, 663, 952
- llamada **IPorPagar**, 364
- public, 265
- pública de la clase, 268
- Internet Explorer**, 1078
- Internet Information Services (IIS)**, 773
- interoperabilidad de lenguajes*, 6
- invalidación regional*, 609
- InvalidCastException**, 965
- inválido para el esquema*, 691
- invocar*, 99
- IPEndPoint**, 921
- Items**, 477
- iteración del ciclo*, 144

J

- Jasc® Paint Shop Pro™
(www.jasc.com), 41, 43
- jerarquía*
 - de clases*, 306
 - de datos*, 634
 - de herencia*, 307
- JPEG** (Grupo de expertos en fotografía unidos), 41
- justifica a la derecha*, 151
- justificar a la izquierda*, 151
- justo a tiempo (JIT)*, compilador, 6

K

- KeyDown**, 447
- KeyUp**, 447

L

- lanzar una excepción*, 387
- LastIndexOf**, 553, 1024
- LastIndexOfAny**, 553
- lenguaje*
 - compatible con .NET*, 6
 - de descripción de servicios Web (WS-DL)*, 867
 - de hojas de estilos extensible (XSL)*, 678
 - de marcado*, 1078
 - de Marcado de Hipertexto*, 4
 - de marcado extensible (XML)*, 4, 676
 - extensible*, 88
 - fuertemente tipificado*, 137
 - Intermedio Común**, 6
 - intermedio de Microsoft (MSIL), 6, 57
 - Unificado de Modelado*, 13

letras, 634

- libro de visitantes*, 824
- LibroCalificaciones**, 85, 86
- LibroCalificaciones.cs**, 86
- limpieza de la pila*, 403
- línea*
 - base*, 1101
 - de ajuste*, 424
 - de vida*, 258
 - punteada*, 117
- LinearGradientBrush**, 603
- LinearGradientMode**
- ForwardDiagonal**, 603, 605
- lista*
 - circular de enlace simple*, 976
 - circular doblemente enlazada*, 976
 - de enlace simple*, 976
 - de parámetros de tipo*, 1003
 - desordenada*, 1091
 - desplegable*, 474
 - Configuraciones de la solución, 58
 - Filtrado por, 420
 - Lenguaje, 782
 - Origen del parámetro, 856
 - doblemente enlazada*, 976
 - enlazada*, 966
 - inicializadora*, 219
 - separada por comas*, 61
- listas*
 - de argumentos de longitud variable*, 250
 - ordenadas*, 1092
 - separadas por comas*, 151

ListView, 484**ListViewItem**, 485**literal**

- de cadena*, 52
- decimal**, 102
- literales*
 - de cadena*, 547
 - de punto flotante*, 102

localhost 926

localización, 1070

lógica

- comercial*, 775
- de control*, 775
- de presentación*, 775

LoginName, 837

Long, 462**Lowest**, 513**Llamada**

- a procedimiento remoto (RPC)*, 866
- a un método*, 85
- asíncrona*, 257
- recursiva*, 201
- síncrona*, 257
- llave izquierda*, 51
- llaves* ({} y {}), 121

M**manejado**

- de eventos*, 415
- Page_Load**, 780
- Page_Unload**, 780

- de excepciones*, 386
- del formulario*, 1103
- de tamaño habilitados*, 36
- mapas de imágenes*, 1096, 1112
- marcado*, 676, 773
- marcador de fin de archivo*, 636
- marcadores de visibilidad*, 299
- “marcar” una ventana*, 25
- marco de pila*, 184
- marcos*, 1096, 1116
- Math.E**, 178
- Math.PI**, 178
- Math.Sqrt**, 178
- MathML**, 677
- MaxDate**, 464
- Maximum**, 442
- MaximumSize**, 424
- MemoryStream**, 636
- mensaje*, 52, 254
 - anidado*, 258
 - SOAP*, 871
- mensajes*, 85
- menú*

Ayuda, 31, 50*de etiquetas inteligentes*, 793

- Tareas de**
 - CreateUserWizard**, 847
 - Tareas de Login**, 849
 - Tareas de LoginStatus**, 851
- Formato**, 424
- General de Visual Web Developer**, 788
- Generar**, 879
- Message**, 402
- MessageBox**, 416
- MessageBoxButtons**, 434
- MessageBoxIcon**, 434
- método*, 84
 - AcceptSocket**, 920
 - Add**, 471, 821, 961
 - Add de ArrayList**, 603
 - AddDays**, 463
 - AddLast**, 1039
 - AddLine de GraphicsPath**, 606
 - AddYears**, 464
 - Bind de la clase Socket**, 920
 - CTear**, 471, 479
 - CTear de ArrayList**, 603
 - CTear de Graphics**, 609
 - Close**, 653
 - Close de BinaryReader**, BinaryWriter, NetworkStream y Socket, 920
 - Connect**, 921
 - Connect de RemotingServices**, 960
 - Console.WriteLine**, 52
 - Contains**, 1027
 - ContainsKey de Hashtable**, 1033
 - CopyTo**, 549
 - CreateGraphics**, 446
 - CreateGraphics de Form**, 609
 - CreateNavigator**, 716

Deserialize, 663

- DeterminarMaximo**, 180
- Dispose*, 418
- DrawEllipse**, 479
- DrawImage de Graphics**, 609
- DrawPie**, 479
- DrawRectangle**, 479
- DrawString**, 588, 591
- EndEdit**, 749, 750
- EndsWith**, 553
- Enter**, 512, 517
- Exists**, 489
 - de Directory*, 640
- Exit**, 518
- ExpandAll de TreeView**, 710
- Find**, 1039
- Focus**, 422
- FromImage**, 605
- “get”*, 1102
- GetDirectories**, 484, 489
- GetDirectories de Directory**, 640
- GetEnumerator**, 1023
- GetFiles**, 489, 644
- GetLength**, 245
- GetStream**, 921
- GetType**, 361
- GetXml de DataSet**, 768
- GoBack de WebBrowser**, 950
- GoForward**, 950
- GoHome**, 951
- IndexOf**, 1027
- InitializeComponent**, 418
- Interrupt**, 513
- InvalidDate**, 584
- Invoke**, 539, 926
- Join**, 513
- LayoutMdi**, 497
- Load del objeto**
 - XslCompiledTransform**, 722
- Match**, 571
- Matches**, 571
- Maximo**, 180
- MoveNext**, 717
- MoveTo**, 630
- Navigate de WebBrowser**, 951
- Next de la clase Random**, 188
- ObtenerMontoPago**, 364
- OnPaint**, 503
- OpenRead de WebClient**, 956
- Page_Init**, 779
- Peek**, 1028
- Play**, 631
- “post”*, 1102
- predicado*, 971
- Pulse o PulseAll**, 512
- Read de XmlReader**, 707
- ReadLine de Console**, 65
- ReadString de BinaryReader**, 920
- recursivo*, 200
- Refresh**, 951
- Refresh de TreeView**, 710
- Remove**, 1027, 1039
- RotateTransform**, 606
- Seek de FileStream**, 662

- Serialize**, 663
 - ShowDialog**, 649
 - Shutdown, 920
 - Sleep**, 512
 - sobrecargado*, 198
 - Sort** de la clase **ArrayList**, 957
 - Sqrt de la clase **Math**, 407
 - Start**, 468
 - Start de **TcpListener**, 920
 - StartsWith, 553
 - static
 - Concat, 556
 - Copy** de **Array**, 1023
 - Create** de **XmlReader**, 706
 - Exit**, 457
 - Format**, 266
 - FromFile** de **Image**, 609
 - Sort** de **Array**, 1023
 - Synchronized**, 1040
 - WaitForPendingFinalizers**, 287
 - Stop**, 951
 - string
 - ToLower**, 1033
 - ToUpper**, 1039
 - Suspend**, 513
 - ToArray**, 603
 - ToDecimal**, 106
 - ToInt32** de la clase **Convert**, 65
 - ToLongDateString**, 463
 - ToLongTimeString**, 504
 - Tostring**, 182
 - Transform** de la clase
 - XslCompiledTransform**, 722
 - Update**, 750
 - Wait**, 512
 - Write
 - de **BinaryWriter**, 920
 - de **Console**, 60
 - método abreviado de teclado, 454
 - métodos, 11
 - abstractos*, 347
 - ayudantes*, 161
 - de acceso abreviado*, 452
 - del servicio Web*, 866
 - FillRectangle** y **DrawRectangle**, 597
 - genéricos, 1000
 - IndexOf**, 553
 - MoveNext** y **Reset**, 1023
 - Replace**, 557
 - static**, 152, 176
 - Substring**, 555
 - ToString** e **Ingresos**, 346
 - utilitarios*, 161
 - Web**, 866
 - WriteXML**, **ReadXML** y **GetXml**, 765
 - Microsoft Agent**, 621
 - Microsoft Paint**, 43
 - Microsoft SansSerif**, 591
 - Microsoft SQL Server 2005 Express**, 726
 - Microsoft XML Core Services (MSXML)**, 678
 - middle, 1100
 - MiddleCenter**, 40
 - miembros*, 55
 - Millimeter**, 591
 - Min y Max, de **Math**, 198
 - MinDate**, 464
 - Minimum**, 442
 - MinimumSize**, 424
 - modelo*
 - de objetos ADO.NET*, 739
 - de reanudación del manejo de excepciones*, 393
 - de terminación del manejo de excepciones*, 393
 - modelos*
 - de cascada*, 78
 - iterativos*, 78
 - modificador*
 - de acceso*, 86
 - partial*, 418
 - modo*
 - de diseño*, 36, 418
 - de ejecución*, 43
 - de interrupción*, 1054
 - Ejecución, 788
 - Monitor**, 511
 - Mostrar la Ayuda, 33
 - Mostrar todas las configuraciones, 33, 54
 - MouseEventArgs**, 444
 - MouseEventHandler**, 444
 - mover*, 444
 - MoveToFirstChild**, 716
 - MoveToNext**, 716
 - MoveToParent**, 716
 - MoveToPrevious**, 716
 - MPEG**, 619
 - MIDI**, *audio*, 619
 - MulticastDelegate**, 419
 - multidifusión de eventos*, 419
 - MultiExtended**, 469
 - MultiSelect**, 484
 - MultiSimple**, 469
 - name**, 1103, 1115
 - N**
 - navegabilidad bidireccional*, 300
 - .NET Remoting**, 951
 - NetworkStream**, 921
 - nivel*
 - cliente*, 776
 - de datos*, 775
 - de información*, 775
 - inferior*, 775
 - intermedio*, 775
 - superior*, 776
 - niveles*, 775
 - nodo*
 - hoja*, 984
 - padre*, 479, 705
 - raíz*, 984
 - Tipos base*, 298
 - nodos*, 479, 680, 966
 - ancestros*, 705
 - descendientes*, 705
 - hermanos*, 479, 705
 - hijos*, 479, 705
 - Nombre**, 53, 66
 - calificado*, 736
 - de clase*
 - completamente calificado*, 92
 - completamente calificado System.Console*, 186
 - descalificado Console*, 186
 - de host*, 773
 - de la clase*, 50
 - de recurso uniforme (URN) y el Localizador de recursos uniforme (URL)*, 685
 - de rol*, 110
 - del arreglo*, 217
 - del atributo*, 139
 - simple*, 292
 - Nombre de usuario, 851
 - None**, 434, 469
 - Normal**, 513
 - normalizan* las bases de datos, 734
 - NOT**, 738
 - notación de complementos a dos*, 1050
 - notas*, 117
 - Nuevo proyecto, 19, 53
 - NullReferenceException**, 394
 - número de puerto*, 920
 - Número de teléfono en EE.UU., 802
 - NumeroComplejo**, 374
 - numeroPieza**, 365
 - números
 - aleatorios*, 187
 - de punto flotante de precisión doble*, 102
 - de punto flotante de precisión simple*, 102
 - seudoaleatorios*, 188
 - numeroSeguroSocial**, 349
 - O**
 - ObjectCollection**, 470
 - Objeto**, 854
 - ArrayList**, 600
 - BoundField**, 829
 - comercial*, 854
 - flujo de entrada estándar*, 636
 - flujo de error estándar*, 636
 - flujo de salida estándar*, 636
 - Font**, 431
 - Graphics**, 446
 - ImageList**, 480
 - NetworkStream**, 920
 - PaintEventArgs**, 503
 - Pen**, 582
 - remoto*, 952
 - serializado*, 663
 - WebClient**, 956
 - XmlReaderSettings**, 706
 - XPathDocument**, 716
 - objetos*, 9
 - proxy*, 952
 - TabPage**, 489
 - objetosPorPagar, 370
 - ocultamiento de información*, 11, 94, 291
 - ocultar automáticamente*, 25
 - OK, 434
 - OKCancel, 434
 - One**, 469
 - OnPaint**, 584
 - opción
 - Actualizar referencia Web*, 870
 - Agregar referencia Web...*, 880
 - Cambiar nombre*, 784
 - Completar de la lista desplegable Paso*, 847
 - Iniciar depuración (menú Depurar)*, 388
 - Editar elementos...*, 793
 - Formato > Color de primer plano...*, 790
 - Generar sitio Web*, 879
 - Habilitar paginación*, 857
 - Habilitar Sólo mi código*, 1065
 - Nueva carpeta*, 792
 - Servidor Web (HTTP)*, 789
 - Simple*, 476
 - top (del cuadro combinado Alineación vertical)*, 789
 - Ver diseñador*, 751
 - Windows Media Player*, 621
 - Establecer como proyecto de inicio*, 27
 - Opciones, 54
 - mutuamente exclusivas*, 433
 - Opciones de autoposición... (menú **Diseño**), 786
 - operación*, 108
 - abstracta*, 378
 - de salida física*, 636
 - operaciones*, 11, 291
 - concurrentes*, 510
 - en paralelo*, 510
 - lógicas de salida*, 634
 - operador*
 - binario*, 65
 - condicional (?)*, 119
 - de asignación*, =, 65
 - de asignación compuesto de suma*, +=, 134
 - de conversión (double)*, 130
 - unario*, 130
 - de decremento*
 - postfijo*, 134
 - prefijo*, 134
 - unario*, --, 134
 - de incremento*
 - postfijo*, 134
 - prefijo*, 134
 - unario*, ++, 134
 - INNER JOIN**, 735
 - LIKE**, 733
 - new*, 88, 966
 - punto* (.), 88
 - residuo*, 67
 - ternario*, 119
- operadores*
 - a nivel de bit*, 432
 - aritméticos*, 67

- de asignación compuestos*, 134
 - de igualdad*, 69
 - lógicos*, 164
 - multiplicativos* *, / y %, 130
 - relacionales*, 69
 - operandos*, 65, 66
 - OR**, 738
 - exclusivo lógico booleano* (^), 166
 - inclusivo lógico booleano* (|), 165
 - ordenamiento de un árbol binario*, 990
 - Organización de interoperabilidad de servicios Web (WS-I)*, 876
 - orientada a las acciones*, 11
 - orientadas a la conexión*, 918
 - orientados a objetos*, 11
 - orígenes de datos*, 739
 - out**, 204
 - OutOfMemoryException*, 394
 - OverflowException*, 291
- P**
- página
 - Agregar nueva regla de acceso, 840
 - de contenido*, 838
 - de inicio*, 19
 - principal*, 838, 843
 - páginas Web *accesibles*, 1088
 - PaintEventArgs*, 584
 - palabra clave
 - abstract*, 347
 - AND**, 738
 - AS** de SQL, 852
 - base, 309
 - class**, 50
 - const**, 178, 213
 - delegate*, 419
 - EMPTY**, 690
 - enum**, 195
 - lock**, 518
 - operator*, 374
 - override*, 313, 348
 - params**, 250
 - private**, 93
 - public**, 51, 86
 - readonly**, 288
 - return**, 95
 - SELECT**, 731
 - static**, 87
 - VALUES** de SQL, 737
 - void**, 52, 87
 - virtual* y *abstract*, 321
 - palabras reservadas*, 50
 - Panel**, 428
 - de control*, 1074
 - Opciones de posición, 786
 - Plantillas**, 781
 - Pantalla**, 109
 - Definir parámetros, 856
 - par dirección IP/número de puerto*, 920
 - parámetro*, 89
 - de salida*, 204
 - formal*, 181
 - implícito llamado *value*, 95
 - parámetros de tipo*, 1003
 - paréntesis*, 52
 - anidados*, 68
 - redundantes*, 69
 - pares clave-valor*, 814
 - partición de tiempo, 513
 - paso*
 - de recursividad*, 201
 - por referencia*, 204
 - por valor*, 204
 - PathGradientBrush**, 603
 - PEPS** (*primero en entrar, primero en salir*), 980
 - Perfil básico 1.1 (BP 1.1)*, 876
 - Permitir contenido bloqueado..., 682
 - personajes animados interactivos*, 621
 - pila*, 184, 976
 - de ejecución del programa*, 184
 - de llamadas a los métodos*, 184, 402
 - Pixel**, 591
 - pixeles*, 1087
 - plantillas*, 697, 700
 - plataforma .net*, 3
 - PNG** (Gráficos portables de red), 41
 - poblar el objeto DataSet*, 739
 - Point**, 591
 - polimorfismo*, 342
 - poner el código en línea*, 362
 - pop*, 977
 - por ciento* (%), 733
 - posicionamiento*
 - absoluto*, 786
 - relativo*, 786
 - postback*, 803
 - precioPorArtículo, 365
 - precisión del número*, 102, 130
 - predicados*, 732
 - prefijo
 - de espacio de nombres reservado xml*, 685
 - de etiqueta asp*, 778
 - prefijos de espacios de nombres*, 685
 - presentación*, 1078
 - presionar*, 444
 - primerNombre, 349
 - primero en entrar, primero en salir* (**PEPS**), 291
 - problema del else suelto*, 121
 - proceso controlado por eventos*, 584
 - productor*, 519
 - Profesional**, 849
 - programación*
 - concurrente*, 510
 - estructurada*, 116
 - orientada a objetos*, 11
 - visual*, 18
 - programador de subprocesos*, 513
 - programas más tolerantes a fallas y robustos*, 386
 - prólogo*, 680
 - promoción*, 130
 - promoción de argumentos*, 184
 - propiedad
 - Activation**, 484
 - ActiveMdiChild**, 495
 - AutoGenerateColumns**, 858
 - AutoPostBack**, 855
 - AutoSize**, 39, 40
 - BackColor**, 36
 - de Alternating-RowStyle*, 856
 - de HeaderStyle*, 856
 - BindingSource**, 749
 - CellPadding** del control **GridView**, 856
 - ClipRectangle**, 503
 - Color**, 589
 - Column**, 853
 - ConformsTo**, 876
 - ConnectionString**, 829
 - ContentPlaceHolderID**, 846
 - ContinueDestination-PageUrl**, 847
 - ControlToValidate**, 801
 - Cookies**, 815
 - Count**, 823
 - de Hashtable*, 1033
 - CreateUserText**, 849
 - CreateUserUrl**, 849
 - Current**, 717, 1023
 - Characters** de **mainAgent**, 631
 - Checked**, 460
 - DashCap**, 606
 - DashStyle**, 606
 - DataBindings.Text**, 765
 - DataMember**, 749
 - DataSource**, 748-749
 - DataSourceID**, 829
 - DayOfWeek**, 463
 - Description** del atributo **WebService**, 876
 - Display**, 802
 - DisplayRememberMe**, 849
 - DropDownStyle**, 476
 - Enabled**, 422
 - ErrorMessage**, 801
 - Format**, 462
 - FullName**, 489
 - Graphics**, 503
 - IgnoreWhitespace**, 710
 - ImageIndex**, 485
 - Images**, 485
 - ImageUrl**, 793
 - InsertQuery**, 828
 - Interval**, 504
 - InvokeRequired**, 926
 - IsBackground**, 513
 - IsMdiContainer**, 494
 - KeyChar**, 446
 - Keys** de la clase **HttpSessionState**, 824
 - LargeImageList**, 485
 - Last**, 1040
 - Length**, 217, 549
 - LogoutAction**, 851
 - LogoutText**, 851
 - MasterPageFile**, 845
 - MaxDropDownItems**, 475
 - MdiChildren**, 495
 - MdiParent**, 495
 - Message**, 394
 - Name**, 427, 489, 516, 716
 - Namespace**, 768
 - del atributo WebService*, 876
 - NavigateUrl**, 793
 - Nombre de archivo**, 55
 - Padding**, 424
 - PageSize** de **GridView**, 857
 - Parent**, 489, 710
 - PasswordChar**, 425
 - ReadOnly**, 443
 - Schemas**, 718
 - SelectCommand**, 832
 - SelectedIndex**, 470
 - SelectedIndices**, 470
 - SelectedItem**, 470
 - Session** de la clase **Page**, 817
 - ShortcutKeyDisplayString**, 454
 - ShortcutKeys**, 454
 - ShowShortcutKeys**, 454
 - Size**, 591
 - SizeMode**, 43
 - SmallImageList**, 485
 - static CurrentThread**, 516
 - StatusText** de **WebBrowser**, 951
 - Style**, 431
 - TabIndex**, 422
 - TabPages**, 490
 - Target** a **_blank**, 793
 - Text**, 36, 420
 - TextAlign**, 40
 - Timeout**, 821
 - Title**, 785
 - Today**, 464
 - ValidationEventHandler**, 718
 - ValidationExpression**, 802
 - ValidationType**, 717
 - Value**, 463, 716, 1036
 - View**, 484
 - Visible**, 801
 - propiedades*, 30, 85
 - CanGoBack** y **CanGoForward**, 951
 - Count** y **Capacity**, 1025
 - de sólo lectura* **Previous** y **Next**, 1036
 - Length**, 558
 - MaximumSize** y **MinimumSize**, 961
 - Suelto**, 353
 - protocolo*, 1101
 - de control de transmisión* (**TCP**), 919
 - de datagramas de usuario* (**UDP**), 919
 - de Transferencia de Hipertexto*, 4, 5
 - simple de acceso a objetos* (**SOAP**), 4, 864
 - protocolos*, 919

- proxy*
 - real*, 952
 - transparente*, 952
 - proyecto de inicio*, 27
 - PruebaLibroCalificaciones, 86
 - punto de entrada*, 179
 - Punto de interrupción >
 - Deshabilitar punto de interrupción, 1055
 - Punto de interrupción > Habilitar
 - punto de interrupción, 1056
 - Punto de interrupción > Insertar
 - punto de interrupción, 1054
 - punto de lanzamiento*, 389
 - punto y coma (;)*, 52
 - puntos de interrupción*, 1052
 - push*, 977
- ## Q
- Question, 434
 - Queue y SortedList, 1019
- ## R
- RadioButton, 430
 - raíz*, 479, 680
 - del documento*, 700
 - RanuraDeposito, 109
 - rastreo*
 - de pila*, 389
 - de sesiones*, 809
 - realización*, 365
 - reanudar*, 539
 - recolección de basura, 184
 - recolector de basura*, 282
 - recopilación de requerimientos*, 78
 - recorrer*, 244
 - recorrido en orden de niveles de un árbol binario*, 991
 - recorridos recursivos, 984
 - rectángulo redondeado*, 169
 - recursividad*, 176
 - recursos*, 1079
 - redefinir*, 309
 - RedirectToLoginPage, 851
 - ref*, 204
 - referencia*, 98, 269
 - a entidad de carácter*, 690
 - referencias*
 - a caracteres numéricos*, 1090
 - a una entidad de carácter*, 1090
 - registro*, 635
 - de activación*, 184
 - regla*
 - de acceso*, 840
 - de integridad de entidades*, 730
 - horizontal*, 1091
 - reglas*
 - comerciales*, 776
 - de precedencia de operadores*, 68
 - de promoción*, 185
 - relación "es un"*, 306
 - relación*
 - de composición*, 110
 - de uno a uno*, 111
 - de uno a varios*, 111
 - de varios a uno*, 111
 - "tiene un"*, 110, 279, 306
 - RemoveAt, 471
 - repetición*
 - controlada por un centinela*, 127
 - controlada por un contador*, 122
 - repositorios*, 688
 - representa*, 775
 - representación de datos*, 291
 - requerimientos*, 12
 - del sistema*, 78
 - resaltado de sintaxis por colores*, 53
 - resolución*, 583
 - resolver el problema de la manera correcta*, 78
 - restricción*, 695
 - de clase*, 1006
 - de constructor [newC]*, 1007
 - de interfaz*, 1006
 - de tipo de referencia (class)*, 1007
 - de tipo de valor (struct)*, 1007
 - de tipos*, 1005
 - restricciones múltiples*, 1007
 - Resultados, casilla, 853
 - Retiro, 109
 - RetryCancel, 434
 - Reverse, 1024
 - rombos (diamantes) y pequeños círculos*, 117
 - round-robin, 513
 - ruta general*, 606
- ## S
- SalarioSemanal, 352
 - salto de línea*, 1088
 - Se esperaba, 59
 - sealed*, 362
 - sección
 - Colores personalizados, 591
 - del cuerpo*, 1080
 - del encabezado*, 1080
 - Inicio de sesión, 834
 - llamada Exceptions, 395
 - Página de inicio, 951
 - Validación, 801
 - secuencia*
 - de escape*, 61
 - de mensajes*, 257
 - secuencias*, 984
 - seguridad de tipos en tiempo de ejecución*, 1000
 - Seleccionar
 - dirección URL, 812
 - la página principal, 851
 - tipo de autenticación, 839
 - SELECT
 - configurar instrucción, 825
 - que devuelve filas, 753
 - SelectedIndex, 477
 - SelectedItem, 477
 - sembrar*, 188
 - sensible a mayúsculas y minúsculas*, 50
 - serializable*, 952
 - serialización*
 - de objetos*, 663
 - XML*, 906
 - Serif, 591
 - "server"*, 777
 - Service.asmx, 878
 - Service.cs, 878
 - servicio*
 - de validación*, 1081
 - Web*, 864
 - Web ASP.NET*, 866
 - servicios*
 - public*, 265
 - Web*, 5
 - Servidor
 - de desarrollo ASP.NET*, 788
 - del sistema de nombre de dominios (DNS)*, 773
 - Web*, 773
 - Web (HTTP)*, 879
 - servidores Web*, 1079
 - SessionID, 821
 - seudocódigo*, 12
 - Short, 462
 - signo*
 - de interrogación (?)*, 689
 - mayor que*, >, 676
 - menor que*, <, 676
 - porcentual (%)*, 67
 - signos « y », 97
 - signos diacríticos*, 1072
 - símbolo
 - @ de XPath, 700
 - de decisión*, 118
 - de fusión*, 122
 - de MS-DOS*, 48
 - de visibilidad privada*, 97
 - de visibilidad pública*, 89
 - Símbolo*
 - del sistema*, 48
 - de estado de acción*, 117
 - especiales*, 634
 - simpleContent*, 696
 - sin conexión*, 739, 918
 - sincronizamos*, 510
 - SingleCall, 958
 - Singleton, 958
 - sintetizador de voz*, 1088
 - sistema*, 79
 - de administración de bases de datos (DBMS)*, 636, 726
 - de archivos*, 782
 - de coordenadas*, 582
 - operativo Windows*, 2
 - Sitio Web ASP.NET, 781
 - Size, 424
 - sobrecarga*
 - de métodos*, 198
 - de operadores*, 176, 372
 - software de edición de imágenes, 41
 - SolicitudSaldo, 109
 - Source, 403
 - SQL, 726
 - StackOverflowException, 394
 - StackTrace, 402
 - Stop, 434
 - StreamReader, 636
 - StreamWriter, 636
 - StretchImage, 43
 - struct*, 565, 964
 - subárbol*
 - derecho*, 984
 - izquierdo*, 984
 - subíndice*, 1091
 - de arreglo fuera de rango*, 394
 - submenús*, 452
 - subprocesamiento múltiple, 510
 - subproceso*
 - de ejecución*, 514
 - de interfaz de usuario*, 539
 - de ejecución*, 510
 - en primer plano*, 513
 - en segundo plano*, 513
 - superíndice*, 1091
 - suspender*, 539
 - sustantivos en la especificación de un sistema*, 11
 - sustitutos*, 1071
 - System.Data.DataSet, 743
 - System.Data.OracleClient, 739
 - System.Data.SqlClient, 739
 - System.Drawing, 582
 - System.Drawing.Drawing2D, 582
 - System.Net.Sockets, 919
 - System.Object, 306
 - System.Xml, 705
 - SystemException, 394
- ## T
- tablas*, 1078, 1096
 - hash, 338, 1030
 - tablas de valores*, 242
 - Tabulaciones, 55
 - tamaño*
 - de la fuente*, 591
 - de sangría*, 55
 - de tabulación*, 55
 - Tareas de
 - DropDownList, 793
 - GridView, 825
 - preparación para la terminación*, 282
 - targetNamespace, 693
 - TargetSite, 403
 - tarifaComision, 355
 - tecla Bloq Despl, 632
 - Teclado, 109
 - teclas*
 - de acceso rápido*, 452
 - de función*, 447
 - modificadoras*, 447
 - técnicas de *simulación*, 176
 - tecnología*
 - abierta*, 676
 - ASP.NET 2.0*, 772
 - texto fijo*, 62
 - TextReader, 636

TextureBrush, 603

TextWriter, 636

Thread, 511

TileHorizontal, 497

TileVertical, 497

Times, 462

tipo, 66

AxAgent, 630

AxMediaPlayer, 621

base, 696

de valor de retorno, 87

del atributo, 139

IAgentCtlCharacter, 630

int, 64

MIME image/jpeg, 774

MIME text/plain, 774

TreeNodeCollection, 480

tipos

Boolean, 64, 964

Byte, 64, 964

Char, 64, 964

complejos, 695

de datos abstractos (ADTs), 291

de punto flotante, 102

Decimal, 64, 964

definidos por el usuario, 11

Double, 64, 964

Float, 64

int, 64

Int16, 964

Int32, 964

Int64, 964

Long, 64

SByte, 64, 964

short, 64,

simples, 64, 695

Single, 964

struct, 964

UInt16, 64, 964

UInt32, 964

UInt64, 964

ulong, 64

ushort, 64

título, 1080

Todos los formularios Windows

Forms, 38, 414

Todos los lenguajes, 54

tolerante a fallas, 65

ToLower, 557

ToolStripMenuItem, 453

ToolTip en, 440

top, 1100

TopLeft, TopRight y

BottomCenter, 41

ToUpper, 557

transiciones, 117, 169

transacciones de negocio a negocio (B2B), 865

transferencia de control, 116

ThreadStart, delegado, 511

Trim, 557

U

Ubicación, 55, 782

de memoria, 66

ubicaciones, 66

UEPS (último en entrar, primero en salir), 977

última ocurrencia de un objeto en un arreglo, 1024

UnauthorizedAccessException, 484

unidades

de diseño, 591

de empaquetamiento, 57

UpdateCommand, 832

URL *mailto*, 1085

Usar instrucciones SQL, 752

uso de un búfer, 636

Usuarios anónimos, 842

UTF-8, UTF-16 y UTF-32, 1071

V

validación de datos, 98

validador, 798

de XML, 683

ValidationType.Schema, 717

válido, 678

para el esquema, 691

valor, 66

centinela, 127

de atributo, 684

de bandera, 127

de código, 1071

de desplazamiento, 188

de la PI, 699

de señal, 127

de señuelo, 127

inicial, 144

inicial predeterminado, 95

RGB, 585

unbounded, 695

valores

ARGB, 585

de multiplicidad, 109

value, 1103

“varargs”, 250

variable

de control, 122, 144

de instancia, 92

salarioSemanal, 352

de iteración, 230

static, 283

variables, 63

de instancia sueldo y horas, 353

de instancia, 85

locales, 92

ventana

activa, 414

Ayuda dinámica, 31

de consola, 48

de interfaz de múltiples documentos, 413

hija, 494

índice, 420

Inspección, 1057

Lista de errores, 59

llamada Subprocesos, 517

MDI, 413

oculta, 19

Orígenes de datos, 740

padre, 494

Propiedades, 30

Variables locales, 1057

ventanas en mosaico, 497

ventasBrutas, 355

Ver, menú

Código fuente, 780

Cuadro de herramientas, 38

Lista de errores, 59

Ver diseñador, 783

Ver en el explorador, 788

verdadera, 69

vértice, 1114

vi, 1078

__VIEWSTATE, 804

vinculación

dinámica, 360

estática, 362

postergada, 360

vínculo

Crear reglas de acceso, 839

Formato automático..., 825

Seguridad, 839

Vínculos

de la Página de inicio, 19

internos, 1096, 1110

visibilidad, 298

Visor

de ayuda externa, 33

de ayuda integrada, 33

Vista

de clases, 297

Sesión iniciada, 851

Vistas, 851

vocabulario **XML Schema**,

691

vocabularios de XML, 677

VoiceXML™, 677

volver a lanzar, 400

W

Warning, 434

WAV, video, 619

Web Forms, 812, 851

Web References, 882

Web.config, 788, 842

widgets, 412

width, 424, 1087

“Windows”, 842

Wordpad, 1078

WsiProfiles.BasicProfile1.1, 876

X

XBRL, 677

Xerces, 678

XHTML, 677, 697

xml.apache.org, 678

XPath, 678

XPathExpression, 717

xs1:template, 700

XSL-FO, 678

XSLT, 678, 679

Y

YesNo, 434

YesNoCancel, 434

Z

zona activa circular, 1114

zonas activas, 1112