FEED YOUR BRAIN®

C# 3.0 y LINQ

Aprende a sacar todo el partido a la última versión de .NET

Octavio Hernández Leal
Prólogo de José Manuel Alarcón, Microsoft MVP.

krasis PRESS

C# 3.0 y LINQ

Octavio Hernández Leal

C# 3.0 Y LINQ

ISBN edición electrónica: 978-84-937921-1-4

...I've seen fire and I've seen rain
I've seen sunny days that I thought would never end
I've seen lonely times when I could not find a friend
But I always thought that I'd see you again...
("Fire and rain", James Taylor, 1970)

Dedico este libro a la memoria de mi padre.

Prólogo

Desde las primeras máquinas que podemos considerar ordenadores hasta la actualidad, los lenguajes de programación han recorrido un largo camino. En los años '50 comenzaron a diseñarse los primeros que todavía se utilizan hoy en día como FORTRAN, de 1954, o COBOL, de 1959, que aún forma la base de muchos sistemas bancarios (de hecho los programadores de COBOL todavía son bastante demandados en el mercado laboral).

A finales de los '60 y primeros '70 llegan lenguajes revolucionarios que sientan la base de muchos de los modernos paradigmas como la programación orientada a objetos (Simula, Smalltalk), los lenguajes tipificados (ML), así como el lenguaje C, el primero considerado como "moderno". Incluso a finales de esa década aparece uno destinado a revolucionar la industria: SQL, para el acceso a bases de datos relacionales.

La década de los '80 fue la que trajo la consolidación de ciertos modelos de programación, dando lugar a C++, toda una revolución que combinaba las bondades de C con la orientación a objetos, creando un verdadero estándar para desarrollo sobre múltiples sistemas operativos. También aparece Perl, máximo exponente de los lenguajes dinámicos. Ambos son aún muy utilizados.

La rápida expansión de Internet en los años '90 trajo consigo un campo abonado para que florecieran muchos nuevos lenguajes al amparo del nuevo medio, que requería técnicas y paradigmas distintos. Así, lenguajes como Java y los lenguajes interpretados de *script* (como Javascript), adquirieron una gran popularidad gracias a su funcionamiento conjunto en los navegadores y en los servidores Web. También aparecen grandes éxitos actuales como ASP, PHP, Python o Ruby.

El último gran lenguaje aparecido en la escena mundial ha sido C#, que oficialmente se presenta en el año 2000, ya en pleno siglo XXI. Anders Hejlsberg —el eminente informático danés que está detrás del lenguaje y es padre también de Turbo Pascal y Delphi— dice que se inspiró para su diseño en la sintaxis de C++, pero procurando evitar las limitaciones e ineficiencias de éste y otros lenguajes. C# sigue las tendencias de los modernos lenguajes de desarrollo como la seguridad del código, modularidad, orientación a componentes, programación orientada a aspectos y reflexión, entre otras. Se trata, sin duda, de un lenguaje orientado a las modernas necesidades de los programadores, y una apuesta de futuro.

En todo este tiempo, una de las dificultades recalcitrantes que todavía persisten es la de tratar de conseguir un manejo eficiente, sencillo y genérico de las estructuras de datos. Y es precisamente de esto de lo que se ocupa LINQ, el tema central de este libro. LINQ ofrece una forma genérica, integrada en la sintaxis de los lenguajes .NET, que define la manera de procesar los datos para obtener información, filtrarla, crear proyecciones, etc. Todo ello sin tener que recurrir a lenguajes externos como SQL o algoritmos para procesado de estructuras.

Esto dicho así parece algo bastante nimio para merecer un libro completo o, ya puestos, una nueva versión de un lenguaje de programación. He de confesar que mi reacción instintiva cuando ya hace unos años oí hablar por primera vez de LINQ fue precisamente la de escepticismo. No sopesé bien sus implicaciones. Y cuando vi cómo la usaban, no sólo para tratar estructuras en memoria, sino también para recuperar información desde bases de datos relacionales, mi primer pensamiento fue el de "¡Cuántas aberraciones se van a dar si esto se presenta alguna vez!".

Ahora, unos años después y en gran parte gracias al trabajo de divulgación que ha realizado desde entonces Octavio, el autor de este libro, mi pensamiento no puede ser más opuesto a ese inicial. Creo realmente que estamos ante un concepto absolutamente innovador, que va a marcar un punto de ruptura. Cuando dentro de unos años alguien haga un repaso histórico de los lenguajes de programación como yo he hecho en este prólogo, sin duda alguna marcará 2007 como el año de aparición oficial de LINQ y las extensiones del lenguaje que lleva asociadas. Nunca antes había sido tan fácil trabajar con datos de toda índole, sean éstos colecciones de estructuras, archivos XML, bases de datos MS SQL o, potencialmente de cualquier otro tipo, gracias a la posibilidad de extender el alcance de LINQ mediante proveedores.

Créame, amigo lector, cuando le digo que este libro le va a ayudar a ser más productivo que nunca en sus desarrollos. Y nadie mejor que Octavio para adentrarse en el interesante mundo de LINQ. Pocas personas saben tanto del lenguaje como él, que lleva usándolo en proyectos reales desde las primeras *betas* y ha traducido al castellano gran parte de la documentación oficial, incluyendo las especificaciones técnicas relacionadas con LINQ.

Este libro es seguramente el primero que hay sobre el tema en el idioma de Cervantes, y uno de los primeros en cualquier otra lengua. Además, es también posiblemente uno de los mejores. No pierda la oportunidad de sacar partido a lo que en él se enseña. Estará por delante de su competencia, y no sólo porque ellos usen Java ;-).

José Manuel Alarcón Aguín
MVP de ASP.NET
Director de Krasis – campusMVP

Contenido

Prólogo del autor

Durante la PDC (*Professional Developers Conference*) de octubre de 2005, Microsoft desveló por primera vez las bases del Proyecto LINQ (*Language Integrated Query*-Consultas Integradas en el Lenguaje), una combinación de extensiones a los lenguajes y librerías de código manejado desarrollada con el objetivo central de permitir expresar de manera uniforme las consultas sobre colecciones de objetos, bases de datos relacionales y documentos XML, para ayudar de esa manera a eliminar el conocido "desajuste de impedancia" (*impedance mismatch*) que provocan las diferencias entre los modelos de programación que proponen los lenguajes de propósito general y los lenguajes de acceso a bases de datos relacionales (SQL) y a otros almacenes de datos como los documentos XML (XPath, XQuery). El efecto que causó aquello sobre mí fue el del amor a primera vista; así que empecé a devorar toda la documentación que iba apareciendo, y luego a escribir sobre el tema, principalmente desde las páginas de dotNetManía. El libro que tiene ahora en sus manos es, en buena parte, el resultado de esa labor.

Este libro pretende ofrecer una introducción rápida, pero completa, a las nuevas posibilidades que incorpora la versión 3.0 de C#, incluyendo las relacionadas con la tecnología LINQ y sus librerías asociadas. Asume un conocimiento bastante completo de C#, y requiere el dominio de características como los tipos genéricos o los métodos anónimos, incorporadas al lenguaje en la versión 2.0. Para los programadores que necesiten familiarizarse con esos conceptos, el libro ofrece al principio cuatro capítulos introductorios (la **Parte 0**); otras posibles fuentes de referencia pueden ser el libro *Pro C# with .NET 3.0* (Apress, 2007) de Andrew Troelsen, o por supuesto la imprescindible obra de referencia *The C# Programming Language* (Addison-Wesley, 2003) de Anders Hejlsberg, Scott Wiltamuth y Peter Golde, miembros destacados del equipo que diseñó el lenguaje. Otro consejo al lector es que descargue la especificación de C# 3.0 (una traducción realizada por este autor está disponible en MSDN) y acompañe la lectura de este libro con la del documento de referencia de las nuevas características del lenguaje.

CONTENIDO DEL LIBRO

El libro comienza con una **Introducción,** en la que se presenta el modelo de objetos que se utiliza en la mayor parte los ejemplos. A partir de ahí, el libro consta de las siguientes partes, que en principio deben leerse en secuencia:

- Como se ha mencionado antes, el libro comienza con una **Parte 0** en la que se hace un repaso (que no pretende ser exhaustivo) de las principales características incorporadas al lenguaje en la versión 2.0: los métodos anónimos, los genéricos, los iteradores y los tipos anulables, en las que se apoyan las novedades más relevantes de C# 3.0. Los programadores ya versados en esas

materias podrían obviar este material, aunque esperamos sinceramente que su lectura les aporte algún que otro detalle de interés.

- La **Parte 1,** "Novedades básicas de C# 3.0" presenta las nuevas características que se han incorporado al lenguaje C# en su versión 3.0 y que sientan las bases necesarias para la implementación de LINQ.
- A continuación, la **Parte 2,** "LINQ en C# 3.0" describe los fundamentos lingüísticos de la tecnología LINQ y la librería básica en la que estos fundamentos se apoyan **(LINQ to Objects),** así como los mecanismos previstos para asegurar la extensibilidad del sistema.
- Para cerrar la obra, la **Parte 3** presenta las principales "extensiones" (librerías derivadas) de LINQ: **LINQ to XML** (para el tratamiento de documentos XML), **LINQ to DataSet** (para la manipulación "a là LINQ" de conjuntos de datos en memoria) y **LINQ to SQL** (para el trabajo contra bases de datos relacionales), con ejemplos de cómo estas tecnologías pueden ayudarnos a alcanzar cotas de expresividad nunca antes vistas en la programación contra tan disímiles fuentes de datos.

El código fuente de todos los ejemplos presentados en el libro está disponible en el sitio Web de la editorial, **http://www.krasispress.com.**

Agradecimientos

Quiero dar las gracias desde aquí a aquellos que llevo en mi corazón y que me aportan el estímulo espiritual necesario para seguir adelante cada día: mi esposa Irina y mis hijos Diana y Denis; mi madre; mi hermano *El Negro* (**www.elnegro.com**), sin cuya ayuda y ejemplo nada habría sido posible, y otros muchos familiares y amigos dispersos por tres continentes y que conforman una lista que sería muy larga de enumerar. También quiero agradecer explícitamente por su apoyo y por la confianza que han depositado en mí a mi socio y gran amigo Daniel Alonso; a mis compañeros de *Plain Concepts* (**www.plainconcepts.com**), donde he encontrado el entorno adecuado para crecer compartiendo conocimientos: Pablo Peláez, Iván González, Rodrigo Corral, Unai Zorrilla, Cristian Manteiga, Jorge Serrano, Marco Amoedo y Yamil Hernández; a Paco Marín, editor de la revista *dotNetManía* (**www.dotnetmania.com**); a José Manuel Alarcón, fundador de Krasis (**www.krasis.com**) y de campusMVP (**www.campusmvp.com**); a Juan de Dios Izquierdo, director de CICE (**www.cicesa.com**); a Alfonso Rodríguez, David Carmona, David Salgado y el resto del excelente equipo de soporte a los desarrolladores de Microsoft España; así como a varios grandes especialistas que han contribuido "en directo" a conformar mi visión sobre la programación en general y el mundo .NET en particular, como (en estricto orden cronológico) Serguéi Renin, Sonia Martínez-Miranda, Miguel Katrib, Luciano García, Ian Marteens, Pablo Reyes, Marino Posadas y Guillermo "Guille" Som.

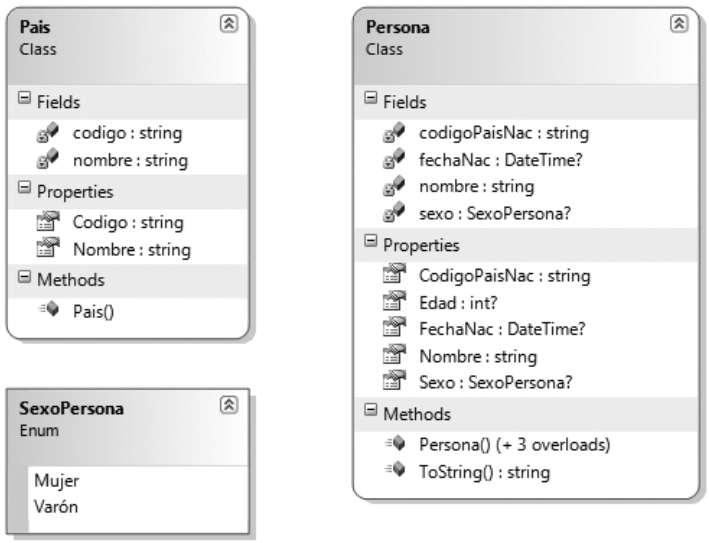
Octavio Hernández

Introducción

Esta introducción tiene como objetivo presentar las principales clases que se utilizarán en la mayoría los ejemplos a lo largo del libro.

EL MODELO DE OBJETOS DE LOS EJEMPLOS

Siguiendo la filosofía KISS (*Keep It Simple, Sir*), la mayor parte de los ejemplos de este libro se apoyará en un modelo de objetos formado por dos clases, `Pais` y `Persona`, que podrían representarse mediante el diagrama de clases que se muestra a continuación. A través de la propiedad `CodigoPaisNac` de las personas, estableceremos relaciones `1:N` entre objetos de ambas clases, lo que aprovecharemos sobre todo a la hora de expresar consultas integradas que combinen información extraída de esos objetos.



La implementación en C# de estas clases se presenta a continuación. Básicamente, el código es C# 1.1; el único elemento de la versión 2.0 que incluye su código fuente son los tipos anulables (vea, por ejemplo, la declaración de la fecha de nacimiento de las personas). Si no sabe qué son los tipos anulables, digamos por ahora que se trata de tipos valor que adicionalmente pueden tomar el valor **null;** en el Capítulo 4 encontrará más información al respecto.

DIRECTORIO DEL CODIGO: CLASES

```
using System;

namespace PlainConcepts.Clases
{
   public class Pais
   {
      #region Campos
      private string codigo, nombre;
      #endregion

      #region Propiedades
      public string Codigo
      {
         get { return codigo; }
         set { codigo = value.ToUpper(); }
      }

      public string Nombre
      {
         get { return nombre; }
         set { nombre = value; }
      }
      #endregion

      #region Constructor
      public Pais(string codigo, string nombre)
      {
         Codigo = codigo;
         Nombre = nombre;
      }
         #endregion
   }
}

using System;

namespace PlainConcepts.Clases
{
   public enum SexoPersona { Mujer, Varón }

   public class Persona
   {
      #region Propiedades
      private string nombre = null;
      private SexoPersona? sexo = null;
      private DateTime? fechaNac = null;
      private string codigoPaisNac = null;

      #endregion
      #region Constructores
```

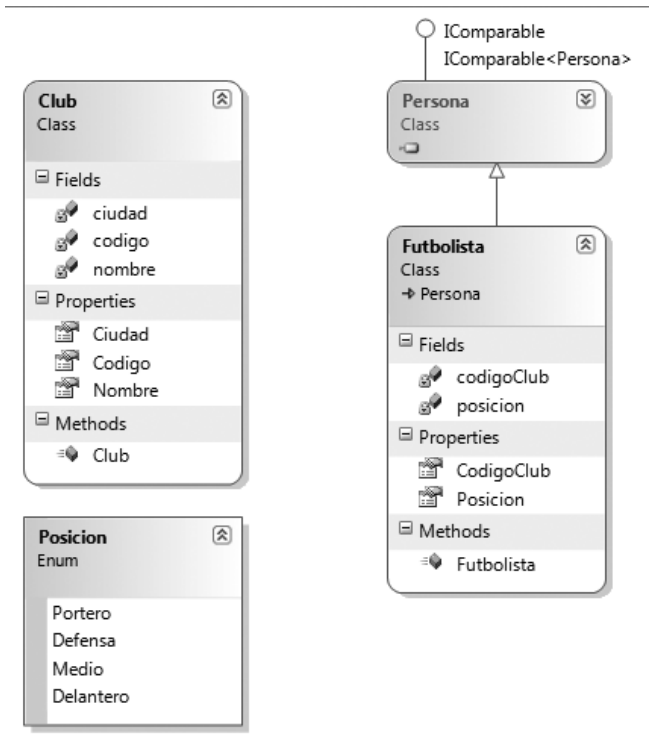
Continúa

```
      public Persona() { }
      public Persona(string nombre, SexoPersona sexo)
      {
         this.Nombre = nombre;
         this.Sexo = sexo;
      }
      public Persona(string nombre, SexoPersona sexo,
         DateTime fechaNac)
         : this(nombre, sexo)
      {
         this.FechaNac = fechaNac;
      }
      public Persona(string nombre, SexoPersona sexo,
         DateTime fechaNac, string codPaisNac)
         : this(nombre, sexo, fechaNac)
      {
         this.CodigoPaisNac = codigoPaisNac;
      }
      #endregion

      #region Propiedades
      public string Nombre
      {
         get { return nombre; }
         set { nombre = value; }
      }
      public SexoPersona? Sexo
      {
         get { return sexo; }
         set { sexo = value; }
      }
      public DateTime? FechaNac
      {
         get { return fechaNac; }
         set { fechaNac = value; }
      }
      public int? Edad
      {
         get
         {
            if (fechaNac != null)
               return (int)
(DateTime.Today.Subtract(fechaNac.Value).TotalDays / 65.25);
            else
               return null;
         }
      }

      public string CodigoPaisNac
      {
         get { return codigoPaisNac; }
```

Continúa

```
            set { codigoPaisNac = value.ToUpper(); }
        }
        #endregion

        #region Métodos
        public override string ToString()
        {
            return (nombre != null ? nombre : "ANONIMO") +
                (sexo!=null?(sexo.Value==SexoPersona.Mujer ?
                    " (M)" : " (V)") : "") +
                (fechaNac != null ? " (" +
                    Edad.Value.ToString("dd/MM/yyyy")+")": "")+
                (codigoPaisNac!=null?"("+CodigoPaisNac+")":
                    "");
        }
        #endregion
    }
}
```

Para los capítulos más avanzados, y en particular el dedicado a LINQ to SQL (la extensión de LINQ para el trabajo con bases de datos relacionales), hemos diseñado un modelo algo más amplio, que incluye clases que permiten almacenar información relativa a los clubes de fútbol y los futbolistas de la Primera División española:



La implementación de las clases en C# es la siguiente:

```
using PlainConcepts.Clases;

namespace PlainConcepts.Clases.Futbol
{
   public class Club
   {
      #region Campos
      private string codigo, nombre, ciudad;
      #endregion

      #region Propiedades
      public string Codigo
      {
         get { return codigo; }
         set { codigo = value.ToUpper(); }
      }
      public string Nombre
      {
         get { return nombre; }
         set { nombre = value; }
      }
      public string Ciudad
      {
         get { return ciudad; }
         set { ciudad = value.ToUpper(); }
      }
      #endregion

      #region Constructor
      public Club(string codigo, string nombre, string ciudad)
      {
         Codigo = codigo;
         Nombre = nombre;
         Ciudad = ciudad;
      }
      #endregion
   }

   Public enum Posicion
   {
      Portero,
      Defensa,
      Medio,
      Delantero
   }
```

Continúa

```
    public class Futbolista: Persona
    {
        #region Campos añadidos
        private string codigoClub;
        private Posicion posicion;
        #endregion

        #region Propiedades añadidas
        public string CodigoClub
        {
            get { return codigoClub; }
            set { codigoClub = value.ToUpper(); }
        }
        public Posicion Posicion
        {
            get { return posicion; }
            set { posicion = value; }
        }
        #endregion

        #region Constructor
        public Futbolista(string nombre, DateTime fechaNac,
            string codigoPaisNac,
            string codigoClub, Posicion posicion):
            base(nombre, SexoPersona.Varón, fechaNac, codigoPaisNac)
        {
            this.CodigoClub = codigoClub;
            this.Posicion = posicion;
        }
        #endregion
    }
}
```

Con el código de los ejemplos que acompañan al libro se ofrece una base de datos de SQL Server 2005 que implementa el modelo anterior e incluye a los 400+ jugadores que participaron en la Liga 2006-2007, ganada por el Real Madrid (aunque vaya por delante que el equipo con el que simpatizó es el Rayo Vallecano, que ahora vaga por la Segunda División B).

Acompañando a las definiciones de tipos anteriores, se suministra una clase estática que carga en tres colecciones genéricas en memoria toda la información sobre los países, clubes y futbolistas:

```
public static class DatosFutbol
{
     public static List<Club> Clubes = new List<Club>();
     public static List<Pais> Paises = new List<Pais>();
     public static List<Futbolista> Futbolistas =
        new List<Futbolista>();

     private static string conStr =
        "Data Source=.\\SQLExpress;Initial Catalog=FUTBOL2006;" +
        "Integrated Security=True";

     static DatosFutbol()
     {
        using (SqlConnection con = new SqlConnection(conStr))
        {
           con.Open();
           // Paises
           using (SqlCommand cmd = new SqlCommand(
              "SELECT Codigo, Nombre FROM Pais", con))
           {
              using (SqlDataReader rdr = cmd.ExecuteReader())
              {
                 while (rdr.Read())
                 {
                    Paises.Add(new Pais(
                      rdr[0].ToString(), rdr[1].ToString()));
                 }
              }
           }
           // Clubes
           using (SqlCommand cmd = new SqlCommand(
              "SELECT Codigo, Nombre, Ciudad FROM Club", con))
           {
           using (SqlDataReader rdr = cmd.ExecuteReader())
           {
              while (rdr.Read())
              {
                 Clubes.Add(new Club(
                    rdr[0].ToString(), rdr[1].ToString(),
                    rdr[2].ToString()));
              }
           }
        }
        // Futbolistas
        using (SqlCommand cmd = new SqlCommand(
```

Continúa

```
            @"SELECT Nombre, FechaNacimiento, CodigoPaisNacimiento,
CodigoClub, Posicion FROM Futbolista", con))
        {
            using (SqlDataReader rdr = cmd.ExecuteReader())
            {
               while (rdr.Read())
               {
                  Futbolistas.Add(new Futbolista(
                     rdr[0].ToString(),
                     DateTime.Parse(rdr[1].ToString()),
                     rdr[2].ToString(), rdr[3].ToString(),
                     ObtenerPosicion(rdr[4].ToString()[0])));
               }
            }
        }
     }
   }

   private static Posicion ObtenerPosicion(char ch)
   {
      switch(ch)
      {
            case 'P':
               return Posicion.Portero;
            case 'D':
               return Posicion.Defensa;
            case 'M':
               return Posicion.Medio;
            case 'L':
               return Posicion.Delantero;
            default:
               throw new ArgumentException("Posición ilegal");
      }
   }
}
```

A partir del Capítulo 10, ejecutaremos consultas integradas contra estos contenedores genéricos; mientras que en los capítulos 13 y 14 utilizaremos directamente la base de datos.

Conceptos preliminares

CAPÍTULO 1

Métodos anónimos

Para lograr una comprensión cabal de las posibilidades que pondrán a disposición del programador .NET 3.5, C# 3.0 y LINQ, es necesario dominar diversos recursos lingüísticos y de librería sobre los que estas tecnologías se apoyan y que fueron incluidos en .NET Framework y los lenguajes Microsoft para la plataforma desde la aparición de .NET 2.0.

En esta primera sección (denominada "Parte 0", en parte porque describe conceptos preliminares a los que se tratarán con más lujo de detalles posteriormente, y en parte porque es tradicional en los lenguajes descendientes de C contar a partir de 0), daremos un breve repaso a esos recursos, como preparación para una mejor asimilación de los conceptos que se describirán en el núcleo central del libro.

Estos recursos son:

- Los métodos anónimos, que permiten especificar directamente el código de un método al que hará referencia un delegado. Una alternativa aún más avanzada de especificación de métodos anónimos, las expresiones lambda, ha sido incorporada en C# 3.0.
- Los genéricos, que hacen posible emitir definiciones reutilizables de tipos y métodos parametrizados con respecto a uno o más tipos de datos.
- Los iteradores, que posibilitan la especificación concisa de mecanismos para la iteración perezosa o bajo demanda sobre los elementos de una secuencia.
- Los tipos anulables, que hacen posible utilizar la semántica del valor nulo propia de los tipos referencia también sobre los tipos valor.

La organización secuencial de los temas no tiene relación alguna con la relevancia relativa de los mismos; intenta simplemente minimizar la cantidad de "referencias hacia delante" en la presentación de los conceptos. En este primer capítulo nos ocuparemos, pues, de los métodos anónimos, un tema relativamente sencillo.

1.1. BREVE REPASO A LOS DELEGADOS Y EVENTOS EN C# 1

En la versión original de C#, la manera de asociar un delegado a un evento era mediante la instanciación explícita del delegado. Si teníamos, por ejemplo, una ventana con

un botón en una sencilla aplicación Windows Forms, el mecanismo sintáctico exigido para asociar código a la pulsación del botón (evento `Click`) era el siguiente:

```
// asociación de instancia de delegado a evento
btn.Click += new EventHandler(Saludo);
```

donde el método `Saludo()` debía estar definido explícitamente de antemano:

```
// método con la firma adecuada (gestor de evento)
public void Saludo(object sender, EventArgs e)
{
   MessageBox.Show("Saludos desde C#");
}
```

La necesidad de declarar de manera explícita el método gestor de eventos no es generalmente muy cómoda, ya que distraerá a quien lea este código fuente, que tendrá que ir a mirar la implementación de `Saludo()` para saber qué ocurrirá cuando el evento se produzca (poner nombres adecuados a las entidades de nuestro código ayuda, pero no siempre es suficiente).

Adicionalmente, con mucha frecuencia tales métodos se definen para ser llamados desde un único sitio del programa, y en ese sentido es un trabajo innecesario tener que definir de manera independiente un método, a sabiendas de que no va a ser reutilizado nunca más.

El programador debe, además, garantizar que la instancia de delegado que asocie al evento cumpla con la firma necesaria, predefinida de antemano. En el código anterior, esa firma viene determinada por el tipo `System.EventHandler`.

Veamos ahora otro ejemplo algo más complejo, en el que se definen eventos a medida. Con el fin de declarar un evento en una clase llamada `Empleado` (que hereda de la clase `Persona` que hemos presentado en la Introducción), se ha definido el siguiente tipo delegado:

DIRECTORIO DEL CODIGO: EJEMPLO01 _ 01

```
public delegate void Exclamacion_CambioSalario(object sender,
   CambioSalarioEventArgs e);
```

La implementación de las clases es la siguiente:

```
public class Empleado : Persona
{
   // campos
   private string string empresa = null;
```

Continúa

```
   private decimal decimal salario;

   // constructores
   public Empleado(string nombre, SexoPersona sexo, string empresa,
      decimal salario)
      : base(nombre, sexo)
   {
      this.Empresa = empresa;
      this.salario = salario;
   }

   // propiedades
   public string Empresa
   {
      get { return empresa; }
      set
      {
         if (value == null)
            throw new Exception("Empresa obligatoria");
         else
            empresa = value;
      }
   }
   public decimal Salario
   {
      get { return salario; }
      set
      {
         decimal anterior = salario;
         salario = value;
         if (salario > anterior &&
            ExclamacionAlSubirSalario != null)
         {
            // disparar evento
            ExclamacionAlSubirSalario(this, new
               CambioSalarioEventArgs(anterior, salario));
         }
      }
   }

   // métodos
   public override string ToString()
   {
      return base.ToString() + Environment.NewLine +
         "Empresa: " + empresa + " Salario: " +
         salario.ToString("#,##0.00");
   }

   // eventos
   public event Exclamacion _ CambioSalario
      ExclamacionAlSubirSalario;
}
```

y la clase para los argumentos del evento a medida es la siguiente:

```
public class CambioSalarioEventArgs: EventArgs
{
   public decimal SalarioAntes;
   public decimal SalarioDespues;
   //
   public CambioSalarioEventArgs(decimal antes,
      decimal despues)
   {
      SalarioAntes = antes;
      SalarioDespues = despues;
   }
}
```

A continuación se muestra un fragmento de código cliente en el que se asigna el evento de cambio de salario a un empleado:

```
// gestor de evento
private static void exclamacion(object sender,
   CambioSalarioEventArgs e)
{
   Console.WriteLine(((Persona)sender).Nombre +
      " está contento!");
   Console.WriteLine("Antes ganaba: " +
      e.SalarioAntes.ToString("#,##0.00"));
   Console.WriteLine("Ahora gana:    " +
      e.SalarioDespues.ToString("#,##0.00"));
}

// código de prueba
private static void Exclamacion()
{
   Empleado pepe = new Empleado("Juan Antonio", SexoPersona.Varón,
      "LA MEJOR EMPRESA", 1075M);
   // asociación de delegado al evento
   pepe.ExclamacionAlSubirSalario +=
      new Exclamacion_CambioSalario(exclamacion);
   // cambio a la propiedad (provoca el evento)
   pepe.Salario = decimal.Round(1.03M * pepe.Salario, 2);

   Console.ReadLine();
}
```

Al igual que en el caso del botón, la asociación del evento con el método a llamar se realiza mediante una construcción explícita de una instancia de delegado que "apunta" al método `Exclamación()`, definido con el juego de parámetros adecuado. Observe que este último método es estático, lo cual no impide que sea utilizado como gestor de eventos.

1.2. LOS MÉTODOS ANÓNIMOS DE C# 2.0

C# 2.0 introdujo una nueva sintaxis que vino a simplificar la manera en que los desarrolladores pueden especificar el código a ejecutar cuando se dispare un evento: los **métodos anónimos.** Básicamente, los métodos anónimos son un ejemplo de lo que se conoce como *azúcar sintáctico:* características que vienen a "endulzar" la sintaxis del lenguaje, ahorrando de paso al programador el tecleo de una buena cantidad de código, que se genera automáticamente por detrás del telón.

La esencia de los métodos anónimos está en la indicación *in situ* del código del gestor del evento, para que no sea necesario definir un método explícitamente. El compilador se encarga si es necesario de deducir, utilizando un mecanismo conocido como **inferencia de tipos,** la firma que debe tener el método y de crearlo por nosotros. En el primer ejemplo del epígrafe anterior, la programación del evento `Click` del botón quedaría así:

DIRECTORIO DEL CODIGO: EJEMPLO01 _ 02

```
// con método anónimo
btn.Click += delegate {
   MessageBox.Show("Saludos desde C#");
};
```

La especificación de un método anónimo comienza con la palabra reservada **`delegate`,** a la que sigue el bloque de código con la funcionalidad deseada. Observe que aquí la llave de cierre debe ir seguida de un punto y coma, el terminador de instrucciones en los lenguajes de la familia de C.

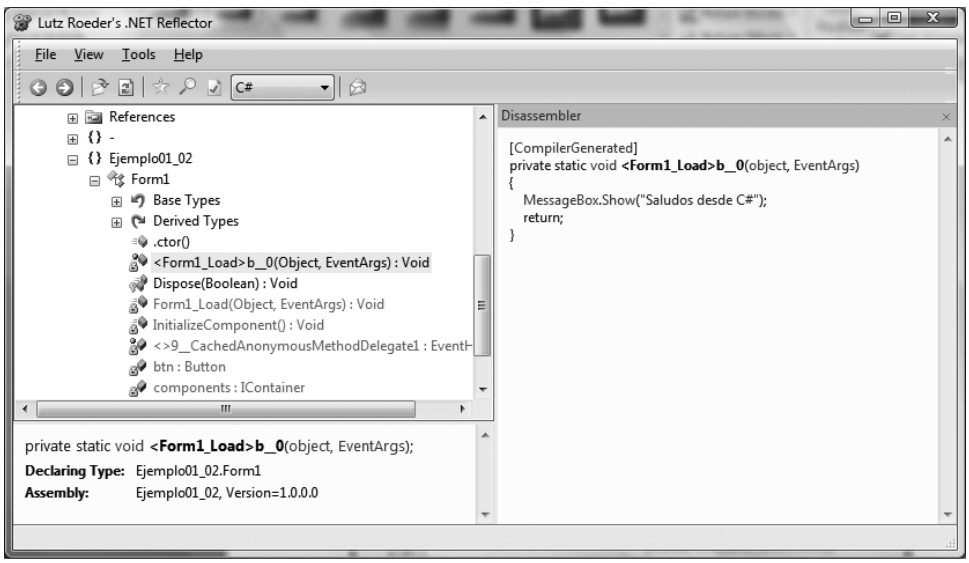


Figura 1.1.

Utilizando una herramienta como el excelente **Reflector,** de Lutz Roeder, se puede ver cómo el compilador genera automáticamente un método con la firma adecuada y un nombre elegido por él. En general, cada vez que en C# se habla de que algo es anónimo, realmente eso significa que su nombre es generado automáticamente por el compilador, y que por tanto es inaccesible para nuestro código fuente.

Opcionalmente, se pueden especificar los argumentos de la llamada a continuación de la palabra reservada **`delegate`;** se hace imprescindible hacer uso de esta posibilidad cuando se desea utilizar dentro del código alguno de los parámetros, como es el caso en nuestro segundo ejemplo:

```
pepe.ExclamacionAlSubirSalario +=
   delegate (object s2, CambioSalarioEventArgs e2)
   {
      MessageBox.Show(((Empleado)s2).Nombre +
         " está contento!");
      MessageBox.Show("Antes ganaba: " +
         e2.SalarioAntes.ToString("#,##0.00"));
      MessageBox.Show("Ahora gana:    " +
         e2.SalarioDespues.ToString("#,##0.00"));
   };
```

1.3. PASO DE DELEGADOS ANÓNIMOS COMO PARÁMETROS

Como cabría esperar, los delegados anónimos pueden utilizarse no solamente en contextos relacionados con la asignación de delegados a eventos, sino también en cualquier otro contexto en el que haga falta una instancia de delegado. Un ejemplo típico podría ser el relacionado con la creación de un hilo de ejecución. Los constructores de la clase `System.Threading.Thread` solicitan un parámetro de uno de los tipos `ThreadStart` o `ParameterizedThreadStart`, definidos de la siguiente forma:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);
```

El siguiente fragmento de programa despliega un hilo de ejecución paralelo que incrementa el valor de un contador y lo muestra en una etiqueta en la ventana principal de la aplicación. El código a ejecutar en el hilo se suministra en un método anónimo. Y como las actualizaciones de controles desde hilos secundarios en las aplicaciones Windows Forms deben realizarse mediante llamadas al método

Invoke() del control, debido a la no-reentrancia de esta librería, hemos hecho uso de otro método anónimo para indicar el código a ejecutar de manera segura:

```
private void Hilo(object sender, EventArgs e)
{
   new Thread(
      delegate () {
         int i = 1;
         while (true)
         {
            lbl.Invoke(
               (MethodInvoker)
               delegate {
                  lbl.Text = i.ToString();
               });
               Thread.Sleep(1000);
               i++;
            }
         }).Start();
}
```

Observe la necesidad de aplicar una conversión explícita al segundo delegado anónimo para convertirlo al tipo `MethodInvoker`, definido en `System.Windows.Forms` como:

```
public delegate void MethodInvoker();
```

(pudo haberse utilizado `ThreadStart`, que tiene la misma firma, o incluso `ParameterizedThreadStart`). El argumento del método `Control.Invoke()` es de tipo `System.Delegate`, y en ese caso (aun cuando este tipo es "el padre" de todos los delegados) el compilador no puede inferir cuáles deben ser los tipos de los parámetros o el valor de retorno del método a generar. Mediante la conversión explícita se le suministra al compilador esa indicación.

1.4. ACCESO A LAS VARIABLES DE ÁMBITOS EXTERNOS

Una de las posibilidades más interesantes y útiles que ofrecen los métodos anónimos es la de acceder a las variables del contexto que rodea al sitio en el que están definidos. Esto incluye (pero no está limitado a) las variables locales del método dentro del que están definidos, los parámetros del mismo, siempre que sean de entrada (o

sea, no **out** ni **ref**) y los campos de la clase a la que pertenece el método que a su vez contiene al método anónimo.

Como ejemplo, supongamos que el código cliente quisiera sumar los salarios de los empleados que reciben aumentos. En tal caso, podría declarar una variable local al método que contiene al método anónimo y utilizarla para acumular los salarios, de la siguiente forma:

```
// acceso al ámbito externo
private void Exclamacion2(object sender, EventArgs e)
{
   Empleado[] arr = {
      new Empleado("Octavio", SexoPersona.Varón, "", 1440M),
      new Empleado("Sergio", SexoPersona.Varón, "", 1210M),
      new Empleado("Denis", SexoPersona.Varón, "", 950.55M)
   };

   decimal suma = 0;
   foreach (Empleado emp in arr)
   {
      emp.ExclamacionAlSubirSalario +=
         delegate (object s2, CambioSalarioEventArgs e2)
         {
         MessageBox.Show(((Persona)s2).Nombre +
            " está contento!");
         MessageBox.Show("Antes ganaba: " +
            e2.SalarioAntes.ToString("#,##0.00"));
         MessageBox.Show("Ahora gana:    " +
            e2.SalarioDespues.ToString("#,##0.00"));
         // acceso a variable de ámbito externo
         suma += e2.SalarioDespues;
      };
      emp.Salario = decimal.Round(1.03M * emp.Salario, 2);
   }
   MessageBox.Show("El salario total es: " +
      suma.ToString("###,##0.00"));
}
```

En general, a las funciones que son capaces de acceder a las variables del entorno que las rodea se les conoce como **clausuras.** Las clausuras son un recurso tradicional de la programación funcional, que aparecieron por primera vez en un lenguaje orientado a objetos "popular" con C#.

Nuevamente, es interesante observar cómo el compilador implementa esta característica. Básicamente, primero genera automáticamente una clase en la que reserva espacio para las variables externas, que se dice que son **capturadas.** Esta clase incluye además un método cuyo cuerpo es el del método anónimo.

```
[CompilerGenerated]
private sealed class <>c__DisplayClass6
{
// Fields
public decimal suma;

// Methods
public void <Exclamacion2>b__4(object s2,
     CambioSalarioEventArgs e2)
{
     MessageBox.Show(((Persona) s2).Nombre +
        " está contento!");
     MessageBox.Show("Antes ganaba: " +
        e2.SalarioAntes.ToString("#,##0.00"));
     MessageBox.Show("Ahora gana:    " +
        e2.SalarioDespues.ToString("#,##0.00"));
     this.suma += e2.SalarioDespues;
}
```

Al principio del método que contiene al método anónimo, se crea una instancia de esta nueva clase. El compilador traduce entonces todas las referencias a la variable capturada, en nuestro caso `suma`, en referencias al campo de la instancia, y utiliza como delegado la referencia al método de la instancia. Todo eso y aún más se puede comprender analizando el ensamblado obtenido con Reflector, aunque hará falta descender al nivel de lenguaje intermedio (IL).

CAPÍTULO 2

Genéricos

Sin lugar a dudas, la más importante novedad que incorporaron los lenguajes Microsoft para la versión 2.0 de .NET Framework fue la **genericidad.** Se trata de una característica que no se implementa exclusivamente a nivel de lenguaje de programación, sino que requirió de extensiones a la máquina virtual de .NET para garantizar numerosas ventajas con relación a lo que hubiera podido lograrse únicamente mediante recursos de lenguaje; la contraposición de los tipos genéricos de C# con las clásicas plantillas (*templates*) de C++ constituye el ejemplo clásico en este sentido.

En este capítulo veremos las posibilidades que se abren ante el programador gracias a la capacidad de expresar definiciones genéricas con respecto a uno o más tipos de datos.

2.1. CADA OVEJA CON SU PAREJA

Introduciremos el recurso de la genericidad a través de un ejemplo sencillo. Con relativa frecuencia, nos surge la necesidad de definir un tipo de datos que aglutine a un par de variables de un mismo tipo. Ejemplos típicos de diferentes ámbitos del mundo real son un punto en la pantalla (determinado por una pareja de números enteros *x* e *y*), un número complejo (que tiene una parte real *a* y una parte imaginaria *b*, ambos números reales), o un matrimonio (compuesto por dos personas).

En la versión 1 de C# habría que haber utilizado tres definiciones totalmente distintas para esos tres tipos de datos:

```
public class Punto
{
   private int x, y;
   public int X { get { return x; } set { x = value; } }
   public int Y { get { return y; } set { y = value; } }
}
```

Continúa

```
public class Complejo
{
   private double a, b;
   public double A { get { return a; } set { a = value; }
   public double B { get { return b; } set { b = value; }
}

public class Matrimonio
{
   private Persona marido, mujer;
   public Persona Marido {
      get { return marido; }
      set { marido = value; }
   }
   public Persona Mujer {
      get { return mujer; }
      set { mujer = value; }
   }
}
```

Ignoraremos, por el momento, el hecho de que para definir estos tipos (especialmente los dos primeros, compuestos íntegramente por campos de tipos valor) pudieron haberse utilizado estructuras en lugar de clases, algo que es básicamente irrelevante de cara a la discusión de este capítulo. De hecho, la práctica totalidad de lo que ejemplificaremos aquí mediante clases es aplicable también a las estructuras. En particular, es perfectamente factible definir estructuras genéricas.

La aparición de la versión 2.0 de .NET Framework y la "ola" de cambios en los lenguajes que acompañó a ésta hicieron posible escribir definiciones **genéricas** como la siguiente:

DIRECTORIO DEL CODIGO: EJEMPLO02 _ 01

```
public class ParOrdenado<T>
{
   // campos
   private T primero = default(T);
   private T segundo = default(T);

   // propiedades
   public T Primero
   {
      get { return primero; }
```

Continúa

```
      set { primero = value; }
   }
   public T Segundo
   {
      get { return segundo; }
      set { segundo = value; }
   }

   // constructores
   public ParOrdenado(T primero, T segundo)
   {
      Primero = primero;
      Segundo = segundo;
   }

   // métodos
   public override string ToString()
   {
      return "(" + Primero.ToString() + ", " +
         Segundo.ToString() + ")";
   }

   // indizador (solo lectura)
   public T this[int i]
   {
      get
      {
         switch (i)
         {
            case 0:
               return Primero;
            case 1:
               return Segundo;
            default:
               throw new Exception("ParOrdenado[]");
         }
      }
   }
}
```

El elemento que juega un papel esencial en la definición anterior es el **parámetro de tipo** `T`. No es ni más ni menos que un sustituto para un tipo concreto al que se desee aplicar la "plantilla" establecida en la declaración genérica. De esta manera, se hace posible reutilizar una misma definición para diferentes tipos de datos, una

de las principales ventajas de la genericidad. Por ejemplo, en base a la declaración anterior se podrían definir variables como las siguientes:

```
// un par de enteros
ParOrdenado<int> p1 = new ParOrdenado<int>(4, 5);
Console.WriteLine(p1.ToString());

// un par de cadenas
ParOrdenado<string> p2 =
   new ParOrdenado<string>("a", "b");
Console.WriteLine(p2.ToString());
```

En las declaraciones anteriores, el parámetro de tipo `T` ha sido "sustituido" por **`int`** y **`string`**, respectivamente. Probablemente más útiles serán las siguientes definiciones de clases:

```
// clase Punto - implementación básica
public class Punto : ParOrdenado<int>
{
   public Punto(int x, int y): base(x, y)
   {
   }

   // desplazar punto en el plano
   public void Mover(int dx, int dy)
   {
      Primero += dx;
      Segundo += dy;
   }
}

// clase Complejo - algo más elaborada que la anterior
// En particular, implementa una semántica de comparación
// de tipo valor
public class Complejo: ParOrdenado<double>
{
   public Complejo(double a, double b): base(a, b)
   {
   }

   // operaciones aritméticas
   public static Complejo operator +(Complejo c1,
      Complejo c2)
   {
      return new Complejo(
         c1.Primero + c2.Primero,
         c1.Segundo + c2.Segundo);
```

Continúa

```
}
public static Complejo operator *(Complejo c1,
   Complejo c2)
{
   return new Complejo(
      c1.Primero * c2.Primero - c1.Segundo * c2.Segundo,
      c1.Primero * c2.Segundo + c1.Segundo * c2.Primero);
}

// conversión implícita
public static implicit operator Complejo(double d)
{
   return new Complejo(d, 0);
}

// igualdad
public override bool Equals(object obj)
{
   Complejo cObj = obj as Complejo;
   if ((object) cObj == null)
      return false;
   else
      return Primero == cObj.Primero &&
         Segundo == cObj.Segundo;
}
public override int GetHashCode()
{
   return ((int) Math.Sqrt(Primero*Primero +
      Segundo*Segundo)).GetHashCode();
}
public static bool operator ==(Complejo a, Complejo b)
{
   if (System.Object.ReferenceEquals(a, b))
      return true;
   if ((object)a == null || (object)b == null)
      return false;
   return a.Primero == b.Primero &&
        a.Segundo == b.Segundo;
}
public static bool operator !=(Complejo a, Complejo b)
{
   return !(a == b);
}

// presentación como cadena
public override string ToString()
{
```

Continúa

```
      return Primero.ToString() + " + " +
         Segundo.ToString() + "*i";
   }
}

// clase Matrimonio
// Se utiliza un tipo anulable (cap. 4) para la fecha
public class Matrimonio: ParOrdenado<Persona>
{
   public DateTime? FechaBoda { get; set; }
   public string LugarBoda { get; set; }

   // constructor(es)
   public Matrimonio(Persona p1, Persona p2,
      DateTime? fecha, string lugar)
      : base(p1, p2)
   {
      FechaBoda = fecha;
      LugarBoda = lugar;
   }
   public Matrimonio(Persona p1, Persona p2)
      : this(p1, p2, null, null)
   {
   }
   public Matrimonio(
      string nombre1, SexoPersona sexo1,
      string nombre2, SexoPersona sexo2,
      DateTime fecha, string lugar)
      : this(new Persona(nombre1, sexo1),
             new Persona(nombre2, sexo2), fecha, lugar)
   {
   }

   // propiedades
   public Persona Marido
   {
      get { return Primero; }
      set { Primero = value; }
   }
   public Persona Mujer
   {
      get { return Segundo; }
      set { Segundo = value; }
   }
}
```

Las tres clases anteriores son tipos construidos **cerrados** (como se les denomina en la literatura oficial) que heredan de `ParOrdenado<T>` (un tipo construido **abierto**). A pesar de que heredan de una clase genérica, ellas mismas ya no lo son, puesto que en ellas el tipo `T` se ha "instanciado" a **`int`**, **`double`** y `Persona`, respectivamente. Pero por supuesto, no es imprescindible "concretar" los parámetros de tipo de una clase para hacer uso de ella mediante la herencia o composición. El ejemplo clásico en este sentido son las colecciones genéricas. Considere por ejemplo la siguiente clase, que implementa una lista genérica de objetos de un tipo `T`:

```
public class Lista<T>
{
   private ArrayList lista = new ArrayList();

   public void Add(T t)
   {
      lista.Add(t);
   }

   public int Count
   {
      get { return lista.Count; }
   }

   public T this[int i]
   {
      get { return (T) lista[i]; }
   }
}
```

Este ejemplo es una muestra de los beneficios que aportan los tipos genéricos de cara a una programación más segura gracias al control de tipos estricto. Aquí, los elementos de la lista se almacenan internamente en un `ArrayList` básico, pero el programador queda protegido contra cualquier intento erróneo de insertar en la lista un elemento de un tipo incorrecto. Por supuesto, en la vida real no hay que crear definiciones tan básicas: la librería de clases de .NET ya ofrece una buena cantidad de colecciones genéricas predefinidas, que describiremos más adelante en este capítulo.

Apoyándonos en los ejemplos anteriores, podemos definir aún una clase más:

```
public class Familia
{
   private Matrimonio padres;
   private Lista<Persona> hijos = new Lista<Persona>();
   public Familia(Persona padre, Persona madre,
      params Persona[] hijos)
```

Continúa

```
    {
        padres = new Matrimonio(padre, madre);
        foreach (Persona p in hijos)
            this.hijos.Add(p);
    }

    public Persona Padre
    {
        get { return padres.Marido; }
    }
    public Persona Madre
    {
        get { return padres.Mujer; }
    }
    public Lista<Persona> Hijos
    {
        get { return hijos; }
    }

    public override string ToString()
    {
        string s = "Padre: " + Padre.ToString() + "\n" +
            "Madre: " + Madre.ToString() + "\n";
        if (hijos.Count > 0)
        {
            s += "Hijos:\n";
            for (int i = 0; i < hijos.Count; i++ )
                s += "   " + hijos[i].ToString() + "\n";
        }
        return s;
    }
}
```

Aquí los hijos de un matrimonio se almacenan en una lista de objetos de tipo `Persona`. El constructor de la clase `Familia` acepta una cantidad variable de parámetros, donde los (posibles) hijos comienzan a partir de la tercera posición. Observe que tal como está definida la clase `Lista<T>`, no es posible recorrer sus elementos utilizando la sentencia **`foreach`**; la clase no implementa el "patrón **`foreach`**" ni ninguna de las interfaces `IEnumerable` o `IEnumerable<T>` (su equivalente genérica). Más adelante resolveremos esa deficiencia.

A continuación, se presenta un ejemplo de utilización de esta clase:

```
Familia f = new Familia(
    new Persona("Octavio"), new Persona("Irina"),
    new Persona("Denis"), new Persona("Diana"));
Console.WriteLine(f);
```

2.2. LA EXPRESIÓN DEFAULT(T)

En el código fuente de la clase `ParOrdenado<T>` se hace uso de la expresión **`default`**`(T)` para indicar que los campos `primero` y `segundo` deben inicializarse al valor predeterminado del tipo genérico `T`, del cual a priori se desconoce si es un tipo valor o un tipo referencia. Si el parámetro de tipo resulta ser un tipo referencia, **`default`**`(T)` produce el valor **`null`** convertido a ese tipo. Si, por el contrario, `T` se traduce en un tipo valor, la expresión producirá el valor predeterminado del tipo correspondiente, que se obtiene aplicando un patrón binario de ceros al espacio de memoria que ocupa una variable de ese tipo valor.

2.3. LA IMPLEMENTACIÓN DE LA GENERICIDAD EN .NET

Como hemos mencionado al principio del capítulo, la implementación de la genericidad que se ha incorporado en .NET Framework requiere la interacción armoniosa del compilador y la plataforma; no es estrictamente una característica de los lenguajes. Sin entrar en muchos detalles de bajo nivel, diremos que en .NET, a diferencia de Java, la personalización de los tipos genéricos para obtener tipos construidos se realiza de manera diferente en dependencia de si el parámetro de tipo va a ser sustituido por un tipo valor o un tipo referencia. En particular, si el tipo "de destino" es un tipo valor, se crea una versión específica del tipo genérico para ese tipo concreto, y esa especialización es la que se utiliza en lo sucesivo. Esto se traduce en una ganancia sensible en el rendimiento, debido a que se hace posible evitar las constantes operaciones de *boxing* y *unboxing* que serían necesarias si se operara sobre una implementación común del tipo genérico. En el caso de los tipos referencia, tal especialización no es necesaria, y el motor de ejecución reutiliza una misma versión del tipo.

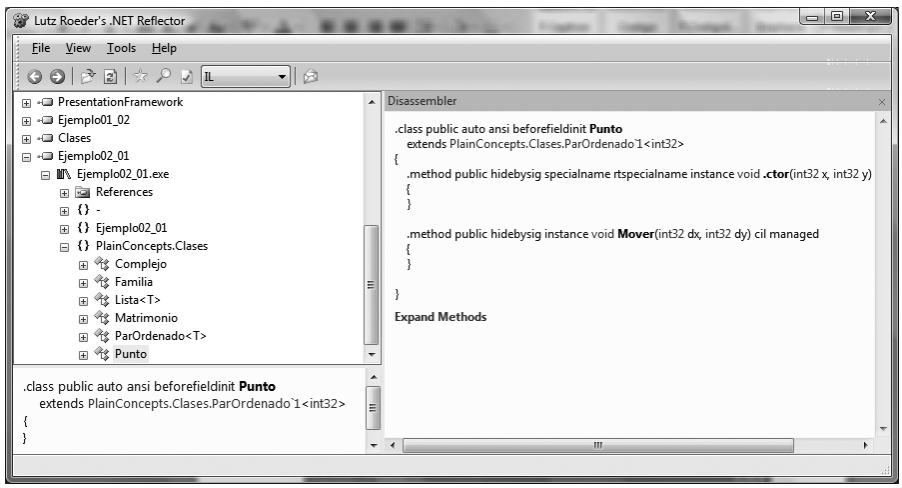


Figura 2.1.

Mediante Reflector es posible ver a nivel de código IL cómo las clases `Punto` y `Complejo` de nuestro ejemplo no heredan directamente de `ParOrdenado` (aunque eso sea lo que hemos indicado), sino de herederas especializadas de ésta, preparadas para trabajar con **`double`** (`float64`, en IL) e **`int`**, respectivamente.

2.4. COLECCIONES GENÉRICAS

Como ya hemos mencionado, una de las áreas donde la genericidad encuentra un campo natural para su uso es en la implementación de toda clase de colecciones genéricas. En ese sentido, los creadores de .NET ya se han preocupado por dotar a la librería de clases base de un amplio conjunto de clases e interfaces genéricas listas para ser utilizadas, todas agrupadas bajo un espacio de nombres común, `System.Collections.Generic`. La siguiente tabla enumera las principales:

Tabla 2.1.

Clase	Objetivo
`ICollection<T>`	Interfaz base para las colecciones genéricas.
`ReadOnlyCollection<T>`	Colección genérica de elementos de sólo lectura.
`List<T>`	Lista genérica.
`LinkedList<T>`	Lista genérica doblemente enlazada.
`Queue<T>`	Cola (lista FIFO) genérica.
`Stack<T>`	Pila (lista LIFO) genérica.
`Dictionary<K, V>`	Colección genérica de pares nombre/valor; el equivalente genérico de `Hashtable`.
`SortedDictionary<K, V>`	Lista ordenada de pares nombre/valor; el equivalente genérico de `SortedList`.

El espacio de nombres antes mencionado contiene igualmente diferentes clases y estructuras genéricas de apoyo a las mostradas en la tabla; por ejemplo, el tipo `LinkedListNode<T>` se utiliza para representar los nodos de una lista doblemente enlazada.

A continuación, se presenta precisamente un pequeño ejemplo de utilización de una lista doblemente enlazada. En él se gestionan, mediante una estructura de datos de este tipo, las ventanas hijas de una aplicación MDI, con el objetivo de navegar por ellas mediante atajos de teclado específicos.

En primer lugar, hemos redefinido el *driver* de la aplicación de la siguiente forma:

DIRECTORIO DEL CODIGO: EJEMPLO02 _ 02

```
static class Program
{
```

Continúa

```
    internal static MainForm MainFormInstance;
    internal static LinkedList<Form> listaVentanas =
       new LinkedList<Form>();

    [STAThread]
    static void Main()
    {
       Application.EnableVisualStyles();
       Application.SetCompatibleTextRenderingDefault(false);
       MainFormInstance = new MainForm();
       Application.Run(MainFormInstance);
    }
}
```

Cuando se abre una ventana hija, la ventana principal la añade a la lista doblemente enlazada:

```
public partial class MainForm : Form
{
   private static int count = 0;

   private void miNueva_Click(object sender, EventArgs e)
   {
      ChildForm f = new ChildForm();
      f.Text = "Ventana " + (++count).ToString("000");
      f.MdiParent = Program.MainFormInstance;
      f.Show();
      // añadir a lista
      Program.listaVentanas.AddLast(f);
   }

   // ...
}
```

En el evento `FormClosed` de las ventanas hijas se eliminan las ventanas de la lista:

```
public partial class ChildForm : Form
{
   private void ChildForm_FormClosed(object sender,
      FormClosedEventArgs e)
   {
      // quitar de la lista
      Program.listaVentanas.Remove(this);
   }

   // ...
}
```

Continúa

En la ventana principal se interceptan ciertas combinaciones de teclas para desplazarse a ventanas específicas:

```
private void MainForm_KeyDown(object sender, KeyEventArgs e)
{
   if (Program.listaVentanas.Count > 0)
   {
      LinkedListNode<Form> activa =
         Program.listaVentanas.Find(this.ActiveMdiChild);
      if (e.Control)
         switch (e.KeyCode)
         {
            case Keys.Right:
               Form siguiente = activa.Next != null ?
                  activa.Next.Value :
                  Program.listaVentanas.First.Value;
               siguiente.BringToFront();
               break;
            case Keys.Left:
               Form anterior = activa.Previous != null ?
                  activa.Previous.Value :
                  Program.listaVentanas.Last.Value;
               anterior.BringToFront();
               break;
            case Keys.Home:
               Program.listaVentanas.
                  First.Value.BringToFront();
               break;
            case Keys.End:
               Program.listaVentanas.
                  Last.Value.BringToFront();
               break;
         }
   }
}
```

2.5. MÁS DE UN PARÁMETRO DE TIPO

La cantidad de parámetros de tipo en una definición genérica no está, ni mucho menos, limitada a uno. En la tabla de la sección anterior, por ejemplo, hemos incluido la clase `SortedDictionary<K, V>` (la 'K' viene de *key* – clave, y la *V* de 'value' – valor), que puede utilizarse en cualquier situación en la que se necesite una estructura de datos capaz de almacenar un conjunto de pares nombre/valor ordenados. Por ejemplo, suponga que tenemos una aplicación que permite gestionar una lista de contactos, que se almacenan en un fichero XML.

El siguiente fragmento de código muestra cómo se recuperan los contactos del archivo y se almacenan en un diccionario:

DIRECTORIO DEL CODIGO: EJEMPLO02 _ 03

```
private SortedDictionary<string, Persona> dict =
   new SortedDictionary<string,Persona>();

private void FormPrincipal_Load(object sender, EventArgs e)
{
   XmlDocument doc = new XmlDocument();
   doc.Load("Agenda.XML");
   XmlNodeList lista =
      doc.SelectNodes("descendant::persona");
   foreach (XmlNode n in lista)
   {
      string nombre = n.Attributes["nombre"].Value;
      SexoPersona sexo = n.Attributes["sexo"].Value == "M" ?
         SexoPersona.Mujer :
         SexoPersona.Varón;
      dict.Add(
         nombre.ToUpper().Trim(),
         new Persona(nombre, sexo));
   }
}
```

Buscar un elemento o recorrer el diccionario (según el orden de claves) son tareas muy simples:

```
private void miBuscar_Click(object sender, EventArgs e)
{
   using (FormBusqueda f = new FormBusqueda())
      if (f.ShowDialog() == DialogResult.OK)
      {
         string nombre = f.NombreABuscar;
         if (dict.ContainsKey(nombre))
            MessageBox.Show("Sí está");
   }
}

private void miListar_Click(object sender, EventArgs e)
{
   lbxPersonas.Items.Clear();
   foreach (KeyValuePair<string, Persona> kv in dict)
      lbxPersonas.Items.Add(kv.Value);
}
```

2.6. RESTRICCIONES

Frecuentemente, al definir una clase genérica, se hace necesario imponer una o más condiciones para algunos de los parámetros de tipo asociados a la definición. Por ejemplo, asumir que un tipo hereda de una cierta clase base o implementa una cierta interfaz. Sólo así será posible aplicar a las variables de ese tipo que se utilicen dentro de la clase ciertas propiedades o métodos específicos; en el caso general, un parámetro de tipo puede ser sustituido por cualquier tipo, por lo que el compilador no puede asumir más que el hecho de que el tipo "concreto" descenderá de **`object`.** Con este fin, C# permite especificar **restricciones** sobre los parámetros de tipo de una definición genérica.

Las restricciones se especifican mediante una cláusula **`where`** asociada a la definición del tipo genérico. Por ejemplo, suponiendo que hubiéramos deseado limitar nuestra clase `ParOrdenado<T>` de modo que el tipo `T` fuera un tipo valor (lo cual invalidaría nuestra definición de `Matrimonio`), podríamos haber escrito:

```
public class ParOrdenado<T> where T: struct
{
   // ...
}
```

Las restricciones pueden ser de cinco tipos:

Tabla 2.2.

Tipo de restricción	Objetivo
De herencia	El tipo debe heredar de una clase base determinada.
De interfaz	El tipo debe implementar una interfaz determinada.
De tipo referencia	El tipo debe ser un tipo referencia (clase). Se utiliza la palabra clave **`class`** en la cláusula **`where`.**
De tipo valor	El tipo debe ser un tipo valor. Se utiliza la palabra clave **`struct`** en la cláusula **`where`.**
De constructor	El tipo debe tener un constructor sin parámetros (explícito o por defecto). Se utiliza la palabra clave **`new`** en la cláusula **`where`.**

Cada uno de los tipos de restricciones es útil en diferentes situaciones; en general, particularmente importantes son las restricciones de interfaz. Una vez que se establece una restricción de interfaz, el compilador comprobará que cualquier tipo que instancie al parámetro implemente dicha interfaz, y nos permitirá, en el cuerpo de la definición del tipo genérico, utilizar las propiedades y métodos de esa interfaz sobre los campos, variables, propiedades, etc., cuyo tipo sea el parámetro de tipo. En lo adelante encontraremos varios ejemplos.

2.7. MÉTODOS GENÉRICOS

C# permite definir no sólo tipos genéricos, sino que también puede hacerse genéricos a métodos individuales, tanto de instancia como estáticos (sin necesidad de que lo sea la clase o estructura en la que el método esté definido).

A continuación, se presenta una clase estática (otra novedad introducida en C# 2.0) a la que hemos incorporado varios métodos genéricos.

Nota:

Las clases estáticas se pueden utilizar únicamente como almacén de métodos y constantes estáticas. El ejemplo más obvio es `System.Math`, que contiene definiciones "globales" de constantes como `Math.PI` o métodos como `Math.Exp()`. Al marcar una clase como **`static`**, garantizamos, por ejemplo, que no se podrán crear objetos de esa clase o heredar de ella, entre otras restricciones.

DIRECTORIO DEL CODIGO: EJEMPLO02 _ 04

```
public static class MetodosGenericos
{
   public static void Swap<T>(ref T x1, ref T x2)
   {
      T temp = x1;
      x1 = x2;
      x2 = temp;
   }

   public static T Max<T>(T x1, T x2) where T : IComparable
   {
      if (x1.CompareTo(x2) >= 0)
         return x1;
      else
         return x2;
   }

   public static T Min<T>(T x1, T x2) where T : IComparable
   {
      if (x1.CompareTo(x2) <= 0)
         return x1;
      else
         return x2;
   }
}
```

`Swap()` es un ejemplo clásico que permite intercambiar los valores de dos variables, sean de tipo valor (en cuyo caso los datos se mueven físicamente en la memoria) o de tipo referencia (caso en el que se intercambian las referencias):

```
int x = 5, y = 7;
Console.WriteLine("ANTES {0} {1}", x, y);
MetodosGenericos.Swap<int>(ref x, ref y); // <int> es opcional
Console.WriteLine("DESPUES {0} {1}", x, y);
```

`Max()` y `Min()`, por otra parte, permiten determinar el máximo o el mínimo, respectivamente, de dos objetos de un tipo `T` que no puede ser un tipo cualquiera, sino uno que implemente la interfaz predefinida `IComparable`, cuya definición es (desgraciadamente, no se permite utilizar operadores en las interfaces):

```
public interface IComparable
{
   // Methods
   int CompareTo(object obj);
}
```

Todos los tipos básicos implementan a su manera `IComparable`. Para permitir comparar instancias de `Persona`, podríamos hacer que esta clase la implementara también:

```
public class Persona: IComparable
{
   // ...

   public int CompareTo(object otro)
   {
      if (otro == null || !(otro is Persona))
         throw new ArgumentException("No es Persona!");

      return this.Nombre.CompareTo((
         otro as Persona).Nombre);
   }
}
```

Observe la necesidad de comprobaciones y conversiones de tipo a las que nos obliga la definición de `IComparable`. Más adelante veremos cómo .NET 2.0 introduce una versión genérica de esta interfaz, `IComparable<T>`, y mejoraremos los métodos genéricos para hacer uso de esta nueva interfaz.

El ejemplo de utilización de estos métodos genéricos restringidos nos permitirá comentar sobre otra potente característica, la **inferencia de tipos** en las llamadas a métodos genéricos:

```
Persona tb1 = new Persona("Dick Tracy", SexoPersona.Varón);
Persona tb2 = new Persona("Batman", SexoPersona.Varón);
Console.WriteLine("El mayor es " +
   MetodosGenericos.Max(tb1, tb2));
// *** equivale a
Console.WriteLine("El mayor es " +
   MetodosGenericos.Max<Persona>(tb1, tb2));
```

En principio, no es necesario indicar el tipo del parámetro genérico al hacer una llamada; el compilador lo inferirá (deducirá) a partir de los tipos de los argumentos.

La librería de clases de .NET 2.0 contiene numerosos ejemplos de métodos genéricos. En muchos casos, se ha añadido una versión genérica de un método que ya existía en .NET 1.1. Por ejemplo, el método `CreateInstance()` de la clase `Activator` ahora tiene una versión genérica, que nos ahorra una buena cantidad de incomodidades al crear objetos mediante reflexión:

```
// sin genericidad
Persona tb3 = (Persona)
   Activator.CreateInstance(typeof(Persona));
// con genericidad
Persona tb4 = Activator.CreateInstance<Persona>();
tb4.Nombre = "Superman";
```

También clases como `System.Array` se han nutrido de versiones genéricas de métodos que ya estaban presentes en la versión anterior de .NET:

```
int[] arr = { 25, 1, 9, 4, 16 };

Array.Sort(arr);   // ordenar el array
foreach (int i in arr)
   Console.WriteLine(i);

if (Array.IndexOf(arr, 9) != -1)
   Console.WriteLine("El 9 está presente.");
```

2.8. DELEGADOS GENÉRICOS

Otra de las entidades que pueden ser definidas de una manera genérica desde C# 2.0 son los delegados. Por ejemplo, supongamos que quisiéramos definir un delegado

que nos permitiera llamar a cualquier método que reciba un único parámetro de un tipo `T` y no devuelva nada. La siguiente sentencia cumple con ese objetivo:

```
public delegate void Procedimiento<T>(T t);
```

A partir de la declaración anterior, se pueden crear y utilizar instancias de delegados como las siguientes:

```
static void Main(string[] args)
{
   // ...

   // Procedimiento<int> p_int =                    (C# 1.1)
   //    new Procedimiento<int>(imprimirEntero);
   Procedimiento<int> p_int2 = imprimirEntero;

   // Procedimiento<string> p_str =
   //    new Procedimiento<string>(imprimirCadena);
   Procedimiento<string> p_str2 = imprimirCadena;

   // llamadas
   p_int2(25);
   p_str2("Hola");

   Console.ReadLine();
}

static void imprimirEntero(int n)
{
   Console.WriteLine(n);
}

static void imprimirCadena(string s)
{
   Console.WriteLine(s != null ? s.ToUpper() : "NULO");
}
```

Nuevamente, es importante estar al tanto de algunas definiciones de delegados genéricos ya presentes en la librería de clases base de .NET Framework. Por ejemplo, un tipo equivalente a nuestro `Procedimiento<T>` ya está disponible, bajo el nombre `Action<T>`:

```
public delegate void Action<T>(T t);
```

Para facilitar la aplicación de una misma acción a todos los elementos de un array, la clase `System.Array` ofrece un método `ForEach<T>()` que recibe, además del

array sobre el que actuar, una instancia de delegado de tipo `Action<T>`. Eso nos permite escribir código como el siguiente:

```
Action<int> imprimir = imprimirEntero;
Array.ForEach(arr, imprimir);
// equivale a Array.ForEach<int>(arr, imprimir);
```

En realidad, si se "bucea" un poco con Reflector, es posible darse cuenta de que `Action` es algo más que un simple `MulticastDelegate`: incorpora adicionalmente varios métodos, y específicamente dos que dan soporte al patrón de llamadas asíncronas, `BeginInvoke()` y `EndInvoke()`. Por ejemplo, si se modifica ligeramente el método `imprimirEntero()` para que produzca una demora aleatoria antes de imprimir:

```
private static Random semilla = new Random();

static void imprimirEntero(int n)
{
   System.Threading.Thread.Sleep(semilla.Next(5000));
   Console.WriteLine(n);
}
```

se podrá comprobar que al ejecutar el siguiente bucle:

```
for (int i = 0; i < 5; i++)
   imprimir.BeginInvoke(arr[i], null, null);
```

los elementos del array se imprimen en un orden diferente cada vez.

Otro delegado genérico interesante es `Predicate<T>`, definido del siguiente modo:

```
public delegate bool Predicate<T>(T t);
```

`Predicate<T>` representa a un delegado que opera sobre un objeto del tipo `T` y produce **`true`** o **`false`** en dependencia de si el objeto cumple cierta condición o no. Nuevamente, `System.Array` ofrece varios métodos que permiten recorrer los elementos de un array buscando cuáles de ellos cumplen una condición llegada "desde fuera":

```
if (Array.Exists(arr, delegate (int e) { return e == 9; }))
   Console.WriteLine("El 9 está presente.");
```

Aquí hemos expresado la condición a comprobar utilizando un método anónimo, de acuerdo con lo presentado en el capítulo anterior.

2.9. INTERFACES GENÉRICAS

Para concluir este capítulo y sentar las bases para el siguiente, hablaremos sobre las **interfaces genéricas,** que aportan seguridad de tipos a sus contrapartidas no genéricas y juegan un papel muy importante en la implementación de colecciones.

Anteriormente hemos creado métodos genéricos que imponían como restricción para el parámetro la interfaz `IComparable`, y dijimos que una manera más "2.0" de lograr lo mismo hubiera sido implementando su contrapartida genérica, `IComparable<T>`. Éste sería el código:

```
public static T Max<T>(T x1, T x2) where T : IComparable<T>
{
   if (x1.CompareTo(x2) >= 0)
      return x1;
   else
      return x2;
}
```

Entonces modificaríamos la clase `Persona` de la siguiente forma:

```
public class Persona: IComparable<Persona>
{
   // ...

   // interfaz IComparable
   public int CompareTo(Persona otra)
   {
      if (otra == null)
         throw new ArgumentException("No es Persona!");

      return this.Nombre.CompareTo(otra.Nombre);
   }
}
```

Observe que el código queda mucho más claro y elegante, al no requerir comprobaciones o conversiones de tipos.

2.10. LA INTERFAZ IENUMERABLE<T>

La interfaz genérica `IEnumerable<T>` (situada en el espacio de nombres `System.Collections.Generic`) fue introducida en .NET 2.0 con el objetivo básico de jugar para los tipos genéricos el papel que cumplía en .NET 1.x su antecesora `IEnumerable`: el de ofrecer un mecanismo (eso sí, seguro en cuanto a tipos) para la iteración sobre los elementos de una secuencia, generalmente con la vista puesta en aplicar a esa secuencia el patrón de programación **`foreach`.**

En C# 3.0, la interfaz `IEnumerable<T>` cobra una importancia vital, ya que cualquier tipo de datos que la implemente puede servir directamente como origen para las expresiones de consulta LINQ. En particular, los arrays y las colecciones genéricas de .NET Framework 2.0 implementan `IEnumerable<T>`.

La definición de `IEnumerable<T>` es la siguiente:

```
public interface IEnumerable<T> : IEnumerable
{
   IEnumerator<T> GetEnumerator();
}

// en System.Collections
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

Como puede verse, la interfaz incluye un único método `GetEnumerator()`, que devuelve un **enumerador** —un objeto cuyo fin es generar elementos secuencialmente. Para hacer posible el recorrido de colecciones genéricas desde código no genérico, `IEnumerable<T>` hereda de su homóloga no genérica, `IEnumerable`, y por tanto debe implementar también una versión no-genérica de `GetEnumerator()`, para la que generalmente sirve el mismo código de la versión genérica.

Por su parte, la interfaz `IEnumerator<T>` está definida de la siguiente forma:

```
public interface IEnumerator<T> : IDisposable, IEnumerator
{
   T Current { get; }
}

// en System.Collections
public interface IEnumerator
{
   object Current { get; }
   void Reset();
   bool MoveNext();
}
```

Nuevamente, la interfaz se apoya en su contrapartida no genérica. En conjunto, `IEnumerator<T>` debe implementar los siguientes miembros:

- La propiedad `Current`, que devuelve el elemento actual de la secuencia (a dos niveles).
- El método `Reset()`, que restablece la enumeración a su valor inicial.

- El método `MoveNext()`, que desplaza el enumerador al siguiente elemento de la secuencia. Devuelve `false` cuando se llega al final de la secuencia.
- El método `Dispose()`, que libera cualesquiera recursos no administrados asociados al enumerador.

Técnicamente, un enumerador es una **máquina de estados;** toda clase que implemente `IEnumerator<T>` deberá encargarse de mantener el estado necesario para garantizar que los métodos de la interfaz funcionen correctamente.

¿Por qué esta separación en dos niveles, en la que básicamente `IEnumerable<T>` es de un nivel más alto, mientras que `IEnumerator<T>` se encarga del "trabajo sucio"? ¿Por qué no dejar que las colecciones implementen directamente `IEnumerator<T>`? La respuesta tiene que ver con la necesidad de permitir la ejecución de iteraciones anidadas sobre una misma secuencia. Si la secuencia implementara directamente la interfaz enumeradora, sólo se dispondría de un "estado de iteración" en la instancia momento y sería imposible implementar bucles anidados sobre una misma secuencia, como por ejemplo los que se encuentran en la implementación típica de la ordenación mediante el algoritmo "de la burbuja" o similares. En vez de eso, las secuencias implementan `IEnumerable<T>`, cuyo método `GetEnumerator()` debe producir un nuevo objeto de enumeración cada vez que es llamado.

2.11. LA SEMÁNTICA DE FOREACH PARA IENUMERABLE<T>

Con la explicación anterior, debe quedar más o menos clara cuál es la semántica del bucle `foreach` cuando se aplica a objetos que implementan `IEnumerable<T>`, o que simple y sencillamente ofrezcan un método público `GetEnumerator()` con la firma adecuada: aquí, los creadores de C# aplicaron un enfoque "basado en patrones", y basta con disponer de un método `GetEnumerator()` para que un objeto pueda ser utilizado como generador de elementos en un `foreach`.

El funcionamiento de `foreach` se basa en obtener un enumerador llamando a `GetEnumerator()`, para entonces recorrerlo de principio a fin utilizando los métodos `Reset()` y `MoveNext()` de éste. Un método para aplicar una misma acción sobre todos los elementos de una secuencia enumerable genérica sería:

```
public static void Iterate<T>(IEnumerable<T> secuencia,
   Action<T> metodo)
{
   foreach(T t in secuencia)
      metodo(t);
}
```

Su equivalente sin utilizar `foreach` sería:

```
public static void Iterate<T>(IEnumerable<T> secuencia,
   Action<T> metodo)
{
   {
      IEnumerator<T> e = secuencia.GetEnumerator();
      try
      {
         e.Reset();
         while (e.MoveNext())
         {
            T t = e.Current;
            metodo(t);
         }
      }
      finally
      {
         e.Dispose();
      }
   }
}
```

Imprimir un array de enteros no se diferenciaría en principio de como ya lo hemos hecho en este capítulo:

```
MetodosGenericos.Iterate(arr, imprimir);
```

Nuevamente, el compilador se encargará de deducir que en este caso `T` es **`int`.**

2.12. EJEMPLO DE IMPLEMENTACIÓN DE IENUMERABLE<T>

Aunque no es algo que sea necesario realizar con frecuencia, a continuación vamos a programar desde cero una clase que implemente `IEnumerable<T>`, lo que ayudará al lector a comprender mejor la complejidad asociada a la implementación de esta interfaz y a la vez sentará las bases para la presentación de los **iteradores** en el siguiente capítulo.

La clase que vamos a desarrollar nos permitirá iterar sobre la secuencia de los números naturales del 1 al 1000, ambos inclusive. En esta implementación "clásica" se programan, de manera explícita, todos los métodos de las interfaces

IEnumerable<**int**> e IEnumerator<**int**>. Observe la definición de la clase del enumerador como una clase anidada, enfoque utilizado con bastante frecuencia en estos casos:

```
public class NaturalNumbersSequence: IEnumerable<int>
{
   public class NaturalEnumerator: IEnumerator<int>
   {
      private int current = 1;
      private bool atStart = true;
      // interface members
      public int Current
      {
         get { return current; }
      }
      object IEnumerator.Current
      {
         get { return current; }
      }
      public void Reset()
      {
         atStart = true; current = 1;
      }
      public bool MoveNext()
      {
         if (atStart)
         {
            atStart = false; return true;
         }
         else
         {
            if (current < 1000)
            {
               current++; return true;
            }
            else
               return false;
         }
      }
      public void Dispose()
      {
         // nada
      }
   }

   public IEnumerator<int> GetEnumerator()
   {
      return new NaturalEnumerator();
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      return new NaturalEnumerator();
   }
}
```

Gracias al esfuerzo realizado para la implementación del iterador, la apariencia de un fragmento de código para iterar sobre secuencias de ese tipo será trivial:

```
foreach (int i in new NaturalNumbersSequence())
   Console.WriteLine(i);
```

2.13. LA INTERFAZ ICOLLECTION<T>

Para finalizar, dedicaremos unos párrafos a otra interfaz que pudo haber ganado en relevancia con la aparición de C# 3.0, aunque al final no fue así, como comprenderá en el Capítulo 5. Se trata de la interfaz genérica `ICollection<T>`, que implementan todas las colecciones genéricas. Ofrece la funcionalidad básica para añadir, eliminar, copiar y recorrer los elementos de una colección. Su definición es la siguiente (Reflector *dixit*):

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
   void Add(T item);
   void Clear();
   bool Contains(T item);
   void CopyTo(T[] array, int arrayIndex);
   bool Remove(T item);
   int Count { get; }
   bool IsReadOnly { get; }
}
```

Los elementos a destacar aquí son:

- Como puede verse, `ICollection<T>` extiende `IEnumerable<T>` e `IEnumerable`, lo que garantiza la posibilidad de iteración.
- Ciertos tipos de colecciones, como `Queue<T>` y `Stack<T>`, implementan `ICollection<T>` directamente.
- Otras colecciones heredan de `ICollection<T>` indirectamente, a través de las herederas de ésta:
  - `IList<T>` : colecciones cuyos elementos pueden ser accedidos mediante un índice, y
  - `IDictionary<T>` : colección de pares nombre/valor.

CAPÍTULO 3

Iteradores

En el capítulo anterior hemos estudiado las interfaces `IEnumerable` e `IEnumerable<T>`, y hemos visto cómo implementar manualmente una enumeración personalizada muy simple. En la práctica, implementar mecanismos de iteración por esa vía "tradicional" puede ser una tarea nada sencilla; piense, por ejemplo, en las complicaciones que podrían derivarse en el caso de estructuras de datos complejas, o si la colección subyacente fuera modificada durante la iteración. Los iteradores o **bloques de iteración** de C# 2.0 son una respuesta a ese problema, pues ofrecen una sintaxis clara para especificar cómo iterar sobre las colecciones, dejando al compilador el trabajo más duro: el de implementar la clase enumeradora.

3.1. BLOQUES DE ITERACIÓN

Los bloques de iteración de C# 2.0 son bloques de código que **producen** una secuencia ordenada de valores. A diferencia de los bloques de código "normales", los bloques de iteración sólo pueden ser utilizados como cuerpo de métodos (incluyendo aquí la implementación de operadores o los métodos de acceso de lectura de propiedades), y siempre que éstos devuelvan `IEnumerable<T>`, `IEnumerator<T>` o alguna de sus contrapartidas no genéricas. Los bloques de iteración no pueden contener sentencias **`return;`** en su lugar se deben utilizar sentencias **`yield`.**

Nota:

Para evitar confusiones, traduzco siempre **yield** como "producir", y **return** como "devolver".

La sentencia **`yield`** se puede presentar en dos variantes:

- **`yield return`,** que se utiliza para producir el siguiente valor de la iteración.
- **`yield break`,** que se utiliza para indicar que la iteración ha llegado a su fin.

En el caso de **`yield return`,** la sentencia debe ir acompañada de una expresión que determine el valor a producir. Si la función miembro devuelve `IEnumerable<T>`

o `IEnumerator<T>`, el tipo estático de la expresión es `T`; en caso contrario, es **`object`.** Algunos ejemplos prácticos nos aclararán estos conceptos.

3.2. LA SECUENCIA DEL 1 AL 1000, *REVISITED*

Veamos la implementación "moderna" del enumerador presentado en el capítulo anterior para recorrer los números enteros del 1 al 1000. Utilizando un iterador, la solución es mucho más concisa:

DIRECTORIO DEL CODIGO: EJEMPLO03 _ 01

```
public class NaturalNumbersSequence : IEnumerable<int>
{
   public IEnumerator<int> GetEnumerator()
   {
      for (int i = 1; i <= 1000; i++)
         yield return i;
   }

   IEnumerator IEnumerable.GetEnumerator()
   {
      for (int i = 1; i <= 1000; i++)
         yield return i;
   }
}
```

Podríamos pensar que la sentencia **`yield return`** `i;` produce el valor actual de `i` y lo entrega al contexto desde el que se le requiere; pero dejando activa toda la "maquinaria" de gestión de la iteración, de modo que en la próxima llamada se produzca el siguiente valor de la secuencia.

En el lado del cliente, el código para iterar sobre esa secuencia será idéntico al que presentamos en el caso del enumerador:

```
foreach (int i in new NaturalNumbersSequence())
   Console.WriteLine(i);
```

3.3. DETALLES INTERNOS

Como acabamos de ver, para el código cliente que los utiliza, los iteradores son muy similares a los enumeradores. ¿Qué "magia" hace esto posible? En este caso, el artífice es única y exclusivamente el compilador de C#. Lo que ocurre es que el compilador, a partir del bloque de iteración, sintetiza internamente una clase anidada (al estilo de como lo hemos hecho antes nosotros, manualmente) que implementa las interfaces `IEnumerable<T>` e `IEnumerable`, así como los correspondientes

métodos `GetEnumerator()`, que se "superponen" a los que nosotros hemos escrito, para crear instancias de esas clases enumeradoras tan pronto el código cliente lo solicite.

Nuevamente, Reflector nos muestra el código generado por el compilador:

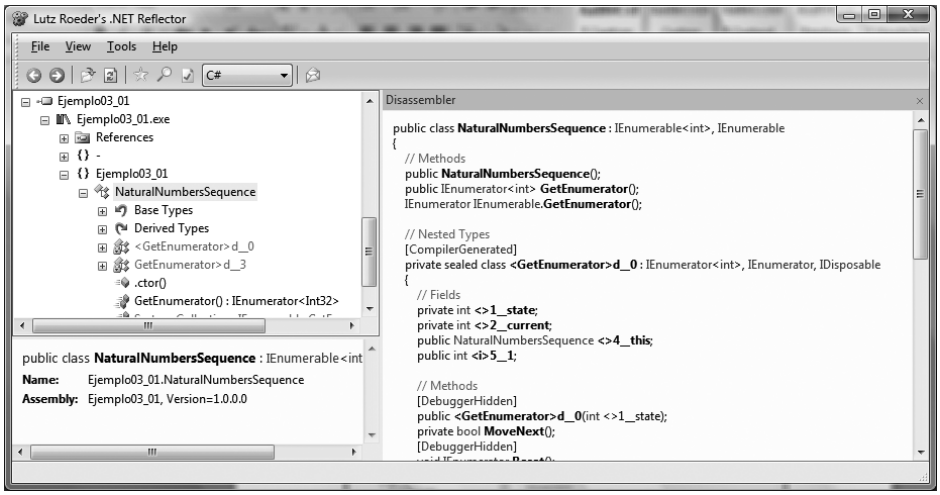


Figura 3.1.

En el momento en que vaya a comenzar una iteración sobre un objeto `NaturalNumbersSequence`, se llamará al método `GetEnumerator()` modificado, que a su vez creará el objeto enumerador, instancia de la clase generada dinámicamente. Este objeto será el encargado de ir suministrando los valores de la secuencia.

3.4. LA GENERACIÓN BAJO DEMANDA DURANTE LA ITERACIÓN

Un elemento a tener en cuenta en relación con los enumeradores e iteradores y que se utiliza ampliamente en las consultas integradas es el hecho de que, a menos que se trate de iterar sobre los elementos de un array o colección ya creados en memoria de antemano, posiblemente los diferentes elementos que componen una secuencia se irán generando en la medida en que vaya siendo necesario consumirlos, hecho que se conoce en el mundo de la programación como evaluación **bajo demanda, diferida** o **perezosa** (*lazy*). Por ejemplo, supongamos que hemos definido una clase enumerable `PrimeSequence` que produce la secuencia de los números primos:

```
public class PrimeSequence : IEnumerable<int>
{
```

Continúa

```
    private IEnumerator<int> getEnumerator()
    {
       int i = 1;
       while (true)
       {
          if (i.IsPrime())
             yield return i;

          if (i == int.MaxValue)
             yield break;
          else
          i++;
       }
    }
    public IEnumerator<int> GetEnumerator()
    {
       return getEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
       return getEnumerator();
    }
}
```

En este ejemplo, los números primos se van calculando y produciendo en la medida en que van siendo solicitados. Si no fuera por el límite natural determinado por los 32 bits que ocupa la representación de un entero (más adelante presentaremos una posible alternativa), la secuencia podría extenderse *ad infinitum*.

Como no es relevante para esta presentación de los iteradores, ignore por el momento esa sintaxis "rara" que hemos utilizado para preguntar si `i` es un número primo. ¿Es que le han añadido un método `IsPrime()` a `System.Int32`? Por supuesto que no. La respuesta está en los métodos extensores de C# 3.0, que no estudiaremos hasta el Capítulo 6.

3.5. *DON'T GO BREAKING MY ITERATION*

En el ejemplo anterior ya se utiliza la sentencia `yield break`, que es la encargada de interrumpir el proceso de enumeración, indicando que la secuencia subyacente ha llegado a su fin. Como otro ejemplo, si quisiéramos dar al cliente la posibilidad de

indicar un límite máximo para la búsqueda de números primos, podríamos mejorar la clase `PrimeSequence` de la siguiente forma:

```
public class PrimeSequence2 : IEnumerable<int>
{
   public int Limit { get; set; }

   public PrimeSequence2() : this(0) { }
   public PrimeSequence2(int limit)
   {
      Limit = limit;
   }

   private IEnumerator<int> getEnumerator()
   {
      int i = 1;
      while (true)
      {
         if (i.IsPrime())
            yield return i;

         if (Limit > 0 && i == Limit)
            yield break;
         i++;
      }
   }

   // ...
}
```

Hemos añadido a la clase una propiedad `Limit`, y añadido los constructores de modo que se pueda opcionalmente establecer el tope máximo para los enteros a recorrer. En el método privado `getEnumerator()`, si se ha establecido un límite y el valor de `i` llega a alcanzarlo, se ejecutará el **`yield break`,** que trae implícita la disposición del objeto enumerador.

3.6. MEJORANDO LA ITERACIÓN

La versión de `getEnumerator()` que se presenta a continuación mejora algo el rendimiento del anterior, gracias a que evita las llamadas al método `IsPrime()`. La presentamos principalmente para mostrar otro ejemplo de implementación directa de enumerador; lo cual no debe hacernos nunca perder la perspectiva de que, en el fondo, un iterador es un mecanismo que encapsula una **máquina de estados:** el objeto enumerador. Ésta es la principal ventaja de los iteradores como construcción del lenguaje.

```
private IEnumerator<int> getEnumerator()
{
   yield return 1;
   yield return 2;
   int i = 3;
   while (true)
   {
      bool esPrimo = true;
      if (i % 2 == 0)
         esPrimo = false;
      else
      {
         for (int n = 3; n < i / 2; n += 2)
            if (i % n == 0)
            {
               esPrimo = false;
               break;
            }
      }
      if (esPrimo)
         yield return i;

      if (Limit > 0 && i == Limit)
         yield break;
      i++;
   }
}
```

3.7. PRIMOS Y ENTEROS GRANDES

Una nueva incorporación a la librería de clases de .NET Framework es la clase `BigInteger`, de momento única residente del espacio de nombres `System.Numeric` (en el nuevo ensamblado `System.Core.dll`, al que se hace referencia automáticamente desde los proyectos creados con Visual Studio 2008). Como su nombre indica, esta clase permite operar sobre números enteros cuyos valores sobrepasen con creces los límites del tipo predefinido **`long`.** Como es natural, ofrece todo lo que en principio cabría esperar de una clase creada para tal fin: una amplia gama de constructores, operadores y métodos (éstos últimos tanto estáticos como de instancia). A continuación, se presenta una versión de la clase que itera sobre los números primos, preparada para trabajar con enteros grandes:

```
public class BigIntegerPrimes : IEnumerable<BigInteger>
{
   // props
   public BigInteger Desde { get; set; }
   public BigInteger Hasta { get; set; }
```

Continúa

```
// cons
public BigIntegerPrimes(BigInteger desde,
                        BigInteger hasta)
{
   Desde = desde;
   Hasta = hasta;
}
public BigIntegerPrimes(long desde, long hasta)
{
   Desde = new BigInteger(desde);
   Hasta = new BigInteger(hasta);
}

// iterator
private IEnumerator<BigInteger> getEnumerator()
{
   BigInteger cero = BigInteger.Zero;
   BigInteger uno = BigInteger.One;
   BigInteger dos = new BigInteger(2);
   BigInteger tres = new BigInteger(3);

   BigInteger candidato = Desde;
   while (true)
   {
      if (candidato == uno || candidato == dos)
         yield return candidato;
      else
      {
         bool esPrimo = true;
         if (candidato % 2 == cero)
            esPrimo = false;
         else
         {
            BigInteger fin = candidato / dos;
            for (BigInteger n = tres; n < fin; n += 2)
               if (candidato % n == cero)
               {
                  esPrimo = false;
                  break;
               }
         }
         if (esPrimo)
            yield return candidato;
      }
      if (candidato == Hasta)
         yield break;
      else
         candidato++;
   }
}
```

Continúa

```
   public IEnumerator<BigInteger> GetEnumerator()
   {
      return getEnumerator();
   }

   IEnumerator IEnumerable.GetEnumerator()
   {
return getEnumerator();
   }
}
```

Esta clase podrá utilizarse de la siguiente forma:

```
BigIntegerPrimes p = new BigIntegerPrimes(
new BigInteger(1),
BigInteger.Parse("10000000000000000000"));
foreach (var x in p)
   Console.WriteLine(x);
```

Eso sí, ármese de paciencia si espera ver terminar este bucle :-). Desgraciadamente, con el algoritmo anterior no podremos optar a esos premios millonarios que otorgan a quienes descubren el número primo más grande conocido hasta el momento.

3.8. UN EJEMPLO MÁS PRÁCTICO

Para finalizar este capítulo, presentaremos un ejemplo en el que la implementación práctica de un iterador puede traer consigo un beneficio sustancial. Suponga que estamos escribiendo un gestor de ficheros en disco, y en cierto momento debemos presentar en pantalla la lista de ficheros de un cierto tipo (por ejemplo, `*.jpg`) situados en un directorio (por ejemplo, `C:\Users\Octavio\Pictures\`) y todos sus subdirectorios anidados. La vía más obvia sería la siguiente:

DIRECTORIO DEL CODIGO: EJEMPLO03 _ 02

```
// enumeración de ficheros
string[] files = System.IO.Directory.GetFiles(
   " C:\\Users\\Octavio\\Pictures", "*.jpg",
   SearchOption.AllDirectories);
foreach (string s in files)
   Console.WriteLine(s);
```

Si ejecuta estas instrucciones, y sobre todo si su directorio de imágenes contiene mucha información, notará una demora sustancial entre el momento en que se lanza la llamada a `GetFiles()` y el momento en que aparece el primer fichero

en pantalla. La razón estriba en que este método "recolecta" los nombres de todos los ficheros que encuentra en un array de cadenas, y sólo cuando ha terminado ese recorrido es que devuelve el control y se puede comenzar a imprimir. Esto puede dar al traste con la experiencia de usuario de una aplicación; generalmente es preferible ir mostrando al usuario información con mayor frecuencia, incluso a costa de que el proceso como un todo se extienda un poco más.

Este fue el razonamiento de Carl Daniel, un colega norteamericano que desarrolló la clase `FileSystemEnumerator` que se describe a continuación ("File system enumeration using lazy matching", publicado originalmente en **http://www.codeproject.com/csharp/FileSystemEnumerator.asp**). Conociendo que la API de Windows ofrece los métodos de bajo nivel `FindFirstFile()` y `FindNextFile()`, que devuelven la información sobre un fichero tan pronto lo encuentran, Carl las encapsuló mediante llamadas `PInvoke` y, basándose en ellas, creó una clase que ofrece un enumerador capaz de ir produciendo según demanda la información sobre los ficheros que satisfacen las condiciones especificadas. Presentamos esta clase sólo a grandes rasgos; contiene muchos detalles de bajo nivel que no vienen a cuento aquí:

```
public sealed class FileSystemEnumerator : IDisposable
{
   // constructor
   public FileSystemEnumerator(
      string pathsToSearch,
      string fileTypesToMatch,
      bool includeSubDirs)
   {
      // ...
   }

   // devuelve un iterador capaz de producir la lista
   // de ficheros que cumplen las condiciones indicadas
   public IEnumerable<FileInfo> Matches()
   {
      // ...
   }
}
```

Teniendo a mano esta clase, se puede redefinir la búsqueda anterior de la siguiente forma:

```
FileSystemEnumerator fse =
      new FileSystemEnumerator("C:\\Users\\Octavio\\Pictures",
         "*.jpg", true);
   foreach (FileInfo fi in fse.Matches())
      Console.WriteLine(fi.FullName);
```

Invito al lector a que compruebe la diferencia en el funcionamiento de ambas variantes.

CAPÍTULO 4

Tipos valor anulables

Otro de los aportes de la genericidad en los lenguajes para la plataforma .NET fue la aparición de los **tipos valor anulables.** Los tipos valor anulables hacen posible aplicar la semántica del valor nulo típica de los tipos referencia también a los tipos valor. Tradicionalmente, los lenguajes de propósito general no han ofrecido ningún soporte en este aspecto, y se ha hecho necesario recurrir a trucos más o menos "sucios" para indicar un valor nulo o desconocido para los tipos valor: utilizar un valor especial del dominio (por ejemplo, `-1` ó `0` para los enteros), o mantener indicadores lógicos en variables booleanas independientes; técnicas todas que adolecen de inconvenientes importantes. Los tipos valor anulables (o simplemente tipos anulables, para abreviar), resuelven definitivamente este problema tan antiguo, sentando las bases para la utilización del valor **`null`** con todos los tipos de la plataforma, algo especialmente importante a la hora de interactuar con bases de datos.

4.1. FUNDAMENTOS

Los tipos valor anulables se construyen en C# añadiendo el modificador de tipo **`?`** al nombre del tipo valor "base". Por ejemplo, **`int?`** es la variante del tipo predefinido **`int`** que admite el valor nulo. El lenguaje provee de conversiones implícitas del literal **`null`** y del tipo valor correspondiente al tipo valor anulable. Esto hace perfectamente válidas las siguientes sentencias:

DIRECTORIO DEL CODIGO: EJEMPLO04 _ 01

```
int? n = null;
int? m = 275;
```

donde se declaran dos variables enteras anulables, `n` y `m`, con valores iniciales **`null`** y `275`, respectivamente. Entre los miembros de que disponen todos los tipos valor anulables están la propiedad lógica `HasValue`, que indica si la variable tiene un valor asignado o no, y `Value`, del tipo subyacente (en este caso **`int`**), que devuelve el valor asignado a la variable:

```
static void imprimir(int? x)
{
     if (x.HasValue)
        Console.WriteLine(x.Value);
     else
        Console.WriteLine("??");
}
```

¿Qué podemos hacer con los tipos valor anulables? Pues básicamente todo lo que se puede hacer sobre sus tipos valor subyacentes. Para permitir esto, el lenguaje ofrece principalmente las **conversiones entre tipos anulables,** las **conversiones promovidas** (*lifted conversions*), los **operadores promovidos** (*lifted operators*) y el nuevo **operador de combinación ??**

- La regla para las conversiones entre tipos anulables establece que para cada conversión predefinida entre dos tipos valor "normales" `T1` y `T2`, existe automáticamente también una conversión de `T1?` a `T2?` Si el valor de origen es distinto de **`null`,** se realizará la conversión "tradicional" para obtener el valor de destino no nulo; si el valor de origen es **`null`,** entonces el valor de destino también será nulo (regla conocida como **propagación del valor nulo**). También están disponibles conversiones de `T1` a `T2?` y de `T1?` a `T2`; en este último caso, la conversión deberá ser explícita, y provocará una excepción si el valor de origen es **`null`.**

```
int i = 27;
int? j = i;                 // int -> int?, Ok
double? x = i;              // int -> double -> double?, Ok
double? y = j;              // int? -> double?, Ok
int? k = y;                 // ERROR DE COMPILACIÓN
int? l = (int?)y;           // double? -> int?, OK
int m = (int)y;             // double? -> int? -> int
                            //    excepción si y es null
```

- Para cada conversión definida por el usuario entre dos tipos valor no anulables `T1` y `T2`, el compilador añade una nueva **conversión promovida** entre `T1?` y `T2?`, que también funciona mediante la propagación del valor nulo.
- Para cada operador que no sea de comparación en el que tanto los operandos como el resultado son de tipo valor, el lenguaje añade un **operador promovido** en el que a los tipos de los operandos y el resultado se les añade el modificador `?` Por ejemplo, el operador promovido correspondiente al operador + que recibe dos enteros y devuelve otro entero, es un operador que recibe dos **`int?`** y devuelve un **`int?`** Los operadores promovidos propagan los nulos: si uno de los operandos es **`null`,** el resultado es **`null`.** Si ninguno de los operandos es nulo, entonces se aplica la semántica del operador original para calcular el resultado.

```
int? a = 27;
int? b = 50;
int? c = null;
int? d = a + b;          // d = 77
int? e = a + c;          // e = null
```

- En el caso de los operadores de comparación, los operadores promovidos producen **bool,** y no **bool?** Aquí es donde la semántica de tratamiento de nulos de C# se distancia de la de SQL.

```
bool b1 = a < b;         // true
bool b2 = a < c;         // false (algún operando es null)
bool b3 = a == c;        // false (uno de los dos es null)
bool b4 = c == null;     // true
```

- Por último, C# 2.0 introdujo un único nuevo operador, el **operador de combinación** (*coalescing operator*), que se denota mediante **??** Trabaja de manera similar a la función COALESCE de SQL Server: si `a` no es nulo, el resultado de `a` **??** `b` es `a`; en caso contrario, el resultado es `b`.

```
int? f = a ?? b;         // f = 27
int? g = c ?? b;         // g = 50
```

4.2. IMPLEMENTACIÓN DE LOS TIPOS VALOR ANULABLES

Los tipos valor anulables se implementan a partir de una estructura genérica incorporada en la librería de .NET Framework, `System.Nullable<T>`. La notación basada en el signo de interrogación es mero azúcar sintáctico, y una declaración como **int?** se traduce internamente por `System.Nullable<`**int**`>`.

`Nullable<T>` se define aproximadamente de la siguiente forma:

```
public struct Nullable<T> where T : struct
{
   // propiedad HasValue
   private bool hasValue;

   public bool HasValue
   {
      get { return hasValue; }
   }

   //propiedad Value
   private T value;
```

Continúa

```
    public T Value
    {
       get
       {
          if (!hasValue)
             throw new InvalidOperationException();
          return value;
       }
    }

    // constructor
    public Nullable(T valor)
    {
       this.value = valor;
       this.hasValue = true;
    }

    // otros métodos...
}
```

Todos los detalles de la implementación de este tipo pueden obtenerse, por supuesto, mediante Reflector. Y esta excelente herramienta también nos permitirá examinar cualquier tipo que incorpore algún campo o método de un tipo anulable, como por ejemplo nuestra clase `Persona`:

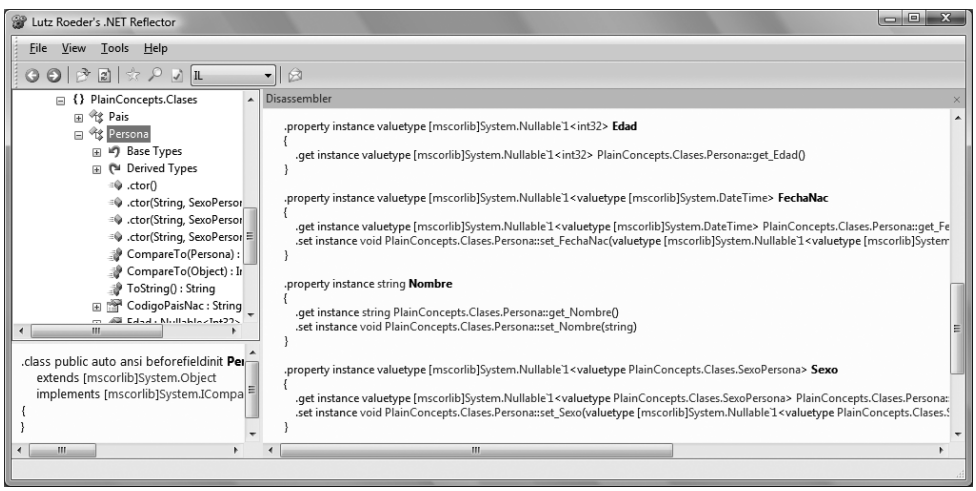


Figura 4.1.

Observe cómo, de acuerdo con lo visto en el Capítulo 2, la clase incluye varias especializaciones de `System.Nullable<T>`: una basada en el tipo **`int`** (para la propiedad `Edad`), otra basada en `DateTime` (para la propiedad `FechaNac`), y aún otra más, basada en la enumeración `SexoPersona` (para la propiedad `Sexo`).

4.3. UN DETALLE A TENER EN CUENTA

Un detalle relacionado con la implementación de los tipos anulables que conviene conocer es el comportamiento de éstos en caso de aplicación de llamadas a métodos virtuales definidos en **`object`**, como `ToString()`, y en caso de conversiones hacia y desde **`object`**, situaciones en las que normalmente tienen lugar los procesos de *boxing* y *unboxing*.

¿Qué efecto esperaría que produjese la ejecución del siguiente fragmento?:

```
int? w = null;
Console.WriteLine(w.ToString());   // imprime la cadena vacía
```

Aunque podría pensarse que el código anterior debería provocar una excepción de referencia nula, esto no ocurre así. La respuesta está en que los tipos valor anulables heredan de `ValueType`, y el método virtual `ToString()` ha sido redefinido en esta clase de modo que se imprima una cadena vacía cuando se trate de un tipo anulable con valor nulo. Por otra parte, si se asigna `w` a una variable de tipo **`object`**, la variable tomará directamente el valor **`null`** (o sea, que un tipo anulable con valor nulo no se "boxea"). Una llamada a `ToString()` sobre esta variable entonces provocará una excepción:

```
object o = w;
Console.WriteLine(o.ToString());   // produce excepción
```

Novedades en C# 3.0

CAPÍTULO 5

Novedades "básicas" en C# 3.0

Este capítulo describe las nuevas características más sencillas (o más fáciles de explicar :-) que se han añadido a C# en la versión 3.0. Estas nuevas características, conjuntamente con las que se presentan en los capítulos 6, 7 y 8, se integran de manera armoniosa con el resto de las posibilidades que ya ofrecía el lenguaje (especialmente las incorporadas en la versión 2.0, como los tipos genéricos, los iteradores o los delegados anónimos), y en conjunto sientan las bases para la implementación de la más importante novedad que incluye C# 3.0: las **expresiones de consulta** (*query expressions*), el principal reflejo de la tecnología LINQ en el lenguaje de programación, que trataremos a partir del Capítulo 9.

5.1. DECLARACIÓN IMPLÍCITA DEL TIPO DE VARIABLES LOCALES

En C# 3.0 es posible dejar al compilador la tarea de determinar el tipo de una variable local explícitamente inicializada, en base al tipo de la expresión de inicialización. Por ejemplo, en el siguiente fragmento de código el compilador infiere que la variable `n` es de tipo **`int`** y que `s` es de tipo **`string`** por el tipo de las constantes que se intenta asignar a cada una:

DIRECTORIO DEL CODIGO: EJEMPLO05 _ 01

```
var n = 2;          // equivale a int n = 2;
var s = "Hola";     // equivale a string s = "Hola";
```

Por supuesto, se mantienen en vigor las reglas del lenguaje para determinar el tipo de las expresiones:

```
var m = 3 * n;     // m es de tipo int
var dosPi = 2 * Math.PI;
// dosPi es de tipo double
```

El mecanismo también funciona para otros tipos cualesquiera:

```
var p1 = new Persona("Ana", SexoPersona.Mujer);
// p1 es de tipo Persona
var p2 = p1;
// p2 también
var listaEnteros = new List<int>();
// listaEnteros es de tipo List<int>
var elMesQViene = DateTime.Now.AddMonths(1);
// elMesQViene es de tipo DateTime
```

Los siguientes son ejemplos de usos incorrectos de **`var`:**

```
var  p;             // la inicialización es obligatoria
var  q = null;      // imposible inferir el tipo de q
var  z1 = 3,
     z2 = 5;        // no se admiten declaraciones múltiples
var  r = { 1, 2, 3 };
     // un inicializador de array no es permitido
     // (se verá más adelante)
```

Hay que destacar que esta característica no tiene nada que ver con la que ofrecen algunos lenguajes dinámicos, en los que una variable puede almacenar valores de distintos tipos a lo largo de su tiempo de vida. A partir de C# 3.0, el compilador simplemente nos libera de la necesidad de especificar el tipo de una variable inicializada e **infiere** (¡de nuevo la inferencia de tipos!) el tipo de la variable declarada mediante el identificador "especial" **`var`** a partir del tipo de la expresión que se le asigna. La verificación estricta de tipos, uno de los pilares del lenguaje, sigue por supuesto en vigor, y la mejor manera de comprobarlo es, nuevamente, utilizando una herramienta como Reflector, que nos dejará ver claramente que esas variables tienen un tipo concreto asignado por el compilador.

La declaración implícita del tipo de una variable local puede utilizarse también en otras construcciones del lenguaje, como el bucle **`foreach`** o la sentencia **`using`:**

```
foreach (var k in listaEnteros)    // k es de tipo int
   Console.WriteLine(k);

using (var con = new SqlConnection(cadenaConexion))
   // con es de tipo SqlConnection
{
   con.Open();
   // interactuar con la base de datos
}
```

5.1.1. Sobre la conveniencia de utilizar var

Mucho se ha escrito ya desde antes de la aparición de C# 3.0 sobre el posible efecto nocivo que podría conllevar el abuso de esta nueva característica del lenguaje. En efecto, su uso indiscriminado podría hacer el código de nuestras aplicaciones menos claro, requiriendo esfuerzos adicionales por parte del lector para determinar el tipo de una variable.

Aunque a priori este nuevo recurso podría utilizarse para declarar la mayoría de las variables locales inicializadas (exceptuando, por ejemplo, aquellas sobre las que queramos utilizar el polimorfismo, o las que queramos inicializar con el valor "genérico" **`null`**), mi recomendación personal es utilizarla única y exclusivamente en las siguientes situaciones:

- En casos en los que el tipo de la variable es evidente; por ejemplo, por el hecho de que la expresión que se va a asignar a la variable es una llamada a un constructor, en el que queda reflejado claramente el tipo:

```
var p1 = new Persona("Ana", SexoPersona.Mujer);
```

- En los casos en que la utilización de **`var`** sea simplemente obligatorio (porque no penséis que se trata de una característica añadida al lenguaje "gratuitamente" :-). La verdadera necesidad de las declaraciones con tipo implícito solo se hace evidente cuando se hace uso de otras de las nuevas características del lenguaje, como los tipos anónimos (que veremos en este capítulo) o las consultas integradas (que veremos en el Capítulo 9).

Una última observación, que es aplicable a todas las demás palabras clave incorporadas al lenguaje a partir de la versión 3.0: **`var`** es una palabra clave **contextual;** esto es, el compilador determina si su aparición debe tratarse como palabra clave o como un identificador normal en función del contexto en el que la encuentra. Esto hace posible que las aplicaciones ya existentes que utilizaran `var` como nombre de variable se puedan seguir compilando en C# 3.0 sin problemas.

5.2. PROPIEDADES IMPLEMENTADAS AUTOMÁTICAMENTE

Frecuentemente, al implementar clases definimos propiedades triviales cuyo único sentido es encapsular un campo privado:

```
public class Libro
{
   private string nombre;

   public string Nombre
   {
      get { return nombre; }
```

Continúa

```
      set { nombre = value; }
   }

   private int añoPublicación;
   public int AñoPublicación
   {
      get { return añoPublicación; }
      set { añoPublicación = value; }
   }

   // ...
}
```

Aún cuando el *snippet* **prop** de Visual Studio nos simplifica en gran medida la tarea de teclear ese código, son líneas de código "mecánico" que nos podríamos ahorrar. Las **propiedades implementadas automáticamente** automatizan ese patrón. En C# 3.0 se podrá escribir simplemente:

```
public string Nombre { get; set; }
```

El campo privado interno será generado automáticamente por el compilador; como ventaja adicional, su nombre (que será algo tan horrible como `<>k__AutomaticallyGeneratedPropertyField0`, según Reflector) no estará a nuestra disposición en el código fuente de la clase.

No se permite definir propiedades de sólo lectura o sólo escritura de esta manera. Pero sí es posible establecer diferentes niveles de visibilidad para los métodos de acceso:

```
// propiedad "de solo lectura"
public int AñoPublicación { get; private set; }
```

5.3. INICIALIZADORES DE OBJETOS Y COLECCIONES

C# 3.0 añade nuevas construcciones sintácticas para facilitar la especificación *inline* de constantes de tipos compuestos (estructuras y clases) y de colecciones de éstos. Estas construcciones constituyen un útil mecanismo de conveniencia, y al igual que ocurre con la inferencia de tipos de las variables locales, toda su potencia sólo se hace visible cuando se utilizan en combinación con otras de las novedades más avanzadas de C# 3.0 que se describen en lo adelante en este libro.

5.3.1. Inicializadores de objetos

C# 3.0 incorpora la figura del inicializador de objetos, que ofrece una manera más clara y concisa de especificar un objeto inicializado. Por ejemplo, en el fragmento

de código que se muestra a continuación se declara una variable de tipo `Persona`, que se inicializa con un objeto de la clase construido ad hoc. Básicamente, en lugar de la tradicional lista de argumentos del constructor ahora se puede utilizar también esta variante más "declarativa", en la que los valores de las propiedades o campos del objeto se especifican mediante un **inicializador de objeto,** consistente en una secuencia de elementos del tipo `nombre = valor` encerrada entre llaves.

La declaración:

```
var p = new Persona
   {
      Nombre = "Amanda",
      Sexo = SexoPersona.Mujer,
      FechaNac = new DateTime(1998, 10, 23)
   };
```

equivale a:

```
Persona p = new Persona(
   "Amanda",
   SexoPersona.Mujer,
   new DateTime(1998, 10, 23));
```

o en última instancia a:

```
Persona p = new Persona();
p.Nombre = "Amanda";
p.Sexo = SexoPersona.Mujer;
p.FechaNac = new DateTime(1998, 10, 23);
```

Esta última versión es la que sintetiza el compilador al generar el código objeto. De ello se desprende que para que este mecanismo funcione es requisito indispensable que el tipo de la variable a inicializar ofrezca un constructor sin argumentos (como ocurre en el caso de las estructuras), de las clases que no tienen un constructor definido explícitamente, así como de aquellas que son sintetizadas artificialmente por el compilador ("tipos anónimos", que presentaremos a continuación).

Los inicializadores de campos pueden anidarse; por ejemplo, podríamos inicializar una variable del tipo `Matrimonio` definido en el Capítulo 2 (eso sí, después de agregarle un constructor sin parámetros) de la siguiente forma:

```
var p = new Persona
   {
      Nombre = "Amanda",
      Sexo = SexoPersona.Mujer,
      FechaNac = new DateTime(1998, 10, 23)
   };
```

Continúa

```
var m1 = new Matrimonio
   {
      Marido = new Persona
         {
            Nombre = "Octavio",
            Sexo = SexoPersona.Varón
         },
      Mujer = new Persona
         {
            Nombre = "Irina",
            Sexo = SexoPersona.Mujer
         },
   };
```

Por supuesto, los valores a asignar a las propiedades que se van a inicializar pueden ser expresiones cualesquiera, que involucren a variables que estén en ámbito en el punto de la inicialización.

5.3.2. Inicializadores de colecciones

La extensión de la sintaxis para la inicialización de objetos se ha ampliado también a las colecciones genéricas, para lo que se ha añadido el concepto de **inicializador de colección:** una secuencia de inicializadores de elementos encerradas entre llaves.

El tipo al que se aplica el inicializador de colección debe implementar la interfaz `IEnumerable<T>` y disponer de uno o más métodos `Add()` públicos. La inicialización se traduce en la construcción del objeto colección, seguida de una secuencia de llamadas a alguno de sus métodos `Add()`, pasando cada vez uno de los "inicializadores de elementos", que en realidad son "argumentos para el método `Add()`".

Analicemos bajo esta luz diferentes ejemplos. La siguiente sentencia:

```
var listaEnteros2 = new List<int> { 1, 2, 3 };
```

se traduce en:

```
var listaEnteros2 = new List<int>();
listaEnteros2.Add(1);
listaEnteros2.Add(2);
listaEnteros2.Add(3);
```

Este ejemplo es más de lo mismo, pero la colección es de objetos `Persona`:

```
var hijos = new List<Persona> {
   new Persona { Nombre = "Diana", Sexo = SexoPersona.Mujer,
                 FechaNac = new DateTime(1996, 2, 4) },
   new Persona { Nombre = "Dennis", Sexo = SexoPersona.Varón,
                 FechaNac = new DateTime(1983, 12, 27) },
   p
};
```

Y en este ejemplo más complejo se construye e inicializa un diccionario con un conjunto de palabras y sus respectivos plurales:

```
var plurales = new Dictionary<string, string> {
   { "nombre", "nombres" },
   { "arroz", "arroces" },
};
```

El código que genera el compilador, como puede comprobarse con Reflector, es el siguiente:

```
var plurales2 = new Dictionary<string, string>();
plurales2.Add("nombre", "nombres");
plurales2.Add("arroz", "arroces");
```

Es interesante señalar que éste es otro aspecto del lenguaje en el que se ha introducido un enfoque "basado en patrones" (algo que ya hemos mencionado con relación al bucle **`foreach`**) por razones puramente prácticas. El diseño original de C# 3.0 establecía que las colecciones debían implementar la interfaz `ICollection<T>`, lo que es mucho más consecuente con la naturaleza *strongly typed* de C#, pero los creadores del lenguaje se dieron cuenta de que muchas de las clases de colección presentes actualmente en la librería de clases no implementan esta interfaz y decidieron rebajar la "barrera de entrada".

5.4. TIPOS ANÓNIMOS INMUTABLES

Los **tipos anónimos inmutables** (o, para abreviar, tipos anónimos) son una nueva característica de C# 3.0 que juega un papel muy importante en las consultas integradas en el lenguaje, y que tiene que ver con la posibilidad de que el compilador sintetice de manera automática nuevos tipos de datos a partir de las características que se le indiquen en expresiones de inicialización.

La siguiente sentencia muestra un ejemplo de un tipo anónimo:

```
var revista1 = new
   {
      Nombre = "dotNetManía",
      EjemplaresPorAño = 11
   };
```

El adjetivo "anónimos" no significa que son tipos que no tienen nombre; simplemente que esos nombres son generados internamente y no están a nuestra disposición para utilizarlos en el código fuente. Al compilar la sentencia anterior, el compilador generará internamente una clase (si se analiza el ensamblado con Reflector, se verá que el nombre asignado a la clase es algo así como `<>f__AnonymousType0`) dotada de las dos propiedades que se indican en su expresión de inicialización, `Nombre`

y `EjemplaresPorAño`. Y cuando se ejecute la aplicación, la variable `revista1` apuntará a un objeto de ese tipo generado de forma automática por el compilador.

El adjetivo "inmutables", por otra parte, tiene relación con el hecho de que las instancias de esas clases (porque son clases) no se podrán modificar a partir del momento en que son creadas. Por ejemplo, la siguiente sentencia fallará:

```
revista1.EjemplaresPorAño = 4; // error de compilación
```

La decisión de hacer inmutables a los tipos anónimos fue tomada relativamente poco antes de la salida de Visual Studio 2008, y tiene que ver con la conveniencia de evitar efectos colaterales en el tipo de programación funcional y concurrente en el que se espera que se utilicen principalmente los tipos anónimos. Para garantizar esa invariabilidad, los tipos anónimos generados por el compilador disponen únicamente de los siguientes miembros (como se puede ver con Reflector):

- Un constructor con tantos parámetros como propiedades se declaren en la inicialización, mecanismo que se utilizará para crear e inicializar las instancias de la clase.
- Las propiedades en sí son de sólo lectura.

En nuestro caso, la clase generada internamente por el compilador será más o menos como la siguiente:

```
public class Revista
{
   // interno
   private string nombre;
   private int ejemplares;

   // propiedades (solo lectura)
   public string Nombre { get { return nombre; } }
   public int EjemplaresPorAño { get { return ejemplares; } }
   // constructor
   public Revista(String nombre, int ejemplares)
   {
      this.nombre = nombre;
      this.ejemplares = ejemplares;
   }
}
```

Un detalle de primordial importancia es el siguiente: supongamos que en una nueva sentencia definimos ahora la siguiente variable:

```
var revista2 = new
   {
      Nombre = "MSDN Magazine",
      EjemplaresPorAño = 12
   };
```

Un examen interno del ensamblado nos convencerá de que las variables `revista1` y `revista2` son **de un mismo tipo anónimo** y no de dos tipos diferentes, lo que habilita, por ejemplo, la asignación o la comparación entre ellas. C# "combina" las definiciones de tipos anónimos siempre que en las expresiones de inicialización se incluyan las mismas propiedades, los tipos de éstas coincidan y estén declaradas en el mismo orden.

Los métodos `Equals()` y `GetHashCode()` de los tipos anónimos se definen en términos de los métodos `Equals()` y `GetHashCode()` de las propiedades, por lo que dos instancias de un mismo tipo anónimo son iguales si y solo si todas sus propiedades tienen el mismo valor.

5.5. ARRAYS DE TIPO DEFINIDO IMPLÍCITAMENTE

Otra novedad de C# 3.0 relacionada con las anteriores es la posibilidad de inicializar un array sin indicar explícitamente el tipo de los elementos, que es inferido por el compilador a partir de los tipos de las constantes suministradas. El siguiente fragmento muestra varios ejemplos de arrays de tipos valor, tipos referencia e incluso de tipos generados automáticamente; en este último caso, se aprovecha la posibilidad de combinación de tipos anónimos que hemos mencionado anteriormente.

```
var a = new[] { 2, 3, 5, 7 }; // int []
                              // new[] es obligatorio ¡!
var b = new[] { "x", "y1", "z" }; // string []
var contactos = new[] {
   new  {
          Nombre = "Antonio López",
          Telefonos = new[] { "914792573", "607409115" }
        },
   new  {
          Nombre = "Pedro Posada",
          Telefonos = new[] { "981234118" }
        }
};
```

5.6. MÉTODOS PARCIALES

Una nueva característica incorporada a última hora a C# 3.0 son los **métodos parciales.** Se trata de una nueva "vuelta de tuerca" a la exitosa incorporación de las **clases parciales** en C# 2.0. Como es sabido, las clases parciales hicieron posible separar la implementación de una clase en dos o más ficheros de código fuente que se combinan en tiempo de compilación, algo muy útil en el caso en que parte de una clase es generada automáticamente por una herramienta. El propio Visual Studio 2008 es el principal consumidor de esta tecnología.

Los métodos parciales son una característica dirigida a proporcionar una suerte de gestión de eventos "ligera" dentro del marco de las clases parciales. Considere, por ejemplo, la siguiente clase:

DIRECTORIO DEL CODIGO: EJEMPLO05 _ 02

```
public partial class ClaseParcial
{
   // declaraciones
   static partial void MetodoParcial1(int n);
   static partial void MetodoParcial2();

   private string nombre;
   partial void OnCambioNombre(string viejo, string nuevo);
   public string Nombre
   {
      get
      {
         return nombre;
      }
      set
      {
         OnCambioNombre(nombre, value);
         nombre = value;
      }
   }
}
```

En la clase parcial anterior se declaran dos métodos parciales estáticos, `MetodoParcial1()` y `MetodoParcial2()`. Se trata meramente de declaraciones; no se suministran implementaciones (cuerpos) de los métodos correspondientes. Adicionalmente, se declara una propiedad `Nombre` que en su método de acceso de escritura llama a otro método parcial declarado (¡sólo declarado!) en este fichero fuente.

Por otra parte, podríamos tener otra parte de la clase parcial, en la que se implementa solamente `MetodoParcial2()`, pero no `MetodoParcial1()`:

```
public partial class ClaseParcial
{
   // definiciones
   static partial void MetodoParcial2()
   {
      Console.WriteLine("Llamada a MetodoParcial2");
   }

   private static int Calculo(int n)
   {
```

Continúa

```
      int x = 0;
      // cálculo que tarda cierto tiempo...
      return x;
   }

   public static void MetodoPublico(int n)
   {
      MetodoParcial1(Calculo(n));
      MetodoParcial2();
   }

   private static Stopwatch mon = new Stopwatch();

   partial void OnCambioNombre(string viejo, string nuevo)
   {
      lock (mon)
      {
         Console.WriteLine("Antes:    " + viejo);
         Console.WriteLine("Después: " + nuevo);
      }
   }
}
```

Finalmente, suponga que desde el cuerpo de `Main()` se producen llamadas a esos métodos parciales:

```
class Program
{
   static void Main(string[] args)
   {
      ClaseParcial p = new ClaseParcial();
      p.Nombre = "SHERLOCK";

      ClaseParcial.MetodoPublico(4);

      Console.ReadLine();
   }
}
```

El resultado de la compilación de la aplicación anterior será el siguiente:

- El compilador no encontrará una implementación del método `MetodoParcial1()`, por lo cual eliminará el método de la clase `ClaseParcial` (incluso de los metadatos), como si éste jamás hubiera existido.
- El compilador eliminará la llamada a `MetodoParcial1()` situada en el cuerpo de `MetodoPublico()`.

- Más aún, el compilador eliminará todo el código IL relacionado con la evaluación de los argumentos de esa llamada; en este caso, no se generará código para la costosa llamada a la función `Calculo()`.
- Sí se generará el código correspondiente a la llamada al método `MetodoParcial2()`, para el que sí hay un cuerpo definido y que sí estará presente en el código objeto de la clase `ClaseParcial`.
- La implementación del método `OnCambioNombre()` será llamada como resultado de la asignación de valor a la propiedad de `p` en `Main()`.

Algunos elementos a destacar sobre los métodos parciales son:

- Los métodos parciales se permiten única y exclusivamente en las clases parciales.
- Los métodos parciales se identifican por la presencia del modificador **`partial`.**
- Los métodos parciales son privados por definición; no es necesario indicarlo explícitamente.
- El tipo de retorno de los métodos privados es siempre **`void`.**
- Los métodos parciales pueden tener cualquier cantidad de parámetros; éstos pueden ser **`ref`,** pero no **`out`.**
- Los métodos parciales pueden ser estáticos; además, pueden ser métodos extensores si forman parte de una clase estática.

5.6.1. Utilidad de los métodos parciales

Los métodos parciales han sido pensados para situaciones en la que cierta clase generada automáticamente por una herramienta debe dejar "ganchos" que permitan a un desarrollador "conectarse" a ellos para personalizar el comportamiento de la clase.

Un ejemplo típico en este sentido es el del método `OnCambioNombre()` del ejemplo anterior. En este caso, el usuario podrá suministrar una implementación para ese método parcial para "enchufarse" a los eventos de cambio de nombre y reaccionar adecuadamente. En este escenario, los métodos parciales son simplemente un mecanismo de conveniencia, más cómodo y ligero que la alternativa utilizada hasta el presente, basada en la definición de un delegado o evento.

CAPÍTULO 6

Métodos extensores

Una de las nuevas características de C# 3.0 que ha causado más revuelo entre la comunidad de programadores son los **métodos extensores** (*extension methods*). Mientras que algunos los ven como "el no va más" por la flexibilidad que proporcionan, otros vaticinan que harán totalmente caótica la programación. En realidad, no es ni lo uno ni lo otro; como todo en esta vida, los métodos extensores tienen sus pros y sus contras, que es conveniente conocer para hacer un uso racional de ellos. Lo que sí parece innegable es que juegan un papel esencial en la implementación de la arquitectura abierta de las consultas integradas, por lo que han llegado indudablemente para quedarse.

Este capítulo muestra qué son los métodos extensores, cómo se utilizan y sus ventajas e inconvenientes.

6.1. INTRODUCCIÓN

Los métodos extensores son un mecanismo de conveniencia que permiten extender o ampliar la funcionalidad de una clase sin recurrir a la herencia, técnica en ocasiones imposible (porque la clase en cuestión ha sido declarada **`sealed`**) o inadecuada desde el punto de vista de diseño, ni a la delegación o composición, que requiere una implementación más laboriosa y frecuentemente no es mucho más adecuada en lo que a diseño se refiere. Para hacer justicia y mostrar que no todo se cuece en Microsoft, debemos señalar que la idea que subyace a los métodos de extensión ha sido tomada de Delphi .NET, donde existe desde hace tres versiones una característica similar, conocida allí como clases auxiliares (*helper classes*).

Suponga que tenemos a nuestra disposición la clase `Persona` de la Introducción, y que queremos ampliarla con métodos adicionales, por ejemplo, añadirle un método lógico que permita conocer si la persona cumple años en lo que resta del mes en curso (si su cumpleaños ya ha pasado, ¿de qué vale felicitarlo?). Suponga, además, que por alguna de las razones antes mencionadas la herencia no es una alternativa viable. Usando los nuevos recursos que nos brinda C# 3.0, la solución consistiría extender `Persona` artificialmente por medio de una clase auxiliar como la clase `Persona _ Helper` que se muestra a continuación, dotada del método extensor correspondiente.

DIRECTORIO DEL CODIGO: EJEMPLO06 _ 01

```
namespace PlainConcepts.Extensions
{
   public static partial class Persona_Helper
   {
      public static bool CumpleAñosEsteMes(this Persona p)
      {
         Console.WriteLine("Uso del método extensor");
         return p.FechaNac.HasValue &&
            p.FechaNac.Value.Month == DateTime.Today.Month &&
            p.FechaNac.Value.Day >= DateTime.Today.Day;
      }
   }
}
```

Hemos incluido en el código la declaración del espacio de nombres porque, como veremos más adelante, los espacios de nombres juegan un papel importante aquí. Observe además que tanto la clase como el método extensor son estáticos (realmente una cosa implica la otra), y que el primer parámetro de `CumpleAñosEsteMes()` es de tipo `Persona`, y va precedido del modificador **`this`**. ¿Cómo utilizaríamos este nuevo método en nuestro código? Inicialmente hay que importar el espacio de nombres que contiene la clase:

```
using PlainConcepts.Extensions;
```

Una vez hecho esto, ya podemos tratar al método `CumpleAñosEnMes()` **como un método más** de la clase `Persona`:

```
var p = new Persona
   {
      Nombre = "Amanda",
      Sexo = SexoPersona.Mujer,
      FechaNac = new DateTime(1998, 10, 23)
   };
if (p.CumpleAñosEsteMes())
   Console.WriteLine(p.Nombre + " cumple años el día " +
      p.FechaNac.Value.Day);
```

Note cómo la ayuda *Intellisense* de Visual Studio nos informa de qué métodos extensores de la clase están disponibles:

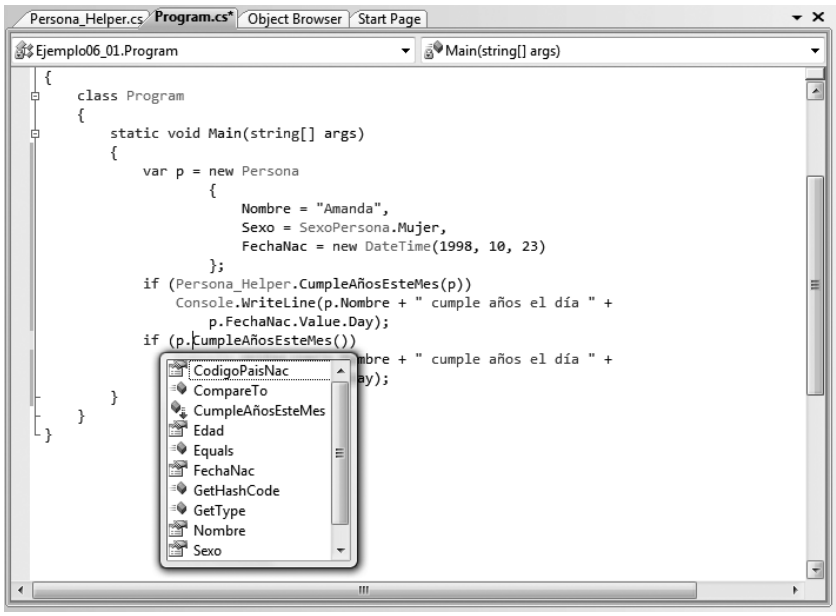


Figura 6.1.

Observe también que los métodos extensores se identifican por un símbolo especial que indica su condición.

6.2. LA SINTAXIS

Son varias las condiciones que impone el lenguaje a los métodos extensores:

- La clase en la que se alojan debe ser una clase estática (las clases estáticas fueron introducidas en C# 2.0 y las hemos mencionado en el Capítulo 2). Más aún, esta clase no deberá ser genérica (aunque los métodos extensores sí podrán serlo) ni estar anidada dentro de otra.
- Ellos mismos deben ser métodos estáticos.
- Si un método extensor tiene como objetivo extender la clase `X`, su declaración deberá tener un primer parámetro del tipo `X`. Más aún, este primer parámetro deberá ir precedido del modificador **`this`,** cuya presencia es la que indica al compilador que está en presencia de un método extensor.
- Aparte de esas restricciones, los métodos extensores pueden hacer todo lo que está al alcance de un método estático tradicional.

Satisfechas las condiciones anteriores, y una vez "en órbita" (perdón, en ámbito), la clase que contiene el método extensor, éste podrá ser aplicado a instancias de la clase extendida como si de un método de instancia común y corriente se tratara. Una de las principales ventajas, entonces, de los métodos extensores, es que hacen más "amistosa" su utilización para el cliente que los usa. Si no existieran los

métodos extensores y quisiéramos lograr algo parecido a lo que hemos hecho con `CumpleAñosEsteMes()`, habríamos creado el método exactamente de la misma manera (sólo que sin el modificador **`this`**), para utilizarlo así:

```
if (Persona_Helper.CumpleAñosEsteMes(p))
   Console.WriteLine(p.Nombre + " cumple años el día " +
      p.FechaNac.Value.Day);
```

Valga decir que los métodos extensores, como métodos estáticos que son, pueden utilizarse también de esta manera. Pero se estaría perdiendo uno de sus atractivos principales: obviamente, la sintaxis de objetos en más natural para el programador OO. Observe, por otra parte, que en la última variante queda reflejado explícitamente en el código cliente **de dónde proviene** el método `CumpleAñosEsteMes()`, algo que queda "camuflado" cuando se utiliza la sintaxis de método de instancia. Más adelante discutiremos detenidamente sobre las ventajas e inconvenientes de los métodos extensores.

6.3. MÁS EJEMPLOS

El siguiente listado presenta unos cuantos ejemplos más de métodos extensores, empaquetados dentro de una clase utilitaria:

```
namespace PlainConcepts.Extensions
{
   public static class Helpers
   {
      // extensiones para int
      public static bool esPrimo(this int num)
      {
         if (num == 1 || num == 2)
            return true;
         if (num % 2 == 0)
            return false;
         for (int i = 3; i < num / 2; i += 2)
            if (num % i == 0)
               return false;
         return true;
      }

   public static bool esMultiploDe(this int num, int otro)
   {
      return num % otro == 0;
   }

   public static bool esImpar(this int num)
```

Continúa

```
{
   return num.esMultiploDe(2);
}

// extensiones para double
public static double ElevadoA(this double a, double b)
{
   // calcula a^b
   return Math.Pow(a, b);
}

// extensiones para string
public static bool EsDireccionCorreo(this string s)
{
   Regex regex = new Regex(
      @"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$");
   return regex.IsMatch(s);
}

public static bool EsNulaOVacia(this string s)
{
   return (s == null || s.Trim().Length == 0);
}

public static string Inversa(this string s)
{
   char[] rev = s.ToCharArray();
   Array.Reverse(rev);
   return (new string(rev));
}

// extensiones para FileInfo
public static bool EsOculto(FileInfo fi)
{
   return (fi.Attributes & FileAttributes.Hidden) != 0;
}

// extensiones para object
public static void Imprimir(this object obj)
{
   Console.WriteLine(obj);
}
public static void Imprimir(this object obj, string fmt)
{
   Console.WriteLine(fmt, obj);
}

// extensiones para T[]
public static T[] Corte<T>(this T[] arr,
   int index, int count)
```

Continúa

```
    {
       if (index < 0 || count < 0 ||
          arr.Length - index < count)
          throw new ArgumentException("Parámetro inválido");
       T[] result = new T[count];
       Array.Copy(arr, index, result, 0, count);
       return result;
       }

       // extensiones para IEnumerable<T>
       public static IEnumerable<T> Duplicar<T>(
          this IEnumerable<T> secuencia)
       {
          if (secuencia == null)
             throw new ArgumentException("Secuencia nula");
          foreach (T elemento in secuencia)
          {
             yield return elemento;
             yield return elemento;
          }
       }
    }
}
```

Aquí hay varias observaciones que hacer:

- Mediante los métodos extensores podemos extender cualquier tipo, tanto propios como de terceros (Microsoft incluido :-).
- Los métodos extensores pueden tener parámetros "adicionales", como muestra, por ejemplo, `esMultiploDe()`.
- Los métodos extensores se pueden sobrecargar, como muestra el método `Imprimir()`. Al definir este método como extensor de **`object`**, podremos utilizarlo para mostrar en la consola prácticamente cualquier cosa que tengamos a mano.
- Al extender una clase (como `System.IO.FileInfo`), dentro del método extensor podemos hacer uso de todas las propiedades, métodos y eventos visibles del objeto-parámetro. Por supuesto, no podemos violar la encapsulación y acceder a los detalles internos de esa clase.

Los dos últimos casos son ejemplos de métodos extensores genéricos, en el primer caso para "seccionar" un array unidimensional cualquiera, creando otro array con un subconjunto de los elementos del array original. El segundo método extiende la interfaz `IEnumerable<T>`, dotándola de un nuevo método llamado `Duplicar()` que produce dos veces cada elemento de la secuencia original. ¿Lo ve? Si entiende cómo funciona este método (probablemente le convendrá repasar nuevamente el Capítulo 3), tendrá adelantada una buena parte del camino a la hora de enfrentarse a las consultas integradas en el lenguaje (Capítulo 9).

Teniendo en ámbito la clase anterior, podremos utilizar sus métodos extensores así:

```
int n = 48;
if (n.esMultiploDe(4))
   n.Imprimir("{0} se divide entre 4");

string dir = "ghost@plainconcepts.com";
if (dir.EsDireccionCorreo())
   p.Imprimir(dir.Inversa());

var a1 = new[] { -1, 0, 1, 2, 3 };

var a2 = a1.Corte(1, 3); // a2 = { 0, 1, 2 }
foreach (var x in a2)
   x.Imprimir();

// imprime { 0, 0, 1, 1, 2, 2 }
foreach (var x in a2.Duplicar())
   x.Imprimir();
```

6.4. EL ACCESO A LOS MÉTODOS EXTENSORES

Como ya hemos mencionado, los métodos extensores están disponibles en un espacio de nombres si forman parte de una clase estática de dicho espacio o de una clase estática importada mediante la directiva **`using`.** De hecho, en C# 3.0 una directiva **`using`,** además de importar todos los tipos definidos en el espacio de nombres indicado, importa también todos los métodos extensores situados en todas las clases estáticas de ese espacio.

Del mismo modo, en C# 3.0 han sido extendidas las reglas de resolución de llamadas a métodos para tener en cuenta la posible existencia de métodos extensores. En este sentido, debe tenerse en cuenta que los métodos extensores tienen siempre **la menor prioridad** a la hora de la resolución de llamadas, y son utilizados por el compilador únicamente en el caso de que no exista ningún método de instancia adecuado en la clase extendida o sus clases base. Por ejemplo, si añadiésemos a la clase `Persona` un método lógico `CumpleAñosEsteMes()`, éste sería el método llamado después de recompilar la aplicación:

```
public class Persona
{
   // ...

   public bool CumpleAñosEsteMes()
   {
      Console.WriteLine("Uso del método de la clase");
      return FechaNac.HasValue &&
         FechaNac.Value.Month == DateTime.Today.Month &&
         FechaNac.Value.Day >= DateTime.Today.Day;
   }
}
```

En el caso de que el compilador encuentre dos métodos extensores aplicables en un momento dado, producirá un error de ambigüedad de nombres. Por ejemplo, si al proyecto se añadiera otra clase extensora:

```
namespace Pokrsoft.Extensions
{
   public static class Helpers2
   {
      public static void Imprimir(this object obj)
      {
         MessageBox.Show(obj.ToString());
      }

      // más...
   }
}
```

y se importara también el espacio de nombres `Pokrsoft.Extensions` desde el módulo que contiene a `Main()`, la aplicación dejaría de compilar (llamada ambigua).

Si la definición de este método `Imprimir()` hubiera sido, por ejemplo:

```
public static void Imprimir(this int obj) // para enteros
{
   MessageBox.Show(obj.ToString());
}
```

entonces no habría habido colisión alguna y se utilizaría esta versión de `Imprimir()` para mostrar cualquier valor entero; pero esto ya no tiene nada que ver con los métodos extensores, son las reglas comunes y corrientes de resolución de sobrecargas las que entran en acción.

Pero lo que es importante comprender aquí es que si eliminamos la sentencia **`using`** `PlainConcepts.Extensions`, dejando visible únicamente el espacio `Pokrsoft.Extensions`, la aplicación compilará y funcionará perfectamente, con la diferencia de que ahora los mensajes al usuario irán a un cuadro de mensaje en lugar de a la consola. Volveremos a hacer mención de esta posibilidad de reemplazar un comportamiento por otro al final del capítulo.

6.5. RECOMENDACIONES DE UTILIZACIÓN

El ejemplo anterior debe haberle provocado la reflexión en la que se basan generalmente los detractores de los métodos extensores (yo mismo fui uno de ellos, Pablo Reyes me hizo cambiar de opinión). En sistemas complejos, en los que se utilicen decenas de librerías de terceros o incluso desarrolladas por diferentes equipos de

la misma casa, la incorporación a un proyecto de una nueva librería que contenga métodos extensores podría introducir un "ruido" en el sistema que hiciera que el código actual dejara de compilar. Algo así como la reaparición del "infierno de las DLL", sólo que a otro nivel.

Se trata de un peligro cierto, y la respuesta pasa en este caso por la educación de los desarrolladores. Siendo fiel a un conjunto mínimo de reglas de utilización de los métodos extensores, tales riesgos se reducen prácticamente a cero. El siguiente es mi conjunto de sugerencias con relación a los métodos extensores:

- La regla primera y fundamental: no utilice los métodos extensores como sustitutos para los métodos de instancia ni para evitar la herencia. La propia especificación de C# 3.0 indica que los métodos extensores sólo deben utilizarse en aquellos casos en los que no sea posible o conveniente utilizar un método de instancia. En los ejemplos que se han ofrecido al principio del capítulo, habría sido preferible añadir un método `CumpleAñosEnMes()` a la clase `Persona` (si dispusiéramos del código fuente) o crear una clase heredera e incorporarle dicho método (a menos que la clase `Persona` fuera sellada u otra razón poderosa no recomendara heredar de ella).
- En general, una librería de métodos extensores (como cualquier librería), podría desarrollarse para uso interno o para ofrecer a terceros. Mi opinión es que, en general, no es recomendable crear librerías de métodos extensores para ofrecer a terceros, por la probabilidad de colisiones como las anteriormente descritas. Excepción a esta regla: los proveedores LINQ (Capítulo 11).
- Sitúe siempre todos sus métodos extensores en un espacio de nombres específico y de nivel interno (por ejemplo, `PlainConcepts.Extensions`).
- Si está desarrollando una librería y en su implementación ha utilizado métodos extensores que no quiere poner a disposición de sus clientes, asegúrese de que las clases que contienen los métodos extensores no sean públicas.
- Es muy cómodo crear clases extensoras para implementar métodos utilitarios que operen sobre los tipos básicos, como algunos de los ejemplos presentados antes. En tal caso, asegúrese de cumplir las reglas anteriores.
- Sí, es sumamente conveniente a nivel de empresa o equipo de programación coordinar el desarrollo de una librería interna de métodos extensores, para no estar todos "reinventando la rueda" cada vez.

6.6. LA RAZÓN DE SER DE LOS MÉTODOS EXTENSORES

Aunque podría argumentarse que los métodos extensores ofrecen ya una ventaja a sus clientes, la ilusión OO a través de la sintaxis, la verdadera razón por la que se han incorporado al lenguaje es la necesidad que tuvieron los diseñadores de LINQ de implementar una arquitectura abierta. Los miembros del equipo que diseñó C# 3.0 querían ofrecer una versión predeterminada de un conjunto de métodos

aplicables a cierto grupo de clases, manteniendo a la vez abierta la posibilidad de que los creadores de cualquiera de esas clases:

- Definan, si lo desean, sus propias versiones de ese conjunto de métodos dentro de la propia clase, y que estos métodos propios tengan mayor prioridad que los predeterminados (como hemos mostrado en el caso de `Persona.CumpleAñosEnMes()`).
- Definan un nuevo conjunto de métodos que sustituyan completamente a los predeterminados, de modo que simplemente sacando unos de ámbito y poniendo a otros (como hemos hecho con `Helpers.Imprimir()`) el comportamiento sea otro diferente.

Los métodos extensores ofrecen precisamente esas posibilidades.

CAPÍTULO 7

Expresiones lambda

No desvelamos ningún secreto al decir que C#, que en su versión inicial se presentaba como una herramienta lingüística destinada en principio a subsumir los paradigmas de programación procedimental (imperativa), orientada a objetos (en línea con los estándares impuestos en su momento por **C++** y **Java**) y basada en componentes (con dos referentes claros, **Delphi** y **Visual Basic**), ha ido evolucionando a partir de ese momento por un camino que intenta tener en cuenta e incorporar al lenguaje las mejores ideas surgidas al calor de otros puntos de vista de la programación como los lenguajes dinámicos (**Perl, Python, Ruby**), los lenguajes de acceso a almacenes de datos (**SQL**, **XML**) o la programación funcional (**LISP**, **ML**).

Las expresiones lambda, una de las novedades más importantes de C# 3.0, son precisamente un recurso proveniente del mundo de la programación funcional.

7.1. INTRODUCCIÓN

En LISP y sus dialectos, como **Scheme,** la construcción **lambda** es un mecanismo para definir funciones. Por ejemplo, la expresión:

```
(define cuadrado (lambda (x) (* x x)))
```

define una variable `cuadrado` cuyo valor es una función que calcula el cuadrado de su argumento. Una de las características más importantes de estos lenguajes, a la que otorgan una enorme importancia los arquitectos de .NET Framework (busque, por ejemplo, el artículo "Scheme is love", de Don Box, en MSDN Magazine), es que las funciones lambda pueden ser tratadas no sólo como piezas de código que pueden ser invocadas, sino también como valores "corrientes y molientes" que pueden ser manipulados por otros bloques de código. Utilizando un término que escuché por primera vez hace más de 20 años de mi entonces tutor, el Dr. Luciano García, las funciones lambda facilitan la **metaprogramación:** la escritura de programas capaces de manipular otros programas.

Este capítulo describe la implementación en C# 3.0 de las **expresiones lambda,** centrándose principalmente en la explotación de éstas como bloques de código; el tema de su representación como datos y posterior tratamiento programático solamente

se introduce en aras de la completitud, quedando en buena medida pendiente para el siguiente capítulo, dedicado a los árboles de expresiones.

7.2. LAS EXPRESIONES LAMBDA COMO OBJETOS DE CÓDIGO

La manera natural de implementar una definición como la de la función lambda anterior en C# sería a través de un tipo y una instancia de delegados:

DIRECTORIO DEL CODIGO: EJEMPLO07 _ 01

```
delegate int Mapeado_I_I(int x);
// ...
Mapeado_I_I cuadrado_I = delegate(int x) { return x * x; };
Console.WriteLine(cuadrado_I(8)); // imprime 64
```

En realidad, esta definición no es exactamente equivalente a la de LISP; nuestro delegado `cuadrado` está limitado a operar sobre números enteros. Un tipo genérico nos ayudará a ampliar un poco ese horizonte:

```
delegate T Mapeado<T>(T x);
// ...
Mapeado<int> cuadrado = delegate(int x) { return x * x; };
```

La primera línea del código anterior define un tipo delegado genérico llamado `Mapeado`. A partir de este modelo podremos crear instancias de delegados para funciones que a partir de un valor de un tipo `T` producen otro valor del mismo tipo `T` (o sea, que **mapean** —recuerde el término, que comenzaremos a utilizar con frecuencia— un valor del tipo `T` a otro valor del mismo tipo).

En la segunda línea, por otra parte, se define una instancia del tipo delegado anterior que "apunta" a una función que devuelve el cuadrado de un número entero. En esta sintaxis se hace uso de un método anónimo, característica incorporada a C# en la versión 2.0 que hemos presentado en el Capítulo 1.

Aunque los métodos anónimos ofrecen una notación más compacta y directa, su sintaxis es aún bastante verbosa y de naturaleza imperativa. En particular, al definir un método anónimo es necesario:

- Utilizar explícitamente la palabra reservada **`delegate`.**
- Indicar explícitamente los tipos de los parámetros, sin ninguna posibilidad de que el compilador los deduzca a partir del contexto de utilización.
- En el caso bastante frecuente en que el cuerpo de la función es una simple sentencia **`return`** seguida de una expresión que evalúa el resultado (como en el caso de nuestro `cuadrado`), se hace aún más evidente el exceso sintáctico.

Las **expresiones lambda** pueden considerarse como una extensión de los métodos anónimos, que ofrecen una sintaxis más concisa y funcional para expresarlos. Por

ejemplo, nuestra definición de cuadrado quedaría de la siguiente forma utilizando una expresión lambda:

```
Mapeado<int> cuadrado1 = x => x * x;
```

¿Más corto imposible, verdad? Se expresa de una manera muy sucinta y natural que `cuadrado1` hace referencia a una función que, a partir de un `x`, produce `x` multiplicado por sí mismo.

7.3. LA SINTAXIS

La sintaxis de una expresión lambda consiste en una lista de variables-parámetros, seguida del símbolo de implicación (aplicación de función) =>, que es seguido a su vez de la expresión o bloque de sentencias que implementa la funcionalidad deseada. En el caso de `cuadrado1`, el compilador deducirá (¡más inferencia de tipos!) automáticamente del contexto que el tipo del parámetro y del resultado deben ser **`int`.** Pudimos haberlo indicado explícitamente:

```
Mapeado<int> cuadrado2 = (int x) => x * x;
```

Si se especifica el tipo del parámetro, o la expresión tiene más de un parámetro (se indiquen explícitamente o no sus tipos) los paréntesis son obligatorios. Por otra parte, en la parte derecha de la implicación se puede colocar una expresión, como hemos hecho hasta el momento, o un bloque de sentencias de cualquier complejidad (siempre que, por supuesto, produzca como resultado un valor del tipo adecuado):

```
Mapeado<int> cuadrado3 = (int x) => { return x * x; };
```

Para completar el panorama, también pudimos haber usado la variante en la que el tipo del parámetro se omite y el cuerpo es un bloque:

```
Mapeado<int> cuadrado4 = x => { return x * x; };
```

Así clasifica la documentación oficial de C# 3.0 las formas sintácticas de las expresiones lambda: según un sistema de coordenadas bidimensional en el cual, por una parte, los parámetros pueden ser de tipo implícito o explícito, y por otra, el cuerpo puede ser una expresión o un bloque de sentencias. Por cierto, en el caso de una expresión lambda sin parámetros de entrada, los paréntesis son obligatorios:

```
delegate void Accion();
// ...
Accion saludo = () => Console.WriteLine("Hola");
```

7.4. EL TIPO DELEGADO FUNC

Aunque para una mejor comprensión inicial hemos definido nuestro propio tipo delegado genérico `Mapeado<T>`, en la práctica la mayor parte de las veces utilizaremos las distintas sobrecargas del tipo genérico predefinido `Func`, que .NET 3.5 pone a nuestra disposición en el ensamblado `System.Core.dll`:

```
// en System.Core.dll
// espacio de nombres System.Linq
public delegate T Func<T>();
public delegate T Func<A0, T>(A0 a0);
public delegate T Func<A0, A1, T>(A0 a0, A1 a1);
public delegate T Func<A0, A1, A2, T>(A0 a0, A1 a1, A2 a2);
public delegate T Func<A0, A1, A2, A3, T>(A0 a0, A1 a1,
   A2 a2, A3 a3);
```

Las diferentes variantes del tipo `Func` representan a los delegados a métodos (con 0, 1, 2 ó 3 argumentos, respectivamente) que devuelven un valor de tipo `T`. Utilizando este tipo, podríamos definir una nueva versión de `cuadrado` así:

```
Func<int, int> cuadrado6 = x => x * x;
```

7.5. MÁS EJEMPLOS

A continuación, se presentan unos cuantos ejemplos más de definiciones de expresiones lambda:

```
Func<int, int, bool> esDivisiblePor =
   (int a, int b) => a % b == 0;

Func<int, bool> esImpar = x => esDivisiblePor(x, 2);

Func<int, bool> esPrimo =
   x =>
   {
      for (int i = 2; i <= x / 2; i++)
         if (esDivisiblePor(x, i))
            return false;
      return true;
   };

Func<double, double, double> hipotenusa =
   (x, y) => Math.Sqrt(x * x + y * y);
```

Continúa

```
Func<double, double, double> elevadoA =
   (x, y) => Math.Pow(x, y);

Func<string, bool> esNulaOVacia =
   s => s == null || s.Trim().Length == 0;

Func<string, string> inversa = s =>
   {
      char[] rev = s.ToCharArray();
      Array.Reverse(rev);
      return (new string(rev));
   };
```

Algunos ejemplos de uso de las funciones anteriores:

```
if (esPrimo(97))
   Console.WriteLine("97 es primo");
Console.WriteLine(inversa("ABRACADABRA"));
```

7.6. UNO MÁS COMPLEJO

Aquí va un "rompecabezas" para que se vaya familiarizando con lo que podrá encontrar por ahí. Analice la siguiente declaración (y no siga leyendo hasta que crea que la entiende :-):

```
Func<int, Func<double, double>> selector =
   x =>
   {
      switch (x)
      {
         case 1:
            return Math.Sqrt;
         case 2:
            return Math.Sin;
         default:
            return delegate(double z) { return 2 * z; };
      }
   };
```

Se trata de un delegado que apunta a una función que recibe un entero y devuelve ¡otro delegado!, esta vez uno que apunta a un método que recibe un **`double`** y devuelve otro **`double`.** Hablando "a bajo nivel", cuando `selector` es llamado con `1` como parámetro devuelve una referencia al método `Math.Sqrt()`; si el valor de entrada es `2`, devuelve una referencia a `Math.Sin()`; en cualquier otro caso, se

devuelve una referencia a una función que duplica el valor que recibe. Este delegado podría usarse así:

```
int sel = 1;
double entrada = 4.5;
double salida = selector(sel)(entrada);
Console.WriteLine(salida);
```

Con esos valores de entrada, el valor de salida será la raíz cuadrada de 4,5.

7.7. PARÁMETROS POR REFERENCIA

La programación funcional (al menos en su variante más tradicional) es un paradigma de programación libre de efectos colaterales (nada de parámetros por referencia o asignación a variables externas a la función). Pero lo que ofrece C# 3.0 no es programación funcional pura (así que esas cosas se pueden hacer).

Ya en el Capítulo 1, dedicado a los métodos anónimos, vimos cómo éstos pueden acceder al contexto que les rodea y los malabares que hace el compilador de C# para hacer esto posible. Con las funciones lambda ocurre exactamente lo mismo, así que no volveremos a repetirlo aquí.

Sí mostraremos un ejemplo de una función lambda que maneja parámetros **`ref`** (podrían ser **`out`** también). Para complicarlo más, viene mezclado con genericidad:

```
delegate T M_TT_T<T>(ref T a, ref T b)
   where T: IComparable<T>;
// ...
M_TT_T<int> maxAndSwap =
   (ref int a, ref int b) =>
   {
      if (a.CompareTo(b) < 0)
      {
         int temp = a;
         a = b;
         b = temp;
      }
      return a;
   };
```

El tipo `M_TT_T` representa a delegados a funciones que reciben (por referencia) dos parámetros de tipo `T` y devuelven otro `T`. Además, `T` debe implementar `IComparable`. El delegado `maxAndSwap` es de ese tipo (con **`int`** en el rol de `T`) y devuelve el menor de dos enteros, intercambiándolos además de modo que el primer parámetro reciba el valor mayor. Por supuesto, para escribir una expresión lambda con parámetros por referencia es necesario utilizar tipado explícito en la lista de parámetros.

En el siguiente fragmento de código se hace una llamada a esa función:

```
int m = 36, n = 45;
int max = maxAndSwap(ref m, ref n);
Console.WriteLine("m = {0}, n = {1}", m, n);
```

7.8. RECURSIVIDAD EN EXPRESIONES LAMBDA

Las expresiones lambda permiten definir funciones de una manera muy concisa, pero a priori tienen una limitación: no permiten expresar (al menos directamente) funciones recursivas. Tomemos como ejemplo la definición del factorial:

```
int factorial(int n)
{
     return n <= 1 ? 1 : n * factorial(n - 1);
}
```

El compilador no permitirá utilizar el nombre del delegado en la propia definición de la función porque ésta aún no ha terminado:

```
Mapeado_I_I ff = (k) => k = 1 ? 1 : k * ff(k - 1); // MAL!
```

A continuación, se presenta una posible solución, basada en la utilización de funciones de orden superior. Primero, se define un tipo delegado para una función capaz de pasarse a sí misma como parámetro:

```
delegate int AutoFuncion(AutoFuncion self, int x);
```

Luego, en base al prototipo anterior se define un delegado intermedio que implementa el patrón del factorial, pero sobre una función auto-aplicable:

```
AutoFuncion F = (f, n) => n == 0 ? 1 : n * f(f, n - 1);
```

Ahora ya estamos listos para definir la función lambda `factorial`:

```
Func<int, int> factorial = x => F(F, x);
Console.WriteLine(factorial(5));   // imprime 120
```

Como puede ver, el asunto tiene su complejidad. La moraleja a extraer de esta sección es la siguiente: si necesita definir funciones recursivas, acuda primero a los recursos "tradicionales" de C#.

7.9. LAS EXPRESIONES LAMBDA COMO OBJETOS DE DATOS

Con lo visto hasta el momento, cualquiera podría pensar que las expresiones lambda son un mero recurso de tipo "azúcar sintáctico" para "endulzar" el consumo de delegados y métodos anónimos. Pero las expresiones lambda de C# 3.0 ofrecen una posibilidad adicional, no disponible para los métodos anónimos, y que juega un papel importante en la implementación de la tecnología LINQ. Como apuntamos al inicio del capítulo, esta posibilidad adicional es la de compilarse como **árboles de expresiones,** objetos de datos que representan en memoria al algoritmo de evaluación de una expresión lambda. Estos árboles en memoria pueden ser luego fácilmente manipulados mediante software (¡metaprogramación!), almacenados, transmitidos, etc.

Precisamente a los árboles de expresiones dedicaremos el próximo capítulo.

CAPÍTULO 8

Árboles de expresiones

En el capítulo anterior hemos presentado las expresiones lambda de C# 3.0 y mencionado cómo éstas tienen dos facetas diferentes de representación y utilización estrechamente relacionadas entre sí: como código (en forma de métodos anónimos, bloques de código IL directamente ejecutables) y como datos (en forma de árboles de expresiones, estructuras de datos capaces de representar de una manera eficiente el algoritmo de evaluación de la expresión). En aquella ocasión nos centramos en la primera faceta; esta vez nos concentraremos en los árboles de expresiones.

8.1. DE EXPRESIONES LAMBDA A ÁRBOLES DE EXPRESIONES

Para introducir el tema del capítulo, comenzaremos presentando el pequeño ejemplo a continuación:

DIRECTORIO DEL CODIGO: EJEMPLO08 _ 01

```
static Expression<Func<double, double>>
   cuboExpr = x => x * x * x;

static void Main(string[] args)
{
   Console.WriteLine(cuboExpr);
   // imprime 'x => ((x * x) * x)

   // aquí se genera IL !
   var cubo = cuboExpr.Compile();

   Console.WriteLine("Cubo(5) = " + cubo(5));
   // imprime 'Cubo(5) = 25'
   Console.ReadLine();
}
```

El fragmento de código anterior comienza con una asignación en la que una expresión lambda para calcular el cubo de un número entero se asigna no a un delegado, sino a una variable del tipo genérico predefinido `Expression`:

```
// en System.Core.dll
// espacio de nombres System.Linq.Expressions
public class Expression<T> : LambdaExpression
{
   // ...
}
```

Como resultado de la asignación antes mencionada, no se genera el código IL correspondiente a la expresión, sino un **árbol de objetos en memoria** que representa a la secuencia de acciones necesarias para su evaluación. De la estructura del árbol generado podemos hacernos una idea observando la salida que produce la llamada (provocada por `Console.WriteLine()`) a su método `ToString()`.

Observe también que la clase `Expression` ofrece un método `Compile()`, mediante el cual se hace posible generar dinámicamente el código ejecutable correspondiente a la expresión en el momento deseado. O sea, que este método sirve como "puente" entre los dos mundos, traduciendo los datos en código en el momento en que desee evaluar la expresión.

8.1.1. Los árboles como representación de expresiones

Como hemos podido ver, las expresiones lambda se compilan como código o como datos en dependencia del contexto en que se utilizan. Si se asigna una expresión lambda (como ejemplo utilizaremos ahora una que implementa la conocida fórmula para calcular el área de un círculo) a una variable de un tipo delegado:

```
Func<double, double> area =
   (radio) => Math.PI * radio * radio;
```

el compilador generará en línea el código IL correspondiente dentro de un método y hará que el delegado "apunte" a ese método, de modo que la definición anterior es equivalente a la asignación del siguiente método anónimo:

```
Func<double, double> area =
   delegate(double radio)
   {
      return Math.PI * radio * radio;
   };
```

Pero, como también hemos ya comentado, una cosa muy diferente ocurre si asignamos la expresión lambda a una variable del tipo `Expression<T>`; en ese caso,

el compilador no traducirá la expresión a código ejecutable, sino que generará una estructura de datos en memoria que representa a la expresión en sí. Estas estructuras de datos conforman lo que en C# 3.0 se conoce como **árboles de expresiones.** Continuando con el ejemplo, si utilizamos la expresión lambda que calcula el área de un círculo de la siguiente forma:

```
Expression<Func<double, double>> areaExpr =
   (radio) => Math.PI * radio * radio;
```

lo que estamos expresando equivale a la siguiente secuencia de asignaciones:

```
ParameterExpression parm =
   Expression.Parameter(typeof(double), "radio");
Expression<Func<double, double>> areaExpr =
   Expression.Lambda<Func<double, double>>(
      Expression.Multiply(
         Expression.Constant(Math.PI),
         Expression.Multiply(
            parm,
            parm)),
      parm);
```

Mediante la primera de las dos sentencias anteriores se crea un objeto de la clase `ParameterExpression`. `ParameterExpression` es la clase mediante la que se representan los parámetros de una función en un árbol de expresiones. Nuestra expresión lambda tiene un solo parámetro, el radio del círculo, así que creamos un objeto que lo representa. En la segunda sentencia, por otra parte, es donde realmente se construye el árbol de expresión correspondiente a la expresión lambda, utilizando el objeto-parámetro obtenido en el paso anterior. En ella se utilizan objetos de otras clases de la jerarquía de clases de expresiones que presentaremos a continuación, como `ConstantExpression` (para representar constantes; la llamada al método estático `Expression.Constant()` crea un objeto de esta clase) o `BinaryExpression` (que representa a una expresión basada en un operador binario; en este caso, se construye un objeto de esta clase como resultado de la llamada a `Expression.Multiply()`).

Observe el estilo de programación funcional que se utiliza normalmente al definir árboles de expresiones mediante código: la expresión se compone de una cadena de llamadas anidadas que van construyendo cada uno de los elementos de la expresión y se van "ensartando" unas con otras. Este estilo de programación se hará muy presente también en una de las variantes específicas de LINQ - LINQ to XML, la extensión para el trabajo con documentos XML (Capítulo 12).

El siguiente gráfico le ayudará a comprender la estructura interna en memoria del objeto-expresión:

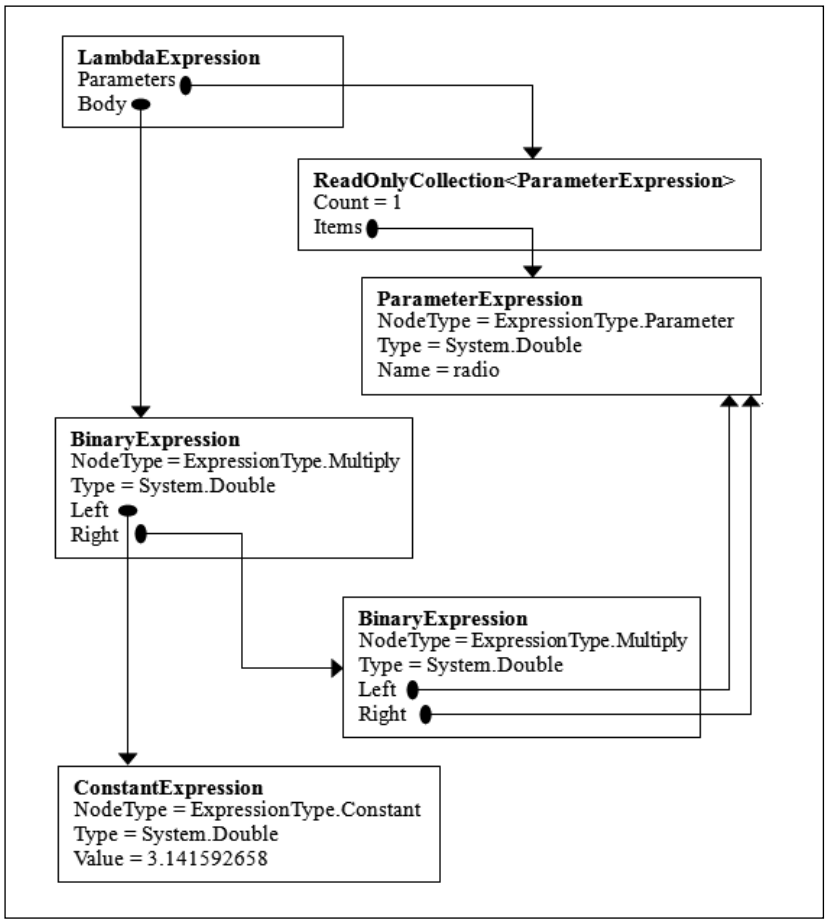


Figura 8.1.

Una vez que se ha construido un árbol de expresión, éste puede ser manipulado casi como cualquier otro objeto .NET – transformado en otro árbol, seriado para su almacenamiento o transmisión a través de la red, etc. En particular, la clase `Expression` ofrece el mecanismo necesario para compilar un árbol de expresión al código IL necesario para su evaluación:

```
Func<double, double> area3 = areaExpr.Compile();
Console.WriteLine("Area(5) = " + area3(5));
```

En el párrafo anterior hemos dicho "casi" porque un detalle a tener en cuenta es que todas las clases de la jerarquía de `Expression` son inmutables —una vez

construida una expresión de cualquier tipo, no hay manera de modificarla *in situ*. Las ventajas de esta inmutabilidad ya se han mencionado en los capítulos 5 y 7.

8.2. LA JERARQUÍA DE CLASES DE EXPRESIONES

Además de la clase `Expression <T>` genérica que hemos utilizado anteriormente, que es la que garantiza el mecanismo de fuerte control de tipos a nivel del lenguaje, existe otra clase `Expression`, ésta no genérica, que es el verdadero "caballo de batalla" sobre el que descansa todo el mecanismo de representación y creación de expresiones. La versión no genérica de `Expression` es similar a las típicas clases que se obtienen al representar estructuras de datos recursivas. Es en sí misma una clase abstracta, y de ella hereda toda una serie de clases tanto abstractas como "concretas" mediante las cuales se representa a los diversos tipos de construcciones que pueden aparecer en una expresión del lenguaje. A continuación se presenta una vista simplificada de la jerarquía de clases que parte de `Expression`:

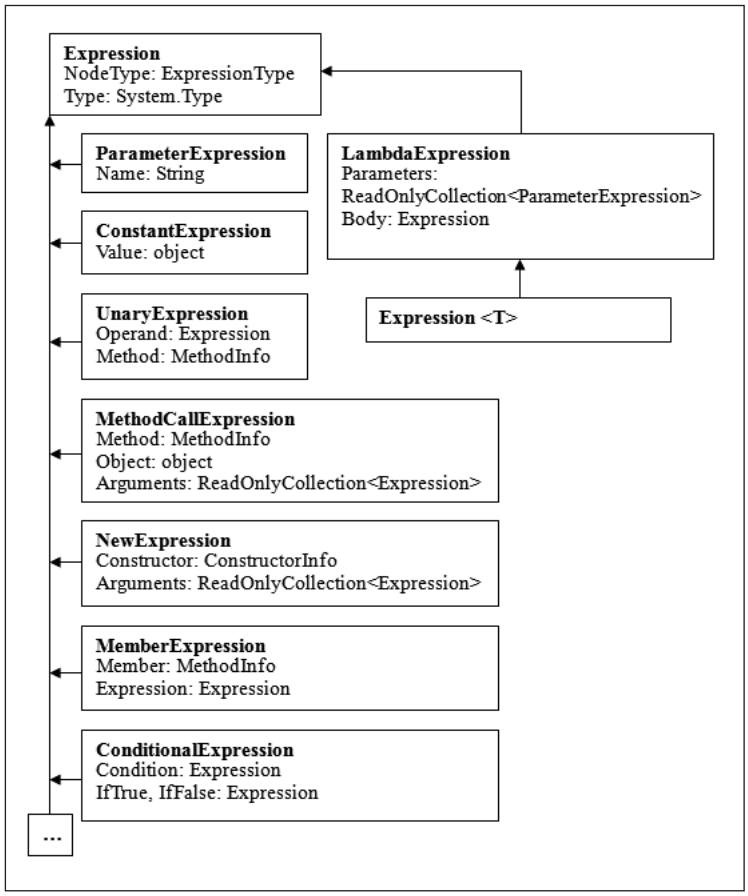


Figura 8.2.

El diagrama anterior muestra solo las subclases más comunes: en total, son alrededor de unas veinte las subclases de `Expression`, y algunas de ellas cubren construcciones que pueden ser poco frecuentes, pero son factibles, como la creación e inicialización de arrays, la promoción de tipos anulables, etc. La siguiente tabla describe el significado de las clases incluidas en el diagrama.

Tabla 8.1.

Clase	Significado
`Expression`	Clase base de la jerarquía. Ofrece dos propiedades que heredan todas sus descendientes: • `NodeType`, del tipo enumerado `ExpressionType`, que indica de manera detallada el tipo concreto de expresión de que se trata. Decimos "detallada" porque, por ejemplo, los cuatro valores de la enumeración `Add`, `Subtract`, `Multiply` y `Divide` corresponden a un mismo tipo de objeto de expresión — `BinaryExpression`. • `Type`, del tipo `System.Type`, que indica el tipo de datos de la expresión. El compilador de árboles de expresiones tiene incorporadas reglas de chequeo de tipos similares a las del propio C#.
`LambdaExpression`	Esta clase representa a una expresión lambda en general, y sirve a la vez como "puente" entre la clase `Expression` no genérica y la clase `Expression<T>` genérica. Cuando el compilador genera un árbol de expresión a partir de una expresión lambda que hemos escrito en el código, el árbol de expresión parte de un objeto de este tipo, como muestra nuestro ejemplo anterior. Sus principales propiedades son `Body`, de tipo `Expression`, que representa el cuerpo de la expresión, y `Parameters`, una colección de objetos `ParameterExpression`, que es la lista de parámetros presentes en la expresión.
`ParameterExpression`	Mediante objetos de este tipo se representan los parámetros (variables) de un árbol de expresión. Para evaluar un árbol, se debe suministrar valores para cada uno de sus parámetros. La principal propiedad en esta clase es `Name`, una cadena que indica el nombre del parámetro.

Continúa

Clase	Significado
`ConstantExpression`	Mediante objetos de este tipo se representan los valores constantes que aparecen en una expresión. Su principal propiedad es `Value`, asociada al valor (constante) de la expresión. `Value` es estáticamente del tipo **`object`**; en ejecución, su tipo será el indicado por la propiedad `Type` heredada.
`UnaryExpression`	Mediante objetos de este tipo se representan expresiones basadas en la aplicación de un operador unario, como el cambio de signo o la conversión de tipo. Su principal propiedad es `Operand`, de tipo `Expression`, asociada al único operando de la expresión. El valor de la propiedad heredada `ExpressionType` determina el tipo concreto de operación de que se trata.
`BinaryExpression`	Mediante objetos de este tipo se representan expresiones basadas en la aplicación de un operador binario, como son la suma, la multiplicación y muchos otros. Las principales propiedades de esta clase son `Left` y `Right`, mediante los que se accede a los nodos (operandos) izquierdo y derecho, respectivamente. El valor de la propiedad heredada `ExpressionType` determina el tipo concreto de operación de que se trata.
`MethodCallExpression`	Mediante objetos de este tipo se representan expresiones basadas en la aplicación de una llamada a un método. Las principales propiedades de esta clase son `Method`, de tipo `MethodInfo` (la información de metadatos del método a llamar); `Object`, el objeto al que se va a aplicar el método (**`null`** en caso de un método estático) y `Parameters`, la lista de las expresiones que sirven como argumentos para la llamada.
`NewExpression`	Mediante objetos de este tipo se representan expresiones basadas en la aplicación de una llamada a un constructor. Las principales propiedades de esta clase son `Constructor`, de tipo `ConstructorInfo` (la información de metadatos del constructor a llamar) y `Parameters`, la lista de las expresiones que sirven como argumentos para la llamada al constructor.

Continúa

Clase	Significado
`MemberExpression`	Mediante objetos de este tipo se representan expresiones basadas en la selección del valor de un campo o propiedad de un objeto. Las principales propiedades de esta clase son `Member`, de tipo `MemberInfo` (la información de metadatos del campo o propiedad a seleccionar) y `Expression`, la expresión para la que se desea seleccionar el valor del campo o propiedad.
`ConditionalExpression`	Mediante objetos de este tipo se representan expresiones del tipo del operador condicional de C# (IIF de VB), que devuelven una expresión u otra en dependencia de si el valor de cierta expresión es verdadero o falso. Las principales propiedades de esta clase son `Condition`, de tipo `Expression` (la expresión a evaluar, que deberá devolver un **`bool`**), e `IfTrue` e `IfFalse`, las expresiones a devolver en caso de que la condición produzca **`true`** o **`false`**, respectivamente.

¿Cómo se crean normalmente instancias de cada uno de esos tipos de expresiones? Aunque podría hacerse invocando a constructores, la vía recomendada pasa por utilizar la propia clase `Expression` (raíz de la jerarquía), que ofrece todo un conjunto de métodos fábrica estáticos capaces de generar expresiones de cada una de sus clases derivadas. El hecho de que estos métodos disponen de parámetros de tipo `Expression` hace posible el anidamiento recursivo de expresiones.

La siguiente tabla lista algunos de estos métodos fábrica. Esta tabla no es exhaustiva y en varios casos se muestra solo una de las sobrecargas existentes, generalmente la más representativa.

Tabla 8.2.

Clase	Métodos fábrica que crean objetos de esa clase

`LambdaExpression`	`Expression.Lambda(`
	`  Expression expresión,`
	`  ParameterExpression[]`
	`  parámetros);`

`ParameterExpression`	`Expression.Parameter(`
	`  Type tipo,`
	`  string nombre);`

`ConstantExpression`	`Expression.Constant(` `  object objeto);`

Continúa

Clase	Métodos fábrica que crean objetos de esa clase

UnaryExpression	`// cambio de signo`
	`Expression.Negate(`
	`  Expression expresión);`
	`// conversión de tipo`
	`Expression.Convert(`
	`  Expression expresión,`
	`  Type tipo);`

BinaryExpression	`//sumar`
	`Expression.Add(`
	`  Expression expresión1,`
	`  Expression expresión2);`
	`// restar`
	`Expression.Subtract(`
	`  Expression expresión1,`
	`  Expression expresión2);`
	`// multiplicar`
	`Expression.Multiply(`
	`  Expression expresión1,`
	`  Expression expresión2);`
	`// dividir`
	`Expression.Divide(`
	`  Expression expresión1,`
	`  Expression expresión2);`
	`// elevar a potencia`
	`Expression.Power(`
	`  Expression base,`
	`  Expression exponente);`
	Nota:
	El autor de este libro sugirió la adición de `Power()`, inicialmente ausente, a la API de expresiones.

MethodCallExpression	`// llamada a método de instancia`
	`Expression.Call(`
	`  Expression objeto,`
	`  MethodInfo método,`
	`  Expression[] argumentos);`
	`// llamada a un método estático`
	`Expression.Call(`
	`  MethodInfo método,`
	`  Expression[] argumentos);`
	`// llamada a método de instancia (virt)`
	`Expression.VirtualCall(`
	`  Expression objeto,`
	`  MethodInfo método,`
	`  Expression[] argumentos);`

Continúa

Clase	Métodos fábrica que crean objetos de esa clase

NewExpression	`// llamada a un constructor`
	`Expression.New(`
	`  ConstructorInfo constructor,`
	`  Expression[] argumentos);`

MemberExpression	`// obtener valor de un campo`
	`Expression.Field(`
	`  Expression objeto,`
	`  FieldInfo campo);`
	`// obtener valor de una propiedad`
	`Expression.Property(`
	`  Expression objeto,`
	`  PropertyInfo propiedad);`

ConditionalExpression	`// producir una y otra expresión`
	`// en función de una condición`
	`Expression.Condition(`
	`  Expression condición,`
	`  Expression siVerdadero,`
	`  Expression siFalso);`

Esta arquitectura basada en métodos fábrica es la que promueve el estilo de programación funcional que se utiliza normalmente al definir mediante código un árbol de expresión: la expresión se compone de una cadena de llamadas anidadas que van construyendo cada uno de los elementos de la expresión, "ensartando" así unos con otros.

8.3. MÁS EJEMPLOS

Como otro ejemplo de la construcción mediante código de árboles de expresiones, veamos cómo se construiría el árbol correspondiente a la hipotenusa de un triángulo rectángulo (la raíz de la suma de los cuadrados de los otros dos lados):

```
ParameterExpression px =
   Expression.Parameter(typeof(double), "x");
ParameterExpression py =
   Expression.Parameter(typeof(double), "y");
ParameterExpression [] parms = { px, py };

Expression<Func<double, double, double >> hipotenusa =
   Expression.Lambda<Func<double, double,double>>
```

Continúa

```
      (Expression.Call(
         typeof(Math).GetMethod("Sqrt"),
         new Expression [] {
            Expression.Add(
               Expression.Multiply(px, px),
               Expression.Multiply(py, py))
         }),
      parms);

Func<double, double, double> hipot2 =
   hipotenusa.Compile();
Console.WriteLine(hipot2(3, 4));
```

Como la hipotenusa se calcula en base a los otros dos lados del triángulo, la expresión necesita dos parámetros, así que creamos dos objetos `ParameterExpression`, y con referencias a ellos conformamos el array de parámetros que requiere el método fábrica `Expression.Lambda<T>()`. Luego volvemos a utilizar estos parámetros durante la construcción de la expresión en sí.

Para "escapar" un poco del mundo de las matemáticas, veamos ahora un ejemplo de construcción de un árbol de expresión para el manejo de cadenas. El siguiente árbol tiene como objetivo "formatear" el nombre completo de una persona, a partir del nombre y el apellido. Para ello, elimina los posibles espacios en ambas cadenas, las convierte a mayúsculas, y luego las concatena con un espacio en blanco por medio. En C# 3.0 la definición es ésta:

```
Expression<Func<string, string, string>> combinación =
   (s1, s2) => (s1.Trim() + " " + s2.Trim()).ToUpper();
```

Si nos encargamos nosotros de la construcción del árbol, podría quedarnos así:

```
ParameterExpression ps1 =
   Expression.Parameter(typeof(string), "nombre");
ParameterExpression ps2 =
   Expression.Parameter(typeof(string), "apellido");
ParameterExpression [] parms2 = { ps1, ps2 };

Expression<Func<string, string, string>> combinación1 =
   Expression.Lambda<Func<string, string, string>>(
      Expression.Call(
         Expression.Call(
            typeof(string).GetMethod("Concat",
```

Continúa

```
            new[] {
               typeof(string),
               typeof(string),
               typeof(string)}),
         new Expression[] {
            Expression.Call(
               ps1,
               typeof(string).GetMethod("Trim",
                  new Type[] {}),
               null),
            Expression.Constant(" "),
            Expression.Call(
               ps2,
               typeof(string).GetMethod("Trim",
                  new Type[] {}),
               null)
         }
      ),
      typeof(string).GetMethod("ToUpper", new Type[] {}),
      null),
   parms2);

Func<string, string, string> nombreCompleto =
   combinación1.Compile();
Console.WriteLine(nombreCompleto(" Pepe ", " Reyes "));
```

Después de estos ejercicios cuasi-masoquistas, espero que haya comprendido los principios generales de construcción de los árboles de expresiones.

8.4. MANIPULACIÓN PROGRAMÁTICA DE ÁRBOLES

Como ya hemos comentado antes, una de las grandes ventajas de representar las funciones lambda mediante árboles de expresiones es que abre ante nosotros la posibilidad de crearlos y manipularlos programáticamente. En este sentido, el ejemplo perfecto para mostrar esas posibilidades consiste en desarrollar un **intérprete de expresiones matemáticas.** Un intérprete de expresiones es una aplicación que recibe del usuario una cadena de caracteres que representa una expresión matemática (como `2 * sin(x)* cos(x)`, por ejemplo) y la traduce a una representación interna que permite la posterior evaluación de la expresión para diferentes valores de las variables (`x` en este caso). Un intérprete de expresiones basado en los árboles de expresiones de C# 3.0 utilizaría los árboles de expresiones para la representación interna de las mismas.

Supongamos para simplificar que se trata de expresiones de un única variable `x`, y que únicamente se utilizan los cuatro operadores aritméticos. Inicialmente podríamos

definir la siguiente clase, con dos miembros para implementar la transformación de texto a expresión y la posterior evaluación de ésta:

```
public class ExprMatematica
{
   private Expression<Func<double, double>> expresión;

   public ExprMatematica(string cadena)
   {
      // recibe una cadena de caracteres y
      // la transforma en un árbol de expresión,
      // que guarda en 'expresión'
   }

   public double Evaluar(double x)
   {
      // evalúa 'expresión' para el valor 'x'
   }
}
```

¿Cómo implementaríamos el método `Evaluar()`? La vía rápida sería ésta:

```
public double Evaluar(double x)
{
   Func<double, double> f = expresión.Compile();
   return f(x);
}
```

Pero ya se imaginará que yo tan fácil no lo quiero. Podríamos recorrer de manera recursiva los nodos del árbol:

```
private static double Evaluar(Expression e, double valor)
{
switch (e.NodeType)
{
   // operadores binarios
   case ExpressionType.Add:
      BinaryExpression be1 = (BinaryExpression) e;
      return Evaluar(be1.Left, valor) +
         Evaluar(be1.Right, valor);
   case ExpressionType.Subtract:
      BinaryExpression be2 = (BinaryExpression) e;
      return Evaluar(be2.Left, valor) -
         Evaluar(be2.Right, valor);
```

Continúa

```
    case ExpressionType.Multiply:
      BinaryExpression be3 = (BinaryExpression) e;
      return Evaluar(be3.Left, valor) *
        Evaluar(be3.Right, valor);
    case ExpressionType.Divide:
      BinaryExpression be4 = (BinaryExpression) e;
      return Evaluar(be4.Left, valor) /
        Evaluar(be4.Right, valor);
    // operador unario
    case ExpressionType.Negate:
      UnaryExpression ue = (UnaryExpression) e;
      return -Evaluar(ue.Operand, valor);
    // constante
    case ExpressionType.Constant:
      ConstantExpression ce = (ConstantExpression) e;
      return (double) ce.Value;
    // variable
    case ExpressionType.Parameter:
      return valor;
    // otra cosa
    default:
      throw new ArgumentException("Nodo no soportado");
    }
}
```

El anterior método es un ejemplo de cómo podemos navegar para nuestros fines por un árbol de expresión. Por supuesto, también podemos construir un nuevo árbol a partir de uno ya existente (no así modificar directamente un árbol, son inmutables). Por ejemplo, analice el siguiente método, que convierte una instancia de nuestra función matemática en su inversa (`1/x`):

```
public void Invertir()
{
  expresión =
    Expression.Lambda<Func<double, double>>(
      Expression.Divide(
        Expression.Constant(1.0), expresión),
      expresión.Parameters);
}
```

Un ejemplo más potente en ese sentido se presenta, como colofón para este capítulo, en la siguiente sección. Pero antes debemos terminar ésta, y para ello mostraremos la implementación del análisis lexical y sintáctico que necesita el constructor de la clase. No piense el lector que este código se presenta gratuitamente, para "estirar" el capítulo. De él podrá sacar más ejemplos de técnicas "modernas"

de programación en C#. El *scanner*, por ejemplo, se ha programado para que implemente la interfaz `IEnumerable<T>` (que es de lo que va realmente este libro, créame :-). Gracias a eso, el *parser* puede ir recorriendo los diferentes lexemas de la expresión utilizando un bucle **`foreach`.**

```
// *** Tipos comunes y scanner
enum TokenType
{
   Var, Number, Plus, Minus, UnaryMinus,
   Mult, Div, ParOpen, ParClose, End
};
struct Token
{
   public readonly TokenType Type;
   public readonly double Value; // only used for constants

   public Token(TokenType tt)
   {
      Type = tt; Value = 0;
   }
   public Token(TokenType tt, double val)
   {
      Type = tt; Value = val;
   }
}

private static int[,] canFollow =
{
   // Var Number Plus Minus UMinus Mul Div Par( Par) END
   { 0, 0, 1, 1, 1, 1, 1, 1, 0, 0 }, // Var
   { 0, 0, 1, 1, 1, 1, 1, 1, 0, 0 }, // Number
   { 1, 1, 0, 0, 0, 0, 0, 0, 1, 0 }, // Plus
   { 1, 1, 1, 1, 1, 1, 1, 1, 1, 0 }, // Minus
   { 1, 1, 1, 1, 1, 1, 1, 1, 1, 0 }, // UnaryMinus
   { 1, 1, 0, 0, 0, 0, 0, 0, 1, 0 }, // Mul
   { 1, 1, 0, 0, 0, 0, 0, 0, 1, 0 }, // Div
   { 0, 0, 1, 1, 0, 1, 1, 1, 0, 0 }, // ParOpen
   { 1, 1, 0, 0, 0, 0, 0, 0, 1, 0 }, // ParClose
   { 1, 1, 0, 0, 0, 0, 0, 0, 1, 0 } // END
};

class Scanner : IEnumerable<Token>
{
   private int pos = 0;
   private string cadena;
```

Continúa

```
public Scanner(string cadena)
{
   // PRECOND: cadena!= null
   this.cadena = cadena.Trim().ToUpper();
}

IEnumerator IEnumerable.GetEnumerator()
{
   return GetEnumerator();
}

public IEnumerator<Token> GetEnumerator()
{
   TokenType previous = TokenType.ParOpen,
      current = TokenType.End;
   int balance = 0;
   double value;

   while (pos < cadena.Length)
   {
      value = 0.0;
   loop:
      switch (cadena[pos])
      {
         case ' ':
         case '\t':
         case '\n':
            pos++;
            goto loop;
         case '(':
            current = TokenType.ParOpen;
            balance++;
            break;
         case ')':
            current = TokenType.ParClose;
            balance--;
            if (balance < 0)
               throw new ArgumentException(
                  "SYNTAX: Nesting error");
            break;
         case '+':
            current = TokenType.Plus; break;
         case '-':
            if (previous != TokenType.Var &&
                 previous != TokenType.Number)
                 current = TokenType.UnaryMinus;
```

Continúa

```
            else
               current = TokenType.Minus;
            break;
         case '*':
            current = TokenType.Mult; break;
         case '/':
            current = TokenType.Div; break;
         case 'X':
            current = TokenType.Var; break;
         case '0':
         case '1':
         case '2':
         case '3':
         case '4':
         case '5':
         case '6':
         case '7':
         case '8':
         case '9':
            // a number
            bool hasDot = false;
            StringBuilder sb = new StringBuilder();
            while (pos < cadena.Length &&
               ".0123456789".Contains(cadena[pos]))
            {
               if (cadena[pos] == '.')
                  if (hasDot)
                           throw new ArgumentException(
                           "SCAN: Invalid constant");
                  else
                    hasDot = true;
               sb.Append(cadena[pos]);
               pos++;
            }
            current = TokenType.Number;
            value = double.Parse(sb.ToString());
            pos--;
            break;
      default:
         throw new ArgumentException(
            "SCAN: Illegal character");
   }
   if (canFollow[(int)current, (int)previous] == 1)
   {
      previous = current;
      yield return new Token(current, value);
      pos++;
   }
```

Continúa

```
      else
         throw new ArgumentException(
            "SCAN: syntax error");
      }
      if (canFollow[(int)TokenType.End, (int)previous] == 1)
         yield return new Token(TokenType.End);
      else
         throw new ArgumentException("SCAN: Syntax error");
   }
}
```

Por su parte, el *parser* genera el árbol de expresiones mediante un algoritmo **recursivo descendente:**

```
// *** Parser
class Parser
{
   private Scanner scanner = null;
   private Expression<Func<double, double>> expr = null;

   public Expression<Func<double, double>> ResultTree
   {
      get { return expr; }
   }

   public Parser(string cadena)
   {
      scanner = new Scanner(cadena);
      // generate tree
      tokens = scanner.GetEnumerator();
      GetToken();
      Expression exp = null;
      Gen1(out exp);
      expr = Expression.Lambda<Func<double, double>>(
         exp,
         new[] {Expression.Parameter(typeof(double), "x")}
      );
   }

   private IEnumerator<Token> tokens;
   private Token curr;
   private void GetToken()
   {
      tokens.MoveNext();
      curr = tokens.Current;
   }
```

Continúa

```
private void Gen1(out Expression exp)
{
   Token t;
   Expression part;

   Gen2(out exp);
   while ((t = curr).Type == TokenType.Plus ||
      t.Type == TokenType.Minus)
   {
      GetToken();
      Gen2(out part);
      switch (t.Type)
      {
           case TokenType.Plus:
           exp = Expression.Add(exp, part);
           break;
        case TokenType.Minus:
           exp = Expression.Subtract(exp, part);
           break;
      }
   }
}

private void Gen2(out Expression exp)
{
   Token t;
   Expression part;

   Gen3(out exp);
   while ((t = curr).Type == TokenType.Mult ||
      t.Type == TokenType.Div)
   {
      GetToken();
      Gen3(out part);
      switch (t.Type)
   {
        case TokenType.Mult:
           exp = Expression.Multiply(exp, part);
           break;
        case TokenType.Div:
           exp = Expression.Divide(exp, part);
        break;
      }
   }
}
```

Continúa

```
   private void Gen3(out Expression exp)
   {
      bool isMinus = false;

      if (curr.Type == TokenType.UnaryMinus)
      {
         isMinus = true;
         GetToken();
      }
      Gen4(out exp);
      if (isMinus)
         exp = Expression.Negate(exp);
      }

   private void Gen4(out Expression exp)
   {
      if (curr.Type == TokenType.ParOpen)
      {
         GetToken();
         Gen1(out exp);
         if (curr.Type != TokenType.ParClose)
            throw new ArgumentException(
               "PARSE: Nesting error");
         GetToken();
      }
      else
         Atom(out exp);
      }

   private void Atom (out Expression exp)
   {
      switch (curr.Type)
      {
         case TokenType.Number:
            exp = Expression.Constant(curr.Value);
            GetToken();
            break;
         case TokenType.Var:
            exp = Expression.Parameter(typeof(double), "x");
            GetToken();
            break;
         default:
            exp = null;
            break;
      }
   }
}
```

Con esta clase a nuestra disposición, podremos hacer cosas como ésta:

```
ExprMatematica em = new ExprMatematica("5*(x + 3)/(x - 1)");
Console.WriteLine(em.Evaluar(3)); // imprime 15
```

8.5. CÁLCULO DE DERIVADAS

Esta sección pretende ser una muestra de las posibilidades que se abren gracias a la disponibilidad de los árboles de expresiones en C# 3.0 para la manipulación simbólica de expresiones, tarea en la que precisamente siempre brillaron lenguajes de programación funcional como LISP, sobre el que se construyeron afamados sistemas de cálculo simbólico como DERIVE o MATHCAD. Aquí presentamos una posible implementación de una librería para el cálculo de derivadas de funciones matemáticas, ejemplo clásico de tarea de cálculo simbólico. Para una presentación básica del concepto de derivada, remitimos al lector a cualquier libro de análisis matemático básico; para los propósitos de esta sección, es suficiente con que digamos que la derivada de una función `f(x)` (por ejemplo, `x + sin(x)`) es otra función (para el caso del ejemplo, `1 + cos(x)`) que se obtiene siguiendo un conjunto de reglas de manipulación simbólica como, por ejemplo, la **regla de la suma:** la derivada de una suma es igual a la suma de las derivadas de los sumandos.

La tarea que tenemos por delante es la de implementar una librería que nos permita, dado un árbol de expresión (un objeto de la clase `Expression<T>`), obtener otro objeto de la misma clase que represente a la derivada de la función original. Llamaremos `Derivada()` al método (sobrecargado) que se encargará de esta tarea; para implementarlo, aprovecharemos otra nueva característica de C# 3.0 ya presentada: los métodos extensores (Capítulo 6). Gracias al uso de métodos extensores, podremos invocar al método `Derivada()` sobre cualquier expresión utilizando la sintaxis natural de la orientación a objetos:

```
Console.WriteLine(ex.Derivada());
```

En lugar de la más procedimental:

```
Console.WriteLine(ExpressionExtensions.Derivada(ex));
```

El código fuente de la clase extensora es el siguiente:

```
public static class ExpressionExtensions
{
   // método auxiliar
```

Continúa

```
private static Expression
   Derivada(this Expression e, string param)
{
   if (e == null)
      throw new ArgumentException("Expresión nula");
   switch (e.NodeType)
   {
      // regla de constante
      case ExpressionType.Constant:
         return Expression.Constant(0.0);
      // parámetro
      case ExpressionType.Parameter:
         if (((ParameterExpression) e).Name == param)
            return Expression.Constant(1.0);
         else
            return Expression.Constant(0.0);
      // cambio de signo
      case ExpressionType.Negate:
         Expression op = ((UnaryExpression) e).Operand;
         return Expression.Negate(op.Derivada(param));
      // regla de la suma
      case ExpressionType.Add:
      {
         Expression dleft =
            ((BinaryExpression) e).Left.Derivada(param);
         Expression dright =
            ((BinaryExpression) e).Right.Derivada(param);
         return Expression.Add(dleft, dright);
      }
      // regla de la resta
      case ExpressionType.Subtract:
      {
         Expression dleft =
            ((BinaryExpression) e).Left.Derivada(param);
         Expression dright =
            ((BinaryExpression) e).Right.Derivada(param);
         return Expression.Subtract(dleft, dright);
      }
      // regla de la multiplicación
      case ExpressionType.Multiply:
      {
         Expression left = ((BinaryExpression) e).Left;
         Expression right = ((BinaryExpression) e).Right;
         Expression dleft = left.Derivada(param);
         Expression dright = right.Derivada(param);
         return Expression.Add(
         Expression.Multiply(left, dright),
         Expression.Multiply(dleft, right));
      }
```

Continúa

```
        // regla del cociente
        case ExpressionType.Divide:
        {
           Expression left = ((BinaryExpression) e).Left;
           Expression right = ((BinaryExpression) e).Right;
           Expression dleft = left.Derivada(param);
           Expression dright = right.Derivada(param);
           return Expression.Divide(
              Expression.Subtract(
                 Expression.Multiply(left, dright),
                 Expression.Multiply(dleft, dright)),
              Expression.Multiply(right, right));
        }

        // potencia (exponente constante) ***
        case ExpressionType.Power:
           {
              Expression left = ((BinaryExpression)e).Left;
              Expression right = ((BinaryExpression)e).Right;
              Expression dleft = left.Derivada(param);
              return Expression.Multiply(
                 dleft,
                 Expression.Power(
                    left,
                    Expression.Subtract(
                      right,
                      Expression.Constant(1.0)
                    )
                 )
              );
        }

        // llamada a la función
        case ExpressionType.Call:
           Expression e1 = null;
           MethodCallExpression me =
              (MethodCallExpression) e;
           MethodInfo mi = me.Method;
           // COMPROBACIÓN
           // el método debe ser estático y su clase Math
           if (!mi.IsStatic ||
                mi.DeclaringType.FullName != "System.Math")
                throw new ArgumentException(
                "NO IMPLEMENTADO");

           ReadOnlyCollection<Expression> parms =
              me.Arguments;
           switch (mi.Name)
           {
              case "Pow":
```

Continúa

```
                // regla de la potencia
                e1 = Expression.Multiply(
                   parms[1],
                   Expression.Call(
                     mi,
                     new Expression[] {
                                parms[0],
Expression.Subtract(parms[1], Expression.Constant(1.0))
                     }));
                break;
             case "Sin":
                // la derivada del seno es el coseno
                e1 = Expression.Call(
                          typeof(Math).GetMethod(
                          "Cos",
                          new Type[] {typeof(double)}),
                          new Expression[] {
                     parms[0]
                });
                break;
             default:
                throw new ArgumentException(
                   "PENDIENTE");
     }
     // regla de la cadena
     return Expression.Multiply(e1,
        parms[0].Derivada(param));
     // otros
     default:
        throw new ArgumentException(
           "No soportada: " + e.NodeType.ToString());
     }
}

   public static Expression<T>
      Derivada<T>(this Expression<T> e)
   {
      if (e == null)
         throw new ArgumentException("Expresión nula");

         // comprobar que hay una sola variable
         if (e.Parameters.Count != 1)
            throw new ArgumentException("Un solo parámetro!");
         return Expression.Lambda<T>(
            e.Body.Derivada(e.Parameters[0].Name),
            e.Parameters);
   }

   public static Expression <T>
      Derivada <T>(this Expression<T> e, string param)
```

Continúa

```
    {
        if ( e == null)
            throw new ArgumentException("Expresión nula");
        // comprobar que 'param' existe
        bool ok = false;
        foreach (ParameterExpression p in e.Parameters)
            if (p.Name == param)
            {
                ok = true; break;
            }
        if (!ok)
            throw new ArgumentException("Parámetro inválido");
        return Expression.Lambda<T>(
            e.Body.Derivada(e.Parameters[0].Name),
            e.Parameters);
    }

    // *** PENDIENTE ( se deja como ejercicio al lector)

    public static Expression<T>
        Simplificar <T>(this Expression<T> e)
    {
        if ( e == null)
            throw new ArgumentException("Expresión nula");
        // TODO
        return e;
    }

    public static void Dibujar<T>(this Expression<T> e,
        Graphics dc, Rectangle rect)
    {
        if (e == null)
            throw new ArgumentException("Expresión nula");
        // TODO: dibujar el rectángulo 'rect' de 'dc'
    }
}
```

La clase `ExpressionExtensions` ofrece dos sobrecargas de `Derive()`:

- Una con un único parámetro, la expresión a derivar. Esta variante asume que la expresión tiene un único parámetro.
- Una segunda versión con un segundo parámetro, el nombre del parámetro (variable) por el que derivar. Con esta variante se implementan las llamadas **derivadas parciales;** básicamente, si una función tiene varias variables, se puede calcular la derivada usando cualquiera de ellas como "la variable" y considerando las demás como valores constantes.

Ambas se apoyan en el verdadero motor de la clase: un método privado que implementa toda la funcionalidad de cálculo de derivadas de una manera recursiva. En él se expresan las reglas de derivación para los distintos tipos de expresiones posibles.

Después de agregar una referencia a la librería en cualquier proyecto, solo será necesario importar el espacio de nombres `PlainConcepts.Expressions`, y podremos calcular derivadas de funciones de la siguiente forma:

```
Expression<Func<double, double>> seno = x => Math.Sin(x);
Expression<Func<double, double>> dSeno = seno.Derivada();
Console.WriteLine(dSeno);
```

El resultado que obtendremos en pantalla vendrá determinado por la implementación del método `ToString()` en `Expression<T>`:

```
x => (Cos(x) * 1)
```

Para llegar a este resultado, el método extensor ha aplicado dos reglas, la "regla del seno" y luego la "regla de la cadena". El uno que aparece en la expresión es la derivada de `x`. Multiplicar por uno es superfluo, pero desgraciadamente, nuestra clase no es capaz de simplificar las expresiones que genera.

8.6. Y AHORA, LOS DEBERES...

Una implementación completa del método para el cálculo de derivadas no es una tarea sencilla, y la versión que se ofrece con el código fuente de este libro no es completa; fundamentalmente queda trabajo pendiente en lo relativo a la programación de las derivadas de las diferentes funciones elementales (logarítmicas, trigonométricas, etc.) que pueden aparecer en una expresión.

Otra de las tareas pendientes que recomendamos al lector como ejercicio es la implementación de un método extensor para la simplificación y factorización de expresiones. La programación de reglas tan simples como **0 + y = y** ó **1 * y = y** para cualquier **y** dado nos habría permitido obtener directamente el resultado ideal en el ejemplo anterior.

Por último, una tarea más compleja, pero que se antoja interesante, es la de dibujar gráficamente sobre un contexto de dispositivo un árbol de expresión, utilizando los símbolo tradicionales de la notación matemática.

Eventualmente, posibles soluciones a estos ejercicios aparecerán en mi blog: **http://geeks.ms/blogs/ohernandez.**

Espero que este capítulo le haya ayudado a comprender la potencia de los árboles de expresiones, que permiten representar como datos de una manera eficiente las expresiones lambda para su posterior tratamiento y evaluación en el momento oportuno. Según nos acerquemos al corazón de este libro, verá cómo los árboles de expresiones juegan un papel esencial en la implementación de ciertos proveedores de **LINQ,** y especialmente en **LINQ to SQL,** donde es crucial la manipulación de árboles de expresiones para, en el momento oportuno, transformarlos en las sentencias SQL que deben enviarse al motor de bases de datos.

Consultas integradas en C#

CAPÍTULO 9

Fundamentos de LINQ

Llegados a este punto, hemos sentado las bases necesarias para enfrentarnos al verdadero protagonista de este libro: las consultas integradas en el lenguaje. Según la modesta opinión de este autor, las consultas integradas (la principal expresión en los lenguajes .NET de la tecnología LINQ) constituyen uno de los avances más significativos de los últimos tiempos en el área de los lenguajes y sistemas de programación, y jugarán un papel fundamental de cara a reducir a su mínima expresión el fenómeno conocido como "**desajuste de impedancias**" (*impedance mismatch*) provocado por la obligación vigente hasta el momento de utilizar, al desarrollar aplicaciones, no solo nuestro lenguaje de programación habitual (que seguramente será C# si está usted leyendo este libro), sino además toda una serie de otros lenguajes diferentes para acceder a las más diversas fuentes de datos, como SQL o XPath/XQuery, cuyas instrucciones se "incrustan" actualmente de manera literal dentro del código C#. Gracias a C# 3.0 y LINQ, podremos realmente empezar a crear aplicaciones que contengan única y exclusivamente código C#, utilizando una sintaxis clara y natural para acceder a esos almacenes de datos, y dejando al compilador y las librerías de soporte la tarea de generar esas construcciones "foráneas".

9.1. PRESENTACIÓN DE LINQ

LINQ (**L**anguage **In**tegrated **Q**uery) es una combinación de extensiones al lenguaje y librerías de código manejado que permite expresar de manera uniforme las consultas sobre colecciones de datos provenientes de las más disímiles fuentes: objetos en memoria, bases de datos relacionales o documentos XML, entre otros.

El propio significado del acrónimo (Consultas Integradas en el Lenguaje) ofrece la clave para comprender el objetivo principal de LINQ: permitir que el programador exprese las consultas sobre datos de diversa procedencia utilizando recursos de su propio lenguaje de programación. Las ventajas fundamentales de esta posibilidad son las siguientes:

- A la hora de expresar sus consultas, el programador podrá aprovechar todas las facilidades que el compilador y el entorno integrado ofrecen (verificación de sintaxis y de tipos por parte del compilador, ayuda IntelliSense dentro del entorno, acceso a metadatos por parte de ambos).

- LINQ permitirá, si no eliminar completamente, sí reducir en gran medida el ya mencionado fenómeno del "desajuste de impedancias" que provocan las diferencias entre los modelos de programación que proponen los lenguajes de propósito general (en nuestro caso C#) y los lenguajes de consulta sobre bases de datos relacionales (SQL), documentos XML (XPath/XQuery) y otros almacenes de datos.
- Por último, otra ventaja importante de LINQ es que permitirá elevar el nivel de abstracción y claridad de la programación de las consultas. Actualmente, un programador que necesite acceder a una base de datos, por ejemplo, deberá especificar en un plan meticuloso **cómo** recuperar los datos que necesita; las expresiones de consulta, en cambio, son una herramienta mucho más declarativa, que permite en gran medida indicar únicamente **qué** se desea obtener, dejando al motor de evaluación de expresiones los detalles sobre cómo lograr ese objetivo.

El otro elemento clave de la tecnología LINQ es su arquitectura abierta, que hace posible la extensibilidad. La semántica de los operadores en las expresiones de consulta, como veremos a continuación, no está en modo alguno "cableada" en el lenguaje, sino que puede ser modificada (o extendida) por librerías de Microsoft o de terceros fabricantes para el acceso a fuentes de datos específicas. El propio Microsoft pone a nuestra disposición, además del mecanismo estándar para aplicar LINQ a arrays y colecciones genéricas (LINQ to Objects), al menos otras tres tecnologías: **LINQ to XML,** para ejecutar consultas integradas sobre documentos XML (Capítulo 12); **LINQ to DataSets,** para la ejecución de consultas al estilo LINQ contra conjuntos de datos tipados y no tipados (Capítulo 13) *y* **LINQ to SQL,** para hacer posible las consultas (y también la actualización) de bases de datos relacionales mediante recursos del lenguaje de programación (Capítulo 14). Pero la arquitectura de las herramientas que se ponen a nuestra disposición con LINQ es tal, que ha hecho posible la aparición de tecnologías ("proveedores") que simplifican y uniformizan el acceso a otras fuentes de datos: **LINQ to Amazon** y **LINQ to SharePoint** son dos de los ejemplos más contundentes que se pueden encontrar al navegar por la red.

La arquitectura de LINQ puede ser descrita gráficamente de la siguiente forma:

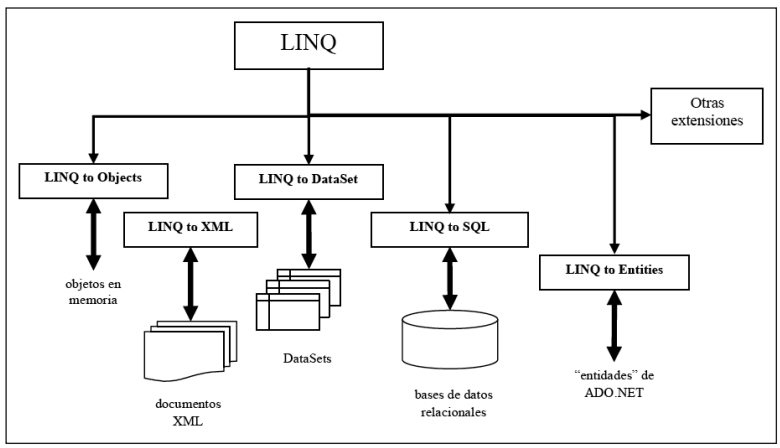


Figura 9.1.

9.2. LAS EXPRESIONES DE CONSULTA

Las **expresiones de consulta** (*query expressions*) son el principal mecanismo mediante el cual cobra vida la tecnología LINQ. No son otra cosa que expresiones que responden a una nueva sintaxis que se ha añadido a C# 3.0 (así como a Visual Basic, además de que se espera que otros lenguajes la adopten también) y que pueden actuar sobre cualquier objeto que implemente la interfaz genérica `IEnumerable<T>` que hemos presentado en el Capítulo 2 (en particular, los arrays y colecciones de .NET 2.0 implementan esta interfaz), transformándolo mediante un conjunto de operadores en otros objetos, generalmente (pero no siempre) objetos que implementan esa misma interfaz.

Nota:

Al referirnos a un objeto que implementa la interfaz `IEnumerable<T>`, utilizaremos frecuentemente el término "**secuencia**"; no sería correcto decir "array" ni "colección", y "enumerable" (que en el fondo es lo que son, objetos que se pueden enumerar) podría causar confusión con los tipos enumerados **(enum)**.

Comenzaremos con un pequeño ejemplo que nos mostrará en qué consisten básicamente las expresiones de consulta. Para ello supondremos que tenemos definida las siguientes colecciones de objetos de las clases `Persona` y `Pais` definidas en la introducción:

DIRECTORIO DEL CODIGO: EJEMPLO09 _ 01

```
List<Persona> Generación = new List<Persona> {
   new Persona { Nombre = "Diana",
      Sexo = SexoPersona.Mujer,
      FechaNac = new DateTime(1996, 2, 4),
      CodigoPaisNac = "ES" },
   new Persona { Nombre = "Dennis",
      Sexo = SexoPersona.Varón,
      FechaNac = new DateTime(1983, 12, 27),
      CodigoPaisNac = "RU" },
   new Persona { Nombre = "Claudia",
      Sexo = SexoPersona.Mujer,
      FechaNac = new DateTime(1989, 7, 26),
      CodigoPaisNac = "CU" },
   new Persona { Nombre = "Jennifer",
      Sexo = SexoPersona.Mujer,
      FechaNac = new DateTime(1982, 8, 12),
      CodigoPaisNac = "CU" }
};

List<Pais> Paises = new List<Pais> {
   new Pais("ES", "ESPAÑA"),
   new Pais("CU", "CUBA"),
   new Pais("RU", "RUSIA"),
   new Pais("US", "ESTADOS UNIDOS")
};
```

Suponga que deseamos obtener otra colección con los nombres y edades de los mayores de 20 años que pertenecen a la lista original, con los nombres convertidos a mayúsculas. Los objetos de la colección resultante deberán, además, aparecer en orden alfabético.

Si no conoce todavía las maravillas de las consultas integradas, casi con total certeza lo primero que le vendrá a la mente será buscar un mecanismo para recorrer la colección original, creando nuevos objetos que contengan las características necesarias de las personas que cumplen con los requisitos solicitados; para luego ordenar alfabéticamente por los nombres esa colección en una segunda "pasada". Gracias a LINQ, ¡esa vía se convierte en anacrónica!

En C# 3.0 habrá una manera mucho más clara y elegante de obtener ese resultado de un solo golpe:

```
var mayores20  = from h in Generación
                 where h.Edad > 20
                 orderby h.Nombre
                 select new {
                   Nombre = h.Nombre.ToUpper(),
                   Edad = h.Edad };

foreach(var h in mayores20)
  Console.WriteLine(h.Nombre + " (" + h.Edad + ")");
```

La semántica de la sentencia de asignación inicial será intuitivamente clara para cualquiera que esté familiarizado con la sentencia SELECT de SQL. Lo menos habitual es el hecho de que la cláusula **`select`,** en la que se especifica qué se desea obtener, va situada al final, a diferencia de SQL. Este hecho fue y sigue siendo motivo de polémica, pero hoy por hoy la decisión ya está tomada y solo nos queda habituarnos a lo que hay. La razón para ese cambio de posición tiene bastante lógica: si **`select`** estuviera al principio, sería imposible ofrecer al programador ayuda Intellisense a la hora de teclear los datos a seleccionar, porque aún no se habría introducido la colección de datos sobre la que se va a ejecutar la consulta. Teniendo a ese "operando" en primer lugar, es fácil para el entorno integrado ofrecer ayuda a lo largo de todo el proceso de tecleo de la expresión: si `Generación` es de tipo `List <Persona>` (y por lo tanto `IEnumerable<Persona>`), entonces `h` (el nombre elegido por nosotros para la variable que irá refiriéndose sucesivamente a cada uno de los elementos de la secuencia) es de tipo `Persona`, y el sistema podrá determinar si es correcta o no cualquiera de las expresiones que aparecen en las cláusulas **`where`, `orderby`,** etc.

En la sentencia se hace uso de tres nuevas características de C# 3.0:

- **Tipos anónimos:** como la estructura del conjunto de resultados deseado no coincide con la del tipo original de los elementos, en la cláusula **`select`** de la expresión se define *ad hoc* un nuevo tipo de datos que será generado

automáticamente por el compilador. El resultado que produce la expresión de consulta es una colección de objetos de este tipo anónimo.

- Ya de por sí los tipos anónimos se apoyan en **inicializadores de objetos,** que permiten asignar los valores iniciales a los campos de los objetos del tipo anónimo.
- Por último, el objeto resultante de la ejecución de la consulta se asigna a una **variable cuyo tipo se infiere automáticamente.** Si bien en algunos casos el uso de la determinación automática del tipo de una variable local es solo una comodidad, en combinación con los tipos anónimos su uso es simplemente imprescindible. También se utiliza esta característica posteriormente al declarar la variable utilizada para recorrer la colección resultante mediante el bucle **`foreach`.**

A continuación veremos cómo por detrás del telón están entrando en funcionamiento también otras dos de las novedades de C# 3.0: las **funciones lambda** y los **métodos extensores.**

9.3. REESCRITURA DE LAS EXPRESIONES DE CONSULTA

¿Qué hace el compilador cuando encuentra una expresión de consulta? Las reglas de C# 3.0 estipulan que toda expresión de consulta que aparece en el código fuente antes de ser compilada es transformada (**reescrita**) de una manera mecánica en una secuencia de llamadas a métodos con nombres predeterminados de antemano, que reciben el nombre de **operadores de consulta** (*query operators*). Las expresiones que forman parte de cada una de las cláusulas que conforman la expresión de consulta también se reescriben para adecuarlas a los requerimientos de esos métodos predeterminados. Antes de compilarla propiamente, el compilador traducirá la expresión de consulta de nuestro ejemplo anterior en lo siguiente:

```
var mayores20  = Generación
   .Where(h => h.Edad > 20)
   .OrderBy(h => h.Nombre)
   .Select(h => new {
      Nombre = h.Nombre.ToUpper(),
      Edad = h.Edad });
```

O lo que es lo mismo:

```
var tmp1 = Generación.Where(h => h.Edad > 20);
var tmp2 = tmp1.OrderBy(h => h.Nombre);
var mayores20 = tmp2.Select(h => new {
   Nombre = h.Nombre.ToUpper(),
   Edad = h.Edad });
```

La cláusula **where** de la expresión de consulta se transforma en una llamada a un método llamado `Where()`. Para pasarla a ese método, la expresión que acompañaba a **where:**

```
h.Edad > 20
```

es transformada en una expresión lambda que a partir de una `Persona h` produce verdadero o falso:

```
h => h.Edad > 20
```

Espero que le encuentre una cierta lógica a esto: si el método `Where()` va a tener un carácter general, o sea, si va a ser capaz de funcionar para cualquier condición que un programador pudiera imaginar, lo lógico es que reciba como parámetro un **delegado** que apunte a un método booleano que compruebe la condición a cumplir. Ya hemos visto en el Capítulo 7 que las expresiones lambda son una manera más funcional de especificar delegados anónimos.

Una vez comprendido lo anterior, la siguiente pregunta es: ¿cuál es (o debe ser) la firma de este método `Where()` en el que se traduce la cláusula **where**? ¿Dónde debe estar situado? Observe que después del primer paso de reescritura de nuestra expresión de consulta de ejemplo la cosa quedaría así:

```
Generación.Where(h => h.Edad > 20)
```

Para que esa llamada sea válida, `Where()` debe ser: a) un método de instancia de la clase a la que pertenece `Generación` o b) un método de una interfaz implementada por la clase a la que pertenece `Generación` (y dado que hemos partido de que los objetos que pueden servir como origen para las consultas integradas deben implementar `IEnumerable<T>`, esa interfaz sería una buena candidata para ser extendida con métodos como `Where()`, etc.). Cualquiera de las dos vías antes mencionadas podría en principio valer, pero los creadores de C# 3.0 creyeron que llevarían a arquitecturas menos abiertas y extensibles. Es en este punto de la representación en que entran en escena los **métodos extensores** introducidos en el Capítulo 6. Teniendo los métodos extensores a nuestra disposición, `Where()`, al igual que `OrderBy()`, `Select()` y demás actores de esta obra que bien podría llamarse "La transformación de las expresiones de consulta" podrían ser también métodos extensores, que potencialmente podrían estar definidos en cualquier clase estática que esté en ámbito en el momento en que la expresión de consulta se compile.

Para comprobar la teoría anterior, haga un pequeño experimento: en el código fuente del ejemplo (que funciona correctamente en el estado en que yo se lo entrego :-) comente la línea en la parte superior del fichero que dice:

```
using System.Linq;
```

Verá que el programa deja de compilar (analice en detalle el mensaje de error: "`List<Persona>` no contiene ninguna definición de '`Where`' y no se encontró ningún método extensor llamado '`Where`' que acepte un primer argumento de tipo `List<Persona>`"). Las implementaciones predeterminadas de los operadores de consulta (conocidas como **operadores de consulta estándar**) están alojadas en una clase estática apropiadamente llamada `System.Linq.Enumerable` (en el ensamblado `System.Core.dll`, donde se almacena una buena parte de las nuevas clases incorporadas a .NET Framework 3.5).

9.4. LA (NO) SEMÁNTICA DE LOS OPERADORES DE CONSULTA

Como habrá sacado en claro, las expresiones de consulta son puro "azúcar sintáctico". En la sección anterior hemos comentado cómo estas expresiones se traducen mecánicamente, siguiendo un conjunto de reglas predefinidas en la especificación del lenguaje, en una secuencia de llamadas a métodos. También hemos visto cómo al eliminar una importación de espacio de nombres el código que contiene una expresión de consulta deja de compilar. En ese momento nos faltó enfatizar que si ponemos en ámbito **otro** espacio de nombres que contenga clases estáticas con las definiciones apropiadas de esos métodos extensores, la expresión de consulta volverá a compilar sin problemas, y utilizará en su ejecución el nuevo conjunto de métodos-operadores. En esto consiste la arquitectura abierta de LINQ: C# 3.0 no define una semántica específica para los operadores que implementan las expresiones de consulta, y cualquiera puede crear una o más clases con implementaciones a medida de los operadores de consulta para colecciones generales o específicas (o sea, abiertas o cerradas) y, poniéndola en ámbito, "enchufarla" al sistema para que sea utilizada cuando se compilen las consultas integradas sobre colecciones de esos tipos. Precisamente ésta es la vía a través de la cual se integran en el lenguaje las extensiones predefinidas de LINQ, como LINQ to XML o LINQ to SQL, y también la vía a través de la cual terceros fabricantes pueden desarrollar proveedores propietarios.

9.5. RESOLUCIÓN DE LLAMADAS A OPERADORES

Trataremos el tema de las extensiones (proveedores) de LINQ con más detenimiento en el Capítulo 11. Por el momento, veamos solo un ejemplo que nos aclare otro tema relacionado: la resolución de las llamadas. Suponga que queremos que nuestras consultas integradas sobre secuencias del tipo `Persona` se comporten de manera tal, que cuando se pregunte quiénes cumplen con una condición `P` dada se devuelvan

sólo las personas de sexo femenino que cumplen con la condición (lo cual no sería del todo correcto, pero sin duda elevaría el "coeficiente de belleza" de nuestros conjuntos de resultados :-). Podríamos definir el siguiente método extensor (no se fije mucho por ahora en lo que hace el método, fíjese fundamentalmente en su firma):

```
public static IEnumerable<Persona> Where(
   this IEnumerable<Persona> origen,
   Func<Persona, bool> filtro)
{
   IEnumerator<Persona> enm = origen.GetEnumerator();
   while (enm.MoveNext())
   {
      if (filtro(enm.Current) &&
         enm.Current.Sexo == SexoPersona.Mujer)
      {
         yield return enm.Current;
      }
   }
}
```

Una variante más concisa y 100% equivalente sería:

```
public static IEnumerable<Persona> Where(
   this IEnumerable<Persona> origen,
   Func<Persona, bool> filtro)
{
   foreach (Persona p in origen)
      if (filtro(p) && p.Sexo == SexoPersona.Mujer)
      {
         yield return p;
      }
}
```

Observe que la clase que acabamos de definir pertenece al mismo espacio de nombres que el programa, y por tanto está en ámbito (del mismo modo que `System.Linq`) durante la compilación de éste. Note también que el método `Where()` que hemos definido opera sobre un tipo genérico cerrado (`IEnumerable<Persona>`).

Si compila y ejecuta nuevamente la consulta que selecciona los mayores de 20 años, comprobará que se hará uso de nuestro método recién creado y no del operador de consulta estándar, y que por tanto en el resultado aparecerán solo chicas. Debido a que la consulta en cuestión opera sobre un objeto de tipo `IEnumerable<Persona>`, nuestro método recibe preferencia sobre el de la clase `System.Linq.Enumerable`. Ahora bien, si ejecutáramos una consulta integrada sobre una colección de otro tipo, nuestro método no sería aplicable y se utilizaría el operador estándar. ¿Y qué pasa con `OrderBy()`, `Select()` y demás? Pues que nosotros no hemos definido esos métodos, pero `IEnumerable<Persona>` es compatible con `IEnumerable<T>`

y por lo tanto se utilizarán las implementaciones de esos métodos situadas en la librería de clases base.

Una última cuestión a tener en cuenta: ¿y si hubiéramos definido el método `Where()` para que operara sobre `IEnumerable<T>`, al igual que en la implementación predeterminada? Pues se utilizaría igualmente nuestra versión, por estar situada en el mismo espacio de nombres que la clase en la que se ejecuta la consulta. El algoritmo de resolución de llamadas del compilador va buscando en los espacios de nombres de nuestro código de dentro hacia afuera, y solo si no encuentra nada por esa vía utiliza los métodos que encuentre en clases estáticas pertenecientes a otros espacios de nombres en ámbito.

9.6. LOS OPERADORES DE CONSULTA ESTÁNDAR

Bueno, ya está claro que potencialmente podemos hacer que los métodos que implementan los operadores de consulta hagan lo que nos dé la gana, siempre que cumplan con las firmas que exige el compilador; pero se supone que les asociemos una funcionalidad que cumpla con lo que en general se espera de ellos. El método `Where()`, por ejemplo, se supone que filtre la secuencia de entrada, dejando en la salida solo los elementos que satisfagan la condición especificada. `OrderBy()`, por su parte, debe recoger la secuencia de entrada y producir otra que contenga los mismos elementos que la original, pero ordenados ascendentemente según un cierto criterio.

Continuando con `Where()`, a estas alturas conocemos en su totalidad la firma del método, al menos en su sobrecarga principal. Suponiendo que se implementa como método extensor, recibe como primer argumento la secuencia de entrada (marcada con **`this`**) y como segundo parámetro un delegado a una función que recibe un `T` y devuelve un **`bool`**. El tipo del valor de retorno es `IEnumerable<T>`, como se desprende de nuestro último ejemplo de código; no es difícil darse cuenta de ello, teniendo en cuenta que el resultado que `Where()` produce servirá como entrada a `OrderBy()`, `Select()` o algún otro de los métodos en la cascada de llamadas que se obtiene como resultado de la reescritura.

La sobrecarga principal del operador de consulta estándar `Where()` está implementada así:

```
public static IEnumerable<T> Where<T>(
   this IEnumerable<T> origen, Func<T, bool> filtro)
{
   if (origen == null || filtro == null)
      throw new ArgumentNullException();
   foreach (T t in origen)
      if (filtro(t))
      {
         yield return t;
      }
}
```

El método comprueba inicialmente la validez de los argumentos de entrada. Acto seguido, entra en un bucle que recorre la secuencia de entrada, llamando para cada uno de los elementos de ésta al predicado para comprobar si el elemento cumple con la condición o no. Solo en el caso de que el elemento cumpla con la condición, lo produce en su secuencia de salida. En nuestro ejemplo de los mayores de 20 años, esa secuencia de salida será a su vez la secuencia de entrada para el operador `OrderBy()`.

Como otro ejemplo, vea cómo se implementa el operador `Select()`:

```
public static IEnumerable<V> Select<T, V>(
   this IEnumerable<T> origen, Func<T, V> selector)
{
   if (origen == null || selector == null)
      throw new ArgumentNullException();
   foreach (T t in origen)
      yield return selector(t);
}
```

Aquí el tipo de los elementos de la secuencia resultante viene dado por el tipo de retorno de la expresión de selección o transformación que se utilice.

La implementación predeterminada (en la clase `System.Linq.Enumerable` de `System.Core.dll`) de un conjunto de métodos extensores entre los que se incuyen `Where()`, `Select()`, `OrderBy()` y otros más, que pueden utilizarse para ejecutar consultas integradas contra cualquier secuencia enumerable de objetos en memoria se conoce como **LINQ to Objects,** y a esos métodos se les conoce como **operadores de consulta estándar** (*standard query operators*). Aplazaremos el análisis detallado de todos y cada uno de los operadores estándar hasta el próximo Capítulo (10), para centrarnos aquí en los fundamentos conceptuales.

9.7. EL PATRÓN DE EXPRESIONES DE CONSULTA

Más adelante veremos que no todos los operadores de consulta estándar tienen un reflejo en la sintaxis del lenguaje. Por ejemplo, existe un operador llamado `Reverse()` que produce los elementos de su secuencia de origen en orden inverso (desde el último hasta el primero). Sin embargo, no existe un "mapeado" sintáctico en el lenguaje C# para ese operador. Cuando lo necesitemos, tendremos que usarlo así:

```
int[] arr1 = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// lo siguiente produce 81, 49, 25, 9, 1
var inversa =
   (from n in arr1 where n % 2 == 1 select n * n).Reverse();
foreach(var x in inversa)
   Console.WriteLine(x);
```

Por supuesto, siempre podremos auxiliarnos de una variable intermedia:

```
var tmp =
   from n in arr1 where n % 2 == 1 select n * n;
var inversa =
   tmp.Reverse();
```

Ni siquiera todas las sobrecargas de un mismo operador de consulta estándar tienen reflejo en la sintaxis de las expresiones de consulta de C#, sino solo algunas. Por ejemplo, el operador `Where()` ofrece dos variantes, pero solo una de ellas, la que hemos presentado aquí, es la que se utiliza para la reescritura de expresiones de consulta. El subconjunto de los operadores de consulta estándar de C# 3.0 de los cuales la sintaxis de expresiones de consulta tiene una dependencia directa (y que, por tanto, cualquier fabricante de extensiones de LINQ debería soportar) da lugar a lo que se conoce como patrón de expresiones de consulta o **patrón LINQ:** una recomendación del conjunto de métodos (subconjunto del conjunto de operadores de consulta estándar) de los que se debe disponer para garantizar un soporte completo para las consultas integradas. Estudiaremos el patrón LINQ en el Capítulo 11.

9.8. LA UTILIDAD OBJECTDUMPER

Visual Studio 2008 incluye, como parte de sus ejemplos (en el directorio `Samples`) un proyecto de librería de clases llamado `ObjectDumper` (algo así como "volcador de objetos"). Esta librería incluye una clase que ofrece dos métodos estáticos que pueden sernos de gran utilidad a la hora de experimentar con las expresiones de consulta:

```
public class ObjectDumper
{
   public static void Write(object o)
   {
      // ...
   }
   public static void Write(object o, int depth)
   {
      // ...
   }
   // ...
}
```

Estos dos métodos actúan sobre cualquier objeto, obteniendo mediante reflexión e imprimiendo en la consola los tipos y valores actuales de sus propiedades. Pero si el propio objeto o alguna de sus propiedades implementa `IEnumerable<T>`, entonces los métodos se "sumergen" dentro de dicha secuencia, volcando las características de cada uno de los elementos de ésta. Este proceso se repite de manera recursiva hasta el nivel máximo de profundidad que se especifique a través del segundo parámetro.

Por ejemplo, éste será el resultado de llamar a `ObjectDumper.Write()` pasando como parámetro la variable `mayores20` del ejemplo anterior:

```
Nombre=DENNIS              Edad=23
Nombre=JENNIFER            Edad=24
```

En los ejemplos que siguen, utilizaremos la clase `ObjectDumper` a la hora de volcar en pantalla los resultados de las consultas. Para hacer uso de esta clase, basta con compilar la librería y agregar una referencia a ella en nuestros proyectos.

9.9. EJEMPLOS BÁSICOS

Partiendo del hecho de que cualquier objeto que implemente `IEnumerable<T>` puede servir como origen para una consulta integrada, y que objetos tan habituales como los arrays, las colecciones genéricas e incluso las cadenas de caracteres (lo que nos permite enumerar los caracteres que las componen) implementan esta interfaz, queda claro que podremos aplicar las consultas integradas en una gran cantidad de situaciones cotidianas para las que anteriormente utilizábamos bucles, contadores y otras técnicas varias.

Personalmente, pienso que es importante para un programador de C# 3.0 alcanzar el "nirvana LINQ": el estado en que, de forma natural, se reconoce en nuestra mente la posibilidad de utilizar una consulta integrada cada vez que ésta se presenta.

A continuación se muestran algunos ejemplos de expresiones de consulta aplicadas a cadenas de caracteres:

```
// produce en orden alfabético las vocales de la cadena 's'
var s1 = from c in s
         where "AEIOU".Contains(char.ToUpper(c))
         orderby c
         select c;
ObjectDumper.Write(s1);

// cuenta los espacios en la cadena 's'
int n1 = (from c in s
         where c == ' '
         select c).Count();
ObjectDumper.Write(n1);
```

En el segundo ejemplo, observe la llamada a `Count()` a continuación de la consulta integrada. `Count()` es otro de los operadores de consulta estándar para el que no existe una sintaxis "endulzada" (al menos en C# 3.0).

Aunque involucra cadenas de caracteres, el siguiente ya no sería un ejemplo de consulta sobre cadenas, sino sobre arrays, porque **`string`**`.Split()` devuelve un

array de cadenas de caracteres. Se trata de una expresión que produce las palabras que forman parte de una oración, en minúsculas y ordenadas alfabéticamente:

```
// palabras diferentes en una oración
var n2 = (from w in s.Split(new char[] { ' ', '\t', '\n' },
            StringSplitOptions.RemoveEmptyEntries)
          orderby w.ToLower()
          select w.ToLower()).Distinct();
ObjectDumper.Write(n2);
```

Para eliminar los duplicados, nuevamente tenemos que recurrir a un operador de consulta que solo puede utilizarse con sintaxis funcional, `Distinct()`.

Los arrays son otro "caldo de cultivo" impresionante para las consultas integradas:

```
// produce una secuencia con los pares en arr1
var pares = from n in arr1
            where n % 2 == 0
            select n;
// también pudo ser:
var pares2 = arr1.Where(n => n % 2 == 0);

// produce la suma de los números de la secuencia
int suma = arr1.Sum();
// lo mismo que:
int suma2 = (from n in arr1 select n).Sum();

// produce los números de la secuencia, incrementados en 1
var otra  = from n in arr1
            select n + 1;
// lo mismo que:
var otra2 = arr1.Select(n => n + 1);

string[] ciudades = { "HAVANA", "MADRID", "NEW YORK",
                      "MIAMI", "SEATTLE" };
// ciudades con más de seis letras,
// en orden alfabético
var c7 = from c in ciudades
         where c.Length > 6
         orderby c
         select c;

var meses = new TipoMeses[] {
   new TipoMeses { Nombre = "Enero", CantidadDias = 31 },
```

Continúa

```
    new TipoMeses { Nombre = "Febrero", CantidadDias = 28 },
    new TipoMeses { Nombre = "Marzo", CantidadDias = 31 },
    new TipoMeses { Nombre = "Abril", CantidadDias = 31 },
    new TipoMeses { Nombre = "Mayo", CantidadDias = 31 },
    new TipoMeses { Nombre = "Junio", CantidadDias = 31 },
    new TipoMeses { Nombre = "Julio", CantidadDias = 31 },
    new TipoMeses { Nombre = "Agosto", CantidadDias = 31 },
    new TipoMeses { Nombre = "Septiembre", CantidadDias = 31 },
    new TipoMeses { Nombre = "Octubre", CantidadDias = 31 },
    new TipoMeses { Nombre = "Noviembre", CantidadDias = 31 },
    new TipoMeses { Nombre = "Diciembre", CantidadDias = 31 },
};

// meses con 31 días
var meses31 = from m in meses
              where m.CantidadDias == 31
              select m.Nombre;
```

Observe de los primeros ejemplos que en los casos triviales (si solo se desea filtrar o transformar la secuencia) las consultas integradas pueden expresarse fácilmente utilizando directamente la sintaxis de métodos. Esto por supuesto ya no ocurre con las consultas más complejas, sobre todo si éstas incluyen múltiples generadores, encuentros o grupos, como las que veremos a continuación.

Por último, veamos algunos ejemplos sobre la lista de personas:

```
// nombres que empiezan por 'D'
var hijos = from h in Generación
            where h.Nombre.StartsWith('D')
            select h.Nombre;
ObjectDumper.Write(hijos);

// lista ordenada, primero por sexos
// luego por edad en orden ascendente
var orden = from h in Generación
            orderby h.Sexo, h.Edad descending
            select h;
ObjectDumper.Write(orden);
```

9.10. DE NUEVO LA EJECUCIÓN DIFERIDA

Recordará del Capítulo 3 la demostración de cómo una llamada a un método cuyo cuerpo es un bloque de iteración (como es el caso de `Where()` y de la gran mayoría de los operadores de consulta estándar) no da lugar a iteración alguna, sino que

únicamente construye el objeto enumerador. Este hecho da lugar a la particularidad de que la asignación de una expresión de consulta como:

```
var mayores20  = from h in Generación
                 where h.Edad > 20
                 orderby h.Nombre
                 select new {
                   Nombre = h.Nombre.ToUpper(),
                   Edad = h.Edad };
```

no hace más que preparar los objetos enumeradores necesarios; el resultado de una consulta no se obtendrá realmente hasta el momento en que se produzca la iteración sobre ella. Esta evaluación **bajo demanda** o **diferida** constituye el comportamiento por defecto de LINQ, y es, en la mayor parte de los casos, la opción más conveniente. No obstante, puede que en cierto momento se desee "cachear" completamente en memoria el resultado de una consulta para su posterior reutilización. Con este objetivo se ofrecen los operadores estándar `ToArray<T>()`, `ToList<T>()`, `ToDictionary<T,K>()` y `ToLookup<T,K,E>()`, que estudiaremos en el próximo capítulo.

Un efecto colateral que puede producirse y debemos tener en cuenta es la posibilidad de que dos evaluaciones sucesivas de una misma consulta produzcan resultados diferentes si entre medias se producen cambios en la fuente de información subyacente. Por ejemplo, dada la consulta:

```
// consulta: nombres que empiezan por 'D'
var hijos =  from h in Generación
             where h.Nombre.StartsWith("D")
             select h.Nombre;
```

Si entre dos recorridos de los resultados se produce una modificación en la colección, los resultados serán diferentes:

```
Console.WriteLine("Primera evaluación");
ObjectDumper.Write(hijos);

// se modifica la colección subyacente
// se cambia el nombre de "Diana" a "Elisa"
Generación[0].Nombre = "Elisa";

Console.WriteLine("Segunda evaluación");
ObjectDumper.Write(hijos);
```

Para evitar ese efecto, podemos utilizar cualquiera de los operadores estándar antes mencionados:

```
// se evalúa y "cachea" la consulta
var hijos_cache = hijos.ToList();

Console.WriteLine("Primera evaluación");
ObjectDumper.Write(hijos_cache);

// se modifica la colección subyacente
Generación[0].Nombre = "Diana";

Console.WriteLine("Segunda evaluación");
ObjectDumper.Write(hijos_cache);
```

9.11. SINTAXIS DE LAS EXPRESIONES DE CONSULTA

La sintaxis completa de las expresiones de consulta es la siguiente:

```
<expr. consulta> ::= <cláusula from> <cuerpo consulta>

<cláusula from> ::=
   from <elemento> in <expr.origen>

<cuerpo consulta> ::=
   <cláusula cuerpo consulta>*
   <cláusula final consulta>
   <continuación>?

<cláusula cuerpo consulta> ::=
   (<cláusula from>
   | <cláusula join>
   | <cláusula join-into>
   | <cláusula let>
   | <cláusula where>
   | <cláusula orderby>)

<cláusula let> ::=
   let <elemento> = <expr.selección>

<cláusula where> ::=
   where <expr.filtro>

<cláusula join> ::=
   join <elemento> in <expr.origen>
   on <expr.clave> equals <expr.clave>
```

Continúa

```
<cláusula join-into> ::=
   join <elemento> in <expr.origen>
   on <expr.clave> equals <expr.clave>
   into <elemento>

<cláusula orderby> ::=
   orderby <ordenaciones>

<ordenaciones> ::=
   <ordenacion>
   | <ordenaciones> , <ordenación>

<ordenación> :=
   <expr.clave> (ascending | descending)?

<cláusula final consulta> ::=
   (<cláusula select> | <cláusula groupby>)

<cláusula select> ::=
   select <expr.selección>

<cláusula groupby> ::=
   group <expr.selección> by <expr.clave>

<continuación> ::=
   into <elemento> <cuerpo consulta>

Meta-lenguaje:
   * - cero o más veces
   ( ... | ... ) - alternativa
   ? - elemento opcional
```

Básicamente, una expresión de consulta siempre comienza por una cláusula **`from`**, en la que especifica la colección de origen sobre la que se ejecutará la consulta. A continuación pueden venir una o más cláusulas **`from`**, **`join`**, **`let`**, **`where`** u **`orderby`**, para terminar con una sentencia **`select`** o **`group by`**. Opcionalmente, al final de la expresión puede aparecer una cláusula de continuación, que comienza con la palabra reservada **`into`** y va seguida del cuerpo de otra consulta. Recuerde que todas las palabras clave que se utilizan aquí son contextuales – tienen un significado especial solo dentro de las expresiones de consulta.

En las próximas secciones iremos estudiando los elementos sintácticos de las expresiones de consulta que aún nos resta por presentar.

9.12. PRODUCTOS CARTESIANOS

En el cálculo relacional, el **producto cartesiano** de dos tablas no es más que el conjunto de filas resultante de combinar cada fila de la primera tabla con cada fila de la segunda. El mismo concepto se puede aplicar aquí a la combinación de dos

secuencias: si se colocan dos cláusulas **`from`** (que actúan como generadoras) seguidas, para cada elemento de la primera secuencia se producirán todos los elementos de la segunda. Por ejemplo, la consulta:

```
/* PRODUCTO CARTESIANO */
var pc1 = from pa in Paises
          from pe in Generación
          select new { pa.Nombre, Nombre2 = pe.Nombre };
ObjectDumper.Write(pc1);
```

produce como resultado:

```
Nombre=ESPAÑA       Nombre2=Diana
Nombre=ESPAÑA       Nombre2=Dennis
Nombre=ESPAÑA       Nombre2=Claudia
Nombre=ESPAÑA       Nombre2=Jennifer
Nombre=CUBA         Nombre2=Diana
Nombre=CUBA         Nombre2=Dennis
Nombre=CUBA         Nombre2=Claudia
Nombre=CUBA         Nombre2=Jennifer
Nombre=RUSIA        Nombre2=Diana
Nombre=RUSIA        Nombre2=Dennis
Nombre=RUSIA        Nombre2=Claudia
Nombre=RUSIA        Nombre2=Jennifer
Nombre=ESTADOS UNIDOS     Nombre2=Diana
Nombre=ESTADOS UNIDOS     Nombre2=Dennis
Nombre=ESTADOS UNIDOS     Nombre2=Claudia
Nombre=ESTADOS UNIDOS     Nombre2=Jennifer
```

Como nota colateral, observe cómo ha sido necesario asignarle otro nombre a la segunda propiedad del tipo anónimo resultante; si no se especifica otra cosa, el compilador asigna a las propiedades del tipo anónimo los mismos nombres de las propiedades o campos originales, pero no admite duplicidades.

Los productos cartesianos se implementan mediante llamadas a un operador de consulta estándar llamado `SelectMany()`, que es el encargado de producir una secuencia en la que se combina cada uno de los elementos de la primera secuencia con cada uno de los elementos de la segunda. Veremos con más detalle este operador en el siguiente capítulo.

Si ha trabajado antes con SQL, tendrá claro el principal peligro de los productos cartesianos: la explosión combinatoria de resultados que pueden provocar. Si `N` es la cardinalidad (cantidad de elementos) de la primera secuencia, y `M` la de la segunda, la cantidad de elementos de la secuencia resultado será igual a `N * M`. Una tercera secuencia en la ecuación elevaría mucho más la cantidad de combinaciones, y así sucesivamente. Por ello en general es recomendable evitar los productos cartesianos

siempre que sea posible, posiblemente aplicando alguna de las técnicas que se describen a continuación.

9.12.1. Restricción de productos y optimización de consultas

Un tipo de producto cartesiano relativamente frecuente es el **auto-producto**, en el que combina una secuencia consigo misma. Por ejemplo, suponga que queremos obtener una lista de parejas de elementos de la lista `Generación` en las que el primer elemento es un chico y el segundo una chica. Un primer intento podría ser:

```
var pc2 = from p1 in Generación
          from p2 in Generación
          where   p1.Sexo == SexoPersona.Varón &&
                  p2.Sexo == SexoPersona.Mujer
          select new { El = p1.Nombre, Ella = p2.Nombre };
ObjectDumper.Write(pc2);
```

El anterior es un ejemplo de lo que podríamos llamar **producto cartesiano restringido:** un producto cartesiano al que se adjuntan condiciones de filtro que reducen el tamaño de la secuencia resultante.

Si analiza con detenimiento la consulta anterior, le parecerá bastante obvio que la siguiente es una mejor opción en relación con el rendimiento, porque los elementos de la primera secuencia que a fin de cuentas no van a servir son eliminados antes en la "tubería" de operadores de consulta que se ejecutan:

```
var pc3 = from p1 in Generación
          where p1.Sexo == SexoPersona.Varón
          from p2 in Generación
          where p2.Sexo == SexoPersona.Mujer
          select new { El = p1.Nombre, Ella = p2.Nombre };
ObjectDumper.Write(pc3);
```

Aunque el estudio de técnicas específicas de optimización de consultas LINQ queda fuera de los objetivos de este libro, no está de más comentar este hecho para que lo tenga en cuenta al programar sus consultas integradas. Otra cuestión totalmente diferente es que un compilador "inteligente" podría transformar de forma transparente al programador la primera expresión en la segunda; seguramente futuras versiones del compilador de C# lo harán, pero no la actual.

9.13. ENCUENTROS

Los encuentros son otra de las construcciones típicas de los lenguajes relacionales como SQL que han sido incorporadas a las expresiones de consulta de C# 3.0. Un

encuentro es básicamente un producto cartesiano sobre dos secuencias limitado a las tuplas `(t1, t2)` en la que el valor de cierta expresión aplicada al elemento de la primera secuencia `t1` **es igual** al valor de otra expresión aplicada al elemento de la segunda tabla `t2`. Básicamente, se trata de limitar seriamente las combinaciones que produciría un producto cartesiano completo, manteniendo únicamente los elementos de las secuencias que "casan" de acuerdo con cierto criterio compartido.

Un ejemplo cercano al clásico, pero aplicado a nuestra lista `Generación` sería el siguiente: suponga que deseamos obtener una secuencia con los nombres de los chicos(as) y los nombres de sus países de nacimiento. En vez de restringir el producto cartesiano, de manera similar a como hemos hecho hace un momento, en este caso haremos lo siguiente:

```
/* ENCUENTRO */
var enc1  = from pa in Paises
            join pe in Generación
              on pa.Codigo equals pe.CodigoPaisNac
            orderby pa.Nombre
            select new { pa.Nombre, Nombre2 = pe.Nombre };
ObjectDumper.Write(enc1);
```

Además de "cruzar" las listas `Paises` y `Generación`, hemos ordenado el resultado por países, "apuntando" hacia un futuro ejemplo con agrupación. Como cabría esperar, el resultado será el siguiente:

```
Nombre=CUBA         Nombre2=Claudia
Nombre=CUBA         Nombre2=Jennifer
Nombre=ESPAÑA       Nombre2=Diana
Nombre=RUSIA        Nombre2=Dennis
```

9.13.1. Particularidades sintácticas

La condición de encuentro entre las dos secuencias consiste en un selector de clave para la secuencia externa, la palabra clave contextual **`equals`** y otro selector de clave para la secuencia interna. Los selectores de claves que se utilizan para la comparación de campos pueden ser cualquier expresión que se obtenga a partir del identificador que representa al elemento de la secuencia correspondiente.

Si viene usted del mundo SQL, tenga en cuenta que el operador `Join()`, en el que se traduce la palabra clave contextual **`join`**, se corresponde con el **encuentro interno (natural) basado en la igualdad de claves** (*inner equijoin*), en el que se valida la igualdad entre los valores de los dos selectores de claves. Se tomó la decisión de utilizar la palabra clave **`equals`** en lugar del símbolo == para evitar que por asociación mental un programador con conocimientos de SQL pudiera creer

que son válidos otros criterios de encuentro como >= ó <= (que en la práctica se utilizan muy poco). Con relación a los encuentros externos, más adelante en este capítulo veremos vías para simularlos en LINQ.

9.13.2. Diferencia con el producto cartesiano restringido

Como ya hemos mencionado, pudimos haber obtenido el mismo resultado de antes ejecutando la siguiente consulta, basada en un producto cartesiano restringido con un **`where`:**

```
var pc4 = from pa in Paises
          from pe in Generación
          where pa.Codigo == pe.CodigoPaisNac
          select new { pa.Nombre, Nombre2 = pe.Nombre };
ObjectDumper.Write(pc2);
```

La diferencia entre ambas variantes está en el rendimiento superior del encuentro. El método extensor `Join()`se orienta a la utilización de una tabla hash para intentar obtener un mejor rendimiento, de un modo similar a como se utilizan los índices de las tablas en el mundo de las bases de datos. Cuando la secuencia externa debe ser enumerada, el método `Join()` recorre la secuencia interna, recopilando los elementos según su clave de comparación en una tabla hash. Una vez que esa tabla hash ha sido creada, se recorre la secuencia externa, utilizando la tabla hash para buscar rápidamente las coincidencias de claves, y produciendo un elemento de salida para cada coincidencia.

9.14. GRUPOS

La sintaxis de las expresiones de consulta ofrece también soporte para la organización de los elementos de una secuencia en grupos según los diferentes valores de una clave de agrupación que se calcula para cada uno de los elementos. Por ejemplo, la siguiente sentencia:

```
var gruposSexo =
   from h in Generación
   group new { h.Nombre, h.Edad } by h.Sexo;
ObjectDumper.Write(gruposSexo, 2);
```

agrupa los elementos de la secuencia original según los diferentes valores de la expresión `h.Sexo`. En este ejemplo, se obtendrá una secuencia de dos elementos, que serán a su vez secuencias: la primera con objetos de un tipo anónimo que incluye los datos solicitados (nombre y edad) de todas las chicas (objetos para los que el valor de la clave de agrupación es `SexoPersona.Mujer`); la segunda, con los datos de todos los chicos, para los que el valor de `h.Sexo` es igual a `SexoPersona.Varón`.

C# 3.0 traduce la expresión de consulta anterior en una llamada al operador estándar `GroupBy()`:

```
var gruposSexo2 =
   Generación.GroupBy(
      h => h.Sexo,
      h => new { h.Nombre, h.Edad });
```

El resultado de la llamada a `GroupBy()` es una secuencia cada uno de cuyos elementos son a su vez otras secuencias internas asociadas a cada grupo, que implementan una interfaz llamada `IGrouping<TKey, TElement>`, heredera de `IEnumerable<TElement>`. Básicamente, esta interfaz añade una propiedad de solo lectura `Key`, del tipo de la clave de agrupación. El siguiente bucle muestra la estructura del resultado de la consulta:

```
foreach (var hh in gruposSexo)
{
   // la clave del grupo
   Console.WriteLine(hh.Key);
   // los elementos del grupo
   foreach (var hhh in hh)
      Console.WriteLine (" - " + hhh.Nombre +
                         " (" + hhh.Edad + ")");
}
```

El ejemplo que se presenta a continuación involucra nuestras dos "tablas" de personas y países. Si quisiéramos agrupar los miembros de `Generación` según sus países de nacimiento, la siguiente sentencia cumpliría con ese fin:

```
Console.WriteLine("GRUPOS POR PAISES");
var gruposPaises =
   from pa in Paises
   join pe in Generación on pa.Codigo equals pe.CodigoPaisNac
   group new { pe.Nombre, pe.Edad } by pa.Nombre;

foreach (var hh in gruposPaises)
{
   // el valor de la clave
   Console.WriteLine(hh.Key);
   // los elementos del grupo
   foreach (var hhh in hh)
      Console.WriteLine(" - " + hhh.Nombre +
                        " (" + hhh.Edad + ")");
}
```

Observe la similitud entre esta última expresión de consulta y la que presentamos en la sección dedicada a los encuentros; la diferencia está en la presencia de la cláusula **group...by** en lugar de **select.** Precisamente estas dos cláusulas son las "cláusulas finales" en la sintaxis de las expresiones de consulta. ¿Y si quisiéramos extender una consulta con alguna cláusula más? Por ejemplo, si ejecuta la agrupación anterior, verá (la mecánica de funcionamiento del método `GroupBy()` la estudiaremos en el próximo capítulo) que los diferentes grupos aparecen en la secuencia resultante en el mismo orden en que aparecen los países en la secuencia original. ¿Y si quisiéramos obtener los grupos en orden alfabético de los países? Podríamos acudir a la sintaxis explícita:

```
var gruposPaises2 =
   (from pa in Paises
    join pe in Generación on pa.Codigo equals pe.CodigoPaisNac
    group new { pe.Nombre, pe.Edad } by pa.Nombre).
   OrderBy(g => g.Key);
```

Por cierto, para ordenar descendentemente según la cantidad de personas de cada país, sería así (los grupos son a su vez secuencias):

```
var gruposPaises3 =
   (from pa in Paises
    join pe in Generación on pa.Codigo equals pe.CodigoPaisNac
    group new { pe.Nombre, pe.Edad } by pa.Nombre).
   OrderByDescending(g => g.Count());
```

Para expresar este tipo de situaciones el lenguaje prevee un mecanismo mejor: las continuaciones, que examinaremos en la próxima sección.

9.15. LA CLÁUSULA INTO

Si analiza la sintaxis de las expresiones de consulta presentada anteriormente, verá que la palabra reservada contextual **into** se utiliza con dos fines diferentes en las expresiones de consulta. Una utilización más sencilla hace posible implementar las **continuaciones;** mientras que un segundo uso, mucho más potente, permite expresar los **encuentros agrupados.** En esta sección se presentan ambas aplicaciones de la cláusula **into.**

9.15.1. Continuaciones

Frecuentemente es conveniente tratar los resultados de una consulta como secuencia de entrada para una consulta subsiguiente. La cláusula **into** permite precisamente ejecutar en cascada dos expresiones de consulta, utilizando como generador para la segunda consulta el resultado de la primera. En C# 3.0 esto se conoce como una

continuación de consulta (*query continuation*). Por ejemplo, los grupos ordenados alfabéticamente de la sección anterior se podrían haber obtenido así:

```
var gruposPaises4 =
   from pa in Paises
   join pe in Generación on pa.Codigo equals pe.CodigoPaisNac
   group new { pe.Nombre, pe.Edad } by pa.Nombre
   into tmp
   orderby tmp.Key
   select tmp;
```

En la práctica, las continuaciones son especialmente útiles para procesar los resultados producidos por una cláusula **`group...by`.** Observe este otro ejemplo relacionado con el anterior:

```
var resumenPaises =
   from pa in Paises
   join pe in Generación on pa.Codigo equals pe.CodigoPaisNac
   group new { pe.Nombre, pe.Edad } by pa.Nombre
   into tmp
   orderby tmp.Count() descending
   select new { Pais = tmp.Key, Cantidad = tmp.Count() };
```

que produce una secuencia de objetos con dos propiedades: el nombre del país y la cantidad de miembros de `Generación` nacidos en ese país.

Un volcado del resultado de esta consulta nos mostrará:

```
Pais=CUBA          Cantidad=2
Pais=ESPAÑA        Cantidad=1
Pais=RUSIA         Cantidad=1
```

Otro ejemplo sencillo de continuación sería la siguiente emulación poco eficiente de un encuentro:

```
var into1   =from h in Generación
             orderby h.Nombre
             select new
                {
                   Nombre = h.Nombre,
                   CodPais = h.CodigoPaisNac
                }
             into tmp
             from p in Paises
             where tmp.CodPais == p.Codigo
             select tmp.Nombre + " - " + p.Nombre;
```

La variable `tmp` representa a los elementos resultantes de la ejecución de la primera consulta, y como tal puede utilizarse en el cuerpo de la segunda. Efectivamente, esta consulta es equivalente a la siguiente:

```
var into2 = from tmp in
            (
            from h in Generación
            orderby h.Nombre
            select new
               {
                  Nombre = h.Nombre,
                  CodPais = h.CodigoPaisNac
               }
            )
            from p in Paises
            where tmp.CodPais == p.Codigo
            select tmp.Nombre + " - " + p.Nombre;
```

En esta nueva versión, los paréntesis alrededor de la consulta interna no son imprescindibles; se han introducido únicamente en aras de una mejor comprensión.

9.15.2. Encuentros agrupados

La segunda y más importante aplicación de la cláusula **`into`** tiene como objetivo implementar lo que se conoce como **encuentros agrupados** (*grouped joins).* Se trata de un tipo de encuentro que no tiene equivalente directo en el mundo de las bases de datos relacionales, y que en lugar de producir la típica secuencia de parejas de un encuentro normal produce una secuencia en la que cada elemento de la secuencia externa se aparea con el **grupo de elementos** de la secuencia interna cuyos valores de clave de comparación coinciden con el valor de la clave de comparación del elemento de la secuencia externa.

Una construcción **`join`**...**`into`** se traduce en una llamada al operador estándar `GroupJoin()`, que se basa, al igual que `Join()`, en el uso de una tabla hash. Cuando la secuencia de entrada (externa) es enumerada, el método `GroupJoin()` enumera la secuencia interna, recopilando los elementos según su clave de comparación en una tabla hash. Una vez que esa tabla hash ha sido rellenada, se enumera la secuencia externa, utilizando la tabla para buscar rápidamente los elementos coincidentes de la secuencia externa, y produciendo para cada elemento de la secuencia externa un único elemento de salida, formado a partir del elemento original y la secuencia (posiblemente vacía) de todos los elementos de la secuencia interna cuyas claves de comparación coinciden con la de aquél.

Por ejemplo, una manera más concisa de haber obtenido una lista de países con las cantidades de personas de ese país similar a la del epígrafe anterior habría sido:

```
var resumenPaises2 =
   from pa in Paises
   orderby pa.Nombre
   join pe in Generación on pa.Codigo equals pe.CodigoPaisNac
   into gp
   select new { Pais = pa.Nombre, Cantidad = gp.Count() };
```

La traducción de esta expresión de consulta sería:

```
var resumenPaises3 =
   Paises.OrderBy(pa => pa.Nombre).
   GroupJoin(Generación,
      pa => pa.Codigo,
      pe => pe.CodigoPaisNac,
      (p, gp) => new { Pais = p.Nombre,
                       Cantidad = gp.Count() });
```

y el resultado:

```
Pais=CUBA                  Cantidad=2
Pais=ESPAÑA                Cantidad=1
Pais=ESTADOS UNIDOS        Cantidad=0
Pais=RUSIA                 Cantidad=1
```

Observe la diferencia con el resultado de la consulta del epígrafe anterior: en este caso se incluyen también los países en los cuales no ha nacido nadie.

9.15.3. Emulando encuentros externos con encuentros agrupados

En el mundo de las bases de datos relacionales, un **encuentro externo por la izquierda** de dos tablas es un encuentro cuyo resultado contiene **todas** las filas de la tabla de la izquierda. En caso de que cierta fila no tenga "compañera" en la tabla de la derecha, se le incluirá en el conjunto de resultados, rellenando la fila del resultado con valores nulos en las columnas que provengan de la tabla de la derecha. De manera simétrica, los encuentros externos por la derecha contienen todas las filas de la tabla de la derecha.

Por su manera de trabajar, que hemos descrito en el epígrafe anterior y sobre la que profundizaremos en el próximo, los encuentros agrupados son, como puede intuirse de nuestro último ejemplo, claves para la implementación de este tipo de encuentros en LINQ.

Suponga que queremos producir un encuentro externo por la izquierda entre los países y los chicos según los países de nacimiento de éstos. Por ejemplo, querríamos obtener pares (`nombre país, nombre persona`). Como no hay nadie que haya

nacido en los Estados Unidos, éste país deberá aparecer en la secuencia resultante acompañado de un valor nulo. La siguiente sentencia cumple con este fin:

```
var outerJoin1 =
   from pa in Paises
   join pe in Generación on pa.Codigo equals pe.CodigoPaisNac
   into tmp
   from res in tmp.DefaultIfEmpty()
   select new {
      Pais = pa.Nombre,
      Nombre = res == null ? null : res.Nombre
   };
ObjectDumper.Write(outerJoin1);
```

La clave de esta solución reside en la utilización de un producto cartesiano (que se traduce en una llamada a `SelectMany()`) sobre el resultado del encuentro agrupado y otra versión de él mismo al que previamente se le ha aplicado el operador estándar `DefaultIfEmpty()`. Este operador deja su secuencia de entrada estándar intacta en caso de que no sea vacía, o produce una secuencia de un único elemento (el valor por defecto del tipo de los elementos de la secuencia, que es **`null`** para los tipos referencia) en caso contrario. En el próximo capítulo se examina con más detalle el comportamiento de estos operadores.

¿Y qué hay de los encuentros externos por la derecha? Bueno, la noción de izquierda o derecha es relativa. Si nos interesara obtener un encuentro en el que aparecieran todas las personas, incluso aunque no tuvieran un país asociado, podríamos cambiar el orden de los operandos del encuentro en la consulta anterior:

```
// ENCUENTRO EXTERNO "POR LA DERECHA"
var outerJoin2 =
   from pe in Generación
   join pa in Paises on pe.CodigoPaisNac equals pa.Codigo
   into tmp
   from res in tmp.DefaultIfEmpty()
   select new
      {
         Pais = res == null ? null : res.Nombre,
         Nombre = pe.Nombre
      };
ObjectDumper.Write(outerJoin2);
```

Pruebe a incorporar a un "John Doe" a la lista de personas y verá que aparecerá en el resultado aún cuando no tenga un país válido asignado.

Algo que sí hay que acometer es la simulación de los **encuentros externos simétricos,** en los que se combinan los comportamientos de los encuentros externos por la izquierda y por la derecha. Aquí simplemente mostraremos una implementación básica:

```
// ENCUENTRO EXTERNO SIMETRICO
var outerJoin3 =
   outerJoin1.Union(outerJoin2);
ObjectDumper.Write(outerJoin3);
```

El operador de consulta estándar `Union()`, que se describe en el siguiente capítulo, realiza una unión conjuntual de los elementos de dos secuencias. Tenga en cuenta que para que esta sentencia compile, el tipo (anónimo) de los elementos de las secuencias que se asignarán a las variables `outerJoin1` y `outerJoin2` debe ser el mismo; como recordará del Capítulo 5, para ello los nombres de las propiedades deben coincidir, y éstas deben aparecer en el mismo orden.

9.16. LA CLÁUSULA LET

Casi que por accidente, la cláusula **`let`** ha quedado relegada al último puesto. Esta cláusula sirve como un mecanismo de conveniencia muy útil en los casos en que necesitamos asignar un nombre a un resultado intermedio para aprovecharlo más adelante o ejecutar una consulta "subsidiaria" (subconsulta) cuyo resultado necesitamos dentro de otra consulta.

Por ejemplo, suponga que tenemos una secuencia de números enteros, y queremos agruparlos según su última cifra. Para cada número de la secuencia necesitaremos obtener esa última cifra, dado que es en base a éste que se va a agrupar. Podríamos hacerlo así:

```
var grupos  = from n in numeros
              let terminacion = n % 10
              orderby terminacion
              group n by terminacion;

foreach (var g in grupos)
{
   Console.WriteLine(g.Key);
   foreach (var elem in g)
      Console.WriteLine("    " + elem);
}
```

El compilador traduce una cláusula **`let`** en una llamada al operador `Select()` que incorpora la variable a los elementos de la secuencia de salida como una nueva propiedad adicional. La sentencia anterior se traducirá a:

```
var grupos2 = numeros.
   Select(n => new { n, terminacion = n % 10 }).
   OrderBy(x => x.terminacion).
   GroupBy(x => x.terminacion, x => x.n);
```

Como ejemplo de utilización de **let** para expresar subconsultas, suponga que queremos obtener las personas de Generación cuya edad es mayor o igual que la media de edad del conjunto. En principio, necesitaríamos ejecutar dos consultas: una inicial para calcular la media, y luego una segunda para obtener los elementos cuya edad supere esa media anteriormente calculada. Con la ayuda de **let,** podríamos expresarlo todo de una vez así:

```
var let1  = from p in Generación
            let media = (Generación.Average(pp => pp.Edad))
            where p.Edad >= media
            select p.Nombre;
ObjectDumper.Write(let1);
```

Si quisiéramos obtener la lista de aquellos que superan la media de edad de las personas de su país, podríamos haberlo expresado así:

```
var let2  = from p in Generación
            let media =
               (from p2 in Generación
               where p2.CodigoPaisNac == p.CodigoPaisNac
               select p2).Average(pp => pp.Edad)
            where p.Edad >= media
            select p.Nombre;
ObjectDumper.Write(let2);
```

9.17. ALGUNOS EJEMPLOS PRÁCTICOS

En la programación diaria es común encontrar situaciones en las que se necesita iterar sobre los elementos de una colección o array devuelto por algún método de la librería de clases de .NET Framework para filtrar, ordenar o transformar sus elementos. Esta sección tiene como objetivo presentar unos pocos ejemplos tomados de la programación cotidiana en los que la utilización de las expresiones de consulta es directa e inmediata. Numerosas áreas de la librería de clases vienen a la mente, y con toda seguridad el lector podrá identificar muchas más situaciones similares:

- Enumeración de procesos o hilos de un proceso.
- Navegación por la Bitácora del sistema.
- Navegación por el sistema de ficheros local.
- Enumeración de unidades de red, recursos compartidos, etc.
- Enumeración de usuarios y grupos del Directorio Activo.
- Enumeración de características de ensamblados o clases mediante reflexión.

A continuación se muestra un fragmento de código en el que se utiliza una expresión de consulta para listar en orden alfabético los procesos "reales" que han estado activos en el sistema durante más de una hora:

```
var procesos =
   from p in System.Diagnostics.Process.GetProcesses()
   where p.ProcessName != "Idle" &&
      DateTime.Now - p.StartTime > new TimeSpan(1, 0, 0)
   orderby p.ProcessName
   select p;
ObjectDumper.Write(procesos);
```

En este otro ejemplo se presentan, agrupadas según su tipo, todas las propiedades públicas de un tipo de datos `tipo`, obtenidas mediante reflexión.

```
BindingFlags flags = BindingFlags.DeclaredOnly |
   BindingFlags.Public |
   BindingFlags.Instance | BindingFlags.Static;
var resumen =
   from prop in tipo.GetProperties(flags)
   orderby prop.PropertyType.FullName, prop.Name
   group new {
      Tipo = prop.PropertyType.FullName,
      Nombre = prop.Name,
      Categoría = (prop.GetGetMethod().IsStatic ?
           "Estática" : "De instancia")
        } by prop.PropertyType.FullName;
ObjectDumper.Write(resumen, 2);
```

Para la estructura `System.TimeSpan` se obtendría lo siguiente:

```
...
   Tipo=System.Double  Nombre=TotalDays    Categoría=De instancia
   Tipo=System.Double  Nombre=TotalHours   Categoría=De instancia
   Tipo=System.Double  Nombre=TotalMilliseconds  Categoría=De instancia

   Tipo=System.Double  Nombre=TotalMinutes Categoría=De instancia
   Tipo=System.Double  Nombre=TotalSeconds Categoría=De instancia
...
   Tipo=System.Int32   Nombre=Days         Categoría=De instancia
   Tipo=System.Int32   Nombre=Hours        Categoría=De instancia
   Tipo=System.Int32   Nombre=Milliseconds Categoría=De instancia
   Tipo=System.Int32   Nombre=Minutes      Categoría=De instancia
   Tipo=System.Int32   Nombre=Seconds      Categoría=De instancia
...
   Tipo=System.Int64   Nombre=Ticks        Categoría=De instancia
...
```

Como último ejemplo, suponga que necesita listar los diez primeros ficheros con extensión `.jpg` disponibles en un directorio dado y sus subdirectorios. La aproximación inicial podría ser:

```
string[] ficheros = Directory.GetFiles(
   " C:\\Users\\Octavio\\Pictures", "*.jpg",
   SearchOption.AllDirectories);
IEnumerable<FileInfo> datosFicheros =
   (from f in ficheros select new FileInfo(f)).Take(10);
ObjectDumper.Write(datosFicheros);
```

`Take()` produce los `n` primeros elementos de una secuencia, y es otro de los operadores de consulta estándar que no tiene representación en la sintaxis de C# 3.0.

Si recuerda la clase `FileSystemEnumerator` presentada en el Capítulo 3, entonces no habrá que repetirle que esta otra versión ofrece un rendimiento muy superior:

```
using (FileSystemEnumerator fse = new FileSystemEnumerator(
   " C:\\Users\\Octavio\\Pictures", "*.jpg", true))
{
   IEnumerable<FileInfo> datosFicheros2 =
      (from f in fse.Matches() select f).Take(10);
ObjectDumper.Write(datosFicheros2);
}
```

9.18. ¿UN NUEVO MODELO DE ESCRITURA DE BUCLES?

Hace relativamente poco tiempo leí un artículo en un blog en el que se intentaba mostrar cómo es posible utilizar las consultas integradas para la resolución de problemas combinatorios. Concretamente, en el artículo se resolvía un problema de distribución de pesas en un sistema de balanzas utilizando un enfoque de fuerza bruta: una serie de bucles anidados en los que se itera sobre todos los posibles valores de cada una de las variables del problema. La "aplicación de LINQ" consistía en la utilización del operador de consulta estándar `Range(min, max)`, que produce secuencialmente los valores enteros entre `min` y `max`, ambos inclusive, para modelar un bucle de tipo **`for`.**

El problema en cuestión me recordó ciertos ejercicios que poníamos hace 20 años en la Universidad a los estudiantes de la asignatura "Introducción a la Programación" (bajo la inestimable dirección del Dr. Miguel Katrib). Así que decidí programar uno de estos ejercicios por las dos vías: mediante "bucles tradicionales" y "bucles LINQ", para comparar ambas implementaciones.

Enunciado: Escriba un programa que determine todos los números de cuatro cifras que son iguales a la suma de las cuartas potencias de cada una de las cifras que lo componen. Por ejemplo, uno de esos números es 1634:

$$1634 = 1^4 + 6^4 + 3^4 + 4^4$$

Solución de fuerza bruta: Iteramos en un cuádruple bucle anidado sobre todos los posibles valores de cada una de las cuatro cifras del número.

A continuación se presentan las dos implementaciones de la solución:

DIRECTORIO DEL CODIGO: EJEMPLO09 _ 02

```
// versión "clásica"
List<int> cumplen = new List<int>();
for (int i1 = 1; i1 <= 9; i1++)
   for (int i2 = 0; i2 <= 9; i2++)
      for (int i3 = 0; i3 <= 9; i3++)
         for (int i4 = 0; i4 <= 9; i4++)
            if (Condicion(i1, i2, i3, i4))
               cumplen.Add(Numero(i1, i2, i3, i4));

foreach (int n in cumplen)
   Console.WriteLine(n);

// versión LINQ
IEnumerable<int> cumplen2 =
   from j1 in Enumerable.Range(1, 9)
   from j2 in Enumerable.Range(0, 9)
   from j3 in Enumerable.Range(0, 9)
   from j4 in Enumerable.Range(0, 9)
   where Condicion(j1, j2, j3, j4)
   select Numero(j1, j2, j3, j4);

foreach (int n in cumplen2)
   Console.WriteLine(n);
```

Ambas versiones se apoyan en un mismo conjunto de métodos auxiliares:

```
static bool Condicion(int a, int b, int c, int d)
{
   return Numero(a, b, c, d) == Cuarta(a) +
      Cuarta(b) + Cuarta(c) + Cuarta(d);
}

static int Numero(int a, int b, int c, int d)
{
   return 1000 * a + 100 * b + 10 * c + d;
}

static int Cuadrado(int a) { return a * a; }

static int Cuarta(int a) { return Cuadrado(a) * Cuadrado(a); }
```

El objetivo de esta sección es hacer un llamamiento a la reflexión: ¿le parece adecuada la utilización de expresiones de consulta para tareas como ésta? Innegablemente el código funciona y es elegante; la diferencia de rendimiento con relación a la implementación "tradicional" es sustancial (puede comprobarlo usted mismo), pero cabe esperar que en futuras versiones de .NET Framework se trabaje más sobre la optimización de consultas y esa diferencia se reduzca. Si lo mío fuera "venderle la moto" de aplicar las consultas integradas hasta a la fabricación de salchichas, seguramente le diría que sí; pero, honestamente, no lo tengo tan claro. En cualquier caso, amigo lector, guíese por su propio juicio para determinar si conviene o no utilizar una consulta integrada en cada situación concreta. Recuerde, como dijo Brooks, no hay balas de plata.

CAPÍTULO 10

Operadores de consulta estándar

Presentados en el capítulo anterior los fundamentos de las consultas integradas en el lenguaje, en éste describiremos con mayor nivel de detalle todos y cada uno de los operadores de consulta estándar incorporados a la librería de clases base, presentando su interfaz de programación, así como ejemplos que ayuden a comprender su utilización. Para economizar espacio, no se describe el comportamiento de los operadores en situaciones de error obvias; por ejemplo, la mayoría de los operadores producen excepciones de tipo `ArgumentNullException` en caso de que se suministre un valor nulo para un parámetro necesario.

Todas las consultas de este capítulo se ejecutan contra las colecciones de datos de clubes y jugadores de fútbol contenidas en la clase `DatosFutbol`, por lo que es conveniente recordar esas definiciones (Introducción).

10.1. TABLA DE OPERADORES DE CONSULTA ESTÁNDAR

Como ya hemos visto, los operadores de consulta estándar son métodos extensores que extienden la interfaz genérica `IEnumerable<T>`. Estos métodos se implementan en una clase estática llamada `System.Linq.Enumerable`, dentro del ensamblado `System.Core.dll`, que se muestra a continuación en el examinador de objetos de Visual Studio 2008.

La Tabla 10.1 lista los operadores de consulta estándar disponibles, agrupados por categorías. Hemos destacado al principio los operadores "básicos" para los cuales la sintaxis de las expresiones de consulta ofrece una cláusula especial (`where`, `orderby`, `select`, `group`, `join`) y algunas de cuyas sobrecargas forman parte del llamado **patrón LINQ,** que se define exactamente en el documento de especificación de C# 3.0 y al que dedicaremos más atención en el siguiente capítulo. Para el resto de los operadores estándar no existe soporte lingüístico directo en C# 3.0, y para utilizarlos se deberá hacer uso de la sintaxis explícita de llamada a método. Observe que, si bien los operadores básicos y muchos de los no básicos producen como resultado otra secuencia, entre los demás operadores hay varios que producen como resultado valores escalares, lo que significa que deberán situarse siempre al final de la "cadena de producción".

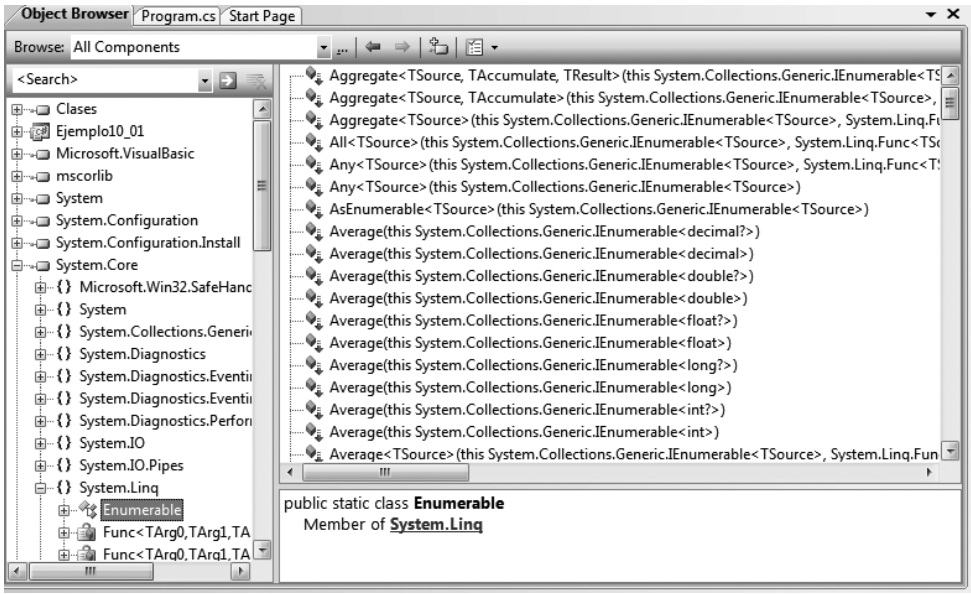


Figura 10.1.

Tabla 10.1. Operadores de consulta estándar.

Operadores del patrón LINQ	
Where()	Filtrado de la secuencia original en base a un predicado lógico.
Select()/SelectMany()	Proyección de la secuencia original en otra en base a una función de transformación.
OrderBy()/ThenBy()	Ordenación ascendente de la secuencia original en base a una función de cálculo de la clave de ordenación.
OrderByDescending()/ ThenByDescending()	Ordenación descendente de la secuencia original en base a una función de cálculo de clave de ordenación.
GroupBy()	Creación de grupos a partir de la secuencia original en base a una función de cálculo de la clave de agrupación.
Join()	Encuentro interno de la secuencia original y una segunda secuencia en base a funciones de cálculo de la clave de encuentro para cada una de las secuencias.
GroupJoin()	Encuentro agrupado de la secuencia original y una segunda secuencia en base a funciones de cálculo de la clave de encuentro para cada una de las secuencias.

Continúa

Operadores de partición	
Take()	Selección de elementos de la secuencia original situados antes de la posición especificada.
Skip()	Selección de elementos de la secuencia original situados después de la posición especificada.
TakeWhile()	Selección de elementos de la secuencia original mientras un predicado es satisfecho.
SkipWhile()	Selección de elementos de la secuencia original situados a partir de que un predicado deja de cumplirse.
Operadores conjuntuales	
Distinct()	Selección de elementos únicos de la secuencia original.
Union()	Unión de la secuencia original y una segunda secuencia.
Intersect()	Intersección de la secuencia original y una segunda secuencia.
Except()	Diferencia de la secuencia original y una segunda secuencia.
Operadores de conversión	
ToArray()	Creación de un array a partir de los elementos de la secuencia original.
ToList()	Creación de una lista genérica (`List<T>`) a partir de los elementos de la secuencia original.
ToDictionary()	Creación de un diccionario de pares clave/valor (`Dictionary<K, V>`) a partir de la secuencia original y de funciones de cálculo de la clave y el valor.
ToLookup()	Creación de un diccionario de pares clave/secuencia de elementos con ese valor de clave (`Lookup<K, V>`) a partir de la secuencia original y de funciones de cálculo de la clave y el valor.
AsEnumerable()	Cambio del tipo de la secuencia original a `IEnumerable<T>`.
Cast<T>()	Conversión del tipo de los elementos de la secuencia original a `T`.
OfType<T>()	Filtrado de los elementos de la secuencia original que son del tipo `T`.
Operadores de generación de secuencias	
Range()	Generación de una secuencia formada por `n` números enteros consecutivos a partir de un cierto valor `m`.
Repeat<T>()	Generación de una secuencia en la que se repite `n` veces un elemento de tipo `T`.
Empty<T>()	Generación de una secuencia vacía de elementos de tipo `T`.

Continúa

Otros operadores de transformación de secuencias	
Concat()	Concatenación de dos secuencias.
Reverse()	Inversión de una secuencia.
Cuantificadores	
Any()	Cuantificador existencial: devuelve `true` si alguno de los elementos de la secuencia original satisface un predicado lógico, o `false` en caso contrario.
All()	Cuantificador universal: devuelve `true` si todos los elementos de la secuencia original satisfacen un predicado lógico, o `false` en caso contrario.
Contains()	Comprueba la existencia de un elemento dado dentro de la secuencia original.
SequenceEqual()	Comprueba si la secuencia original y una segunda secuencia son iguales.
Operadores de elementos	
First()	Devuelve el primer elemento de la secuencia original, o el primer elemento para el que se cumple una condición especificada.
FirstOrDefault()	Devuelve el primer elemento de la secuencia original, o el primer elemento para el que se cumple una condición especificada. Si tal elemento no existe, devuelve el valor predeterminado del tipo de los elementos de la secuencia original.
Last()	Devuelve el último elemento de la secuencia original, o el último elemento para el que se cumple una condición especificada.
LastOrDefault()	Devuelve el último elemento de la secuencia original, o el último elemento para el que se cumple una condición especificada. Si tal elemento no existe, devuelve el valor predeterminado del tipo de los elementos de la secuencia original.
Single()	Devuelve el único elemento de la secuencia original, o el único elemento para el que se cumple una condición especificada.
SingleOrDefault()	Devuelve el único elemento de la secuencia original, o el único elemento para el que se cumple una condición especificada. Si tal elemento no existe, devuelve el valor predeterminado del tipo de los elementos de la secuencia original.
ElementAt()	Devuelve el elemento de la secuencia original situado en la posición especificada.
ElementAtOrDefault()	Devuelve el elemento de la secuencia original situado en la posición especificada. Si tal elemento no existe, devuelve el valor predeterminado del tipo de los elementos de la secuencia original.
DefaultIfEmpty()	Devuelve la misma secuencia original, o una secuencia formada por un valor predeterminado si la secuencia original es vacía.

Continúa

Operadores de consolidación (agregados)	
Count()/ LongCount()	Devuelve la cantidad de elementos en la secuencia original, o la cantidad de elementos que satisfacen un predicado lógico especificado.
Max()/Min()	Devuelve el mayor (menor) de los elementos de la secuencia original.
Sum()	Devuelve la suma de los elementos de la secuencia (numérica) original.
Average()	Devuelve la media de los elementos de la secuencia (numérica) original.
Aggregate()	Devuelve el resultado de aplicar una función de consolidación dada a los elementos de la secuencia original.

10.2. OPERADORES BÁSICOS

Los operadores básicos, algunas de cuyas sobrecargas conforman el patrón LINQ, son los que más atención han recibido de nuestra parte hasta el momento. En esta sección "redondearemos" los conocimientos ya adquiridos sobre estos operadores, que sin duda son los más relevantes de todos.

10.2.1. El operador Where()

El operador `Where()` (técnicamente, un **operador de restricción**) filtra la secuencia de entrada en base a un predicado lógico. Devuelve un objeto enumerable (secuencia) que, al ser enumerado, enumerará a su vez su secuencia de entrada y producirá los elementos de esa secuencia que satisfagan el predicado lógico. Si cualquiera de los dos argumentos es `null` (recuerde que se trata de métodos extensores), se lanza una excepción de tipo `ArgumentNullException`.

El Explorador de Objetos de Visual Studio nos dirá que este método extensor tiene dos sobrecargas:

```
public static IEnumerable<TSource> Where<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
public static IEnumerable<TSource> Where<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, int, bool> predicate);
```

De las dos, la que realmente utiliza el compilador en la traducción de expresiones de consulta es la primera. Por ejemplo, la siguiente consulta, que produce los futbolistas del Real Madrid con más de 30 años:

DIRECTORIO DEL CODIGO: EJEMPLO10 _ 01

```
var s1 = from f in DatosFutbol.Futbolistas
         where f.CodigoClub == "RMA" && f.Edad > 30
         select f.Nombre;
```

es traducida por el compilador a:

```
var s2 = DatosFutbol.Futbolistas.
     Where(f => f.CodigoClub == "RMA" && f.Edad > 30).
     Select(f => f.Nombre);
```

En la segunda variante, el predicado de filtro tiene un parámetro adicional de tipo entero, a través del cual se nos envía el índice del elemento a comprobar dentro de la secuencia original. Para hacer uso de esta variante hay que recurrir a una llamada explícita. Por ejemplo, la siguiente consulta produce los futbolistas del Real Madrid con más de 30 años que están entre los 10 jugadores más viejos de la liga:

```
var s3 = DatosFutbol.Futbolistas.
   OrderByDescending(f => f.Edad).
   Where((f, n) => f.CodigoClub == "RMA" && f.Edad > 30 && n < 10).
   Select(f => f.Nombre);
```

Hemos puesto delante del `Where()` un `OrderByDescending()`, que se describe más adelante, para asegurar una ordenación descendente previa de los futbolistas según su edad.

10.2.2. El operador Select()

Dos son los operadores estándar relacionados con la cláusula `select` de las expresiones de consulta, mediante la cual especificamos el tipo de los elementos de la secuencia de salida: `Select()` y `SelectMany()`, que en la literatura oficial se conocen como **operadores de proyección.** El primero es el que se utiliza generalmente en las consultas simples, y en su versión principal (la que forma parte del patrón LINQ) recibe como segundo parámetro una expresión lambda que representa la proyección a realizar sobre los elementos de la secuencia original.

```
public static IEnumerable<TResult> Select<TSource, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, TResult> selector);
public static IEnumerable<TResult> Select<TSource, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, int, TResult> selector);
```

Al ser enumerada, la secuencia devuelta por `Select()` enumera a su vez su secuencia de entrada y produce el resultado de la evaluación de la función de selección para cada uno de los elementos de esa secuencia. El primer argumento de la función de selección representa el elemento a tratar, mientras que el segundo, si está presente, es el índice del elemento dentro de la secuencia de entrada. La función de selección típicamente devuelve alguno de los campos del objeto original, o genera

un nuevo objeto de un tipo anónimo que contiene la información del objeto original que nos interesa seleccionar.

La siguiente consulta produce una secuencia de tuplas con los nombres y fechas de nacimiento de todos los delanteros del Fútbol Club Barcelona:

```
var s4 = from f in DatosFutbol.Futbolistas
         where f.CodigoClub == "BAR" &&
               f.Posicion == Posicion.Delantero
         select new { f.Nombre, f.FechaNac };
```

El compilador de C# 3.0 traduce la expresión de consulta anterior a:

```
var s5 = DatosFutbol.Futbolistas.
  Where(f => f.CodigoClub == "BAR" &&
             f.Posicion == Posicion.Delantero).
  Select(f => new { f.Nombre, f.FechaNac });
```

Como ejemplo de utilización (con sintaxis de llamada) de la segunda variante, analice la siguiente sentencia, que produce los índices de esos elementos dentro de la colección `Futbolistas`:

```
var s6 = DatosFutbol.Futbolistas.
  Select((f, i) => new { Futbolista = f, Indice = i }).
  Where(f => f.Futbolista.CodigoClub == "BAR" &&
             f.Futbolista.Posicion == Posicion.Delantero).
  Select(x => x.Indice);
```

El primer `Select()` adjunta a cada futbolista su índice en la colección, que luego se utiliza en el segundo `Select()`.

10.2.3. Caso trivial de Select()

Un caso especial de uso del operador `Select()` en el que el compilador de C# 3.0 aplica "de serie" una optimización es el que se produce cuando una expresión de consulta establece que se debe seleccionar el propio elemento de iteración. Por ejemplo, suponga que tenemos la siguiente consulta:

```
var s77 = from f in DatosFutbol.Futbolistas
          where f.Edad < 25
          select f;
```

Traducido a sintaxis de métodos, se trataría de lo siguiente:

```
var s77a = DatosFutbol.Futbolistas.
  Where(f => f.Edad < 25).
  Select(f => f);
```

¿Qué aporta la última llamada a `Select()`? Produce exactamente la misma secuencia que recibe. Por esta razón, el compilador optimiza estas llamadas: simplemente, no genera código para ellas.

10.2.4. El operador SelectMany()

A diferencia de `Select()`, el operador de consulta estándar `SelectMany()` se utiliza normalmente en las consultas con generadores anidados, y su objetivo es realizar una proyección de uno a muchos: está orientado al caso en que a partir de cada elemento de la secuencia de entrada se obtiene una colección de elementos relacionados, y `SelectMany()` actúa produciendo en la secuencia de salida un elemento por cada uno de los miembros de cada una de esas colecciones.

Técnicamente, `SelectMany()` se presenta en tres variantes, de las cuales la tercera (la más general) forma parte del patrón LINQ:

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, IEnumerable<TResult>> selector);
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, int, IEnumerable<TResult>> selector);
public static IEnumerable<TResult>
      SelectMany<TSource, TCollection, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, IEnumerable<TCollection>> collectionSelector,
   Func<TSource, TCollection, TResult> resultSelector);
```

En la variante básica de este operador, cuando la secuencia devuelta por `SelectMany()` es enumerada, se enumera la secuencia de entrada, mapeando cada elemento de ésta a un objeto enumerable mediante la función de selección y enumerando a su vez y produciendo en la salida los elementos de cada uno de esos objetos enumerables.

El compilador de C# 3.0 genera una llamada a `SelectMany()` para la segunda y subsiguientes cláusulas `from` de una expresión de consulta. Observe la siguiente consulta, que produce una secuencia con los delanteros de los equipos madrileños y los clubes a los que pertenecen:

```
var s7 = from c in DatosFutbol.Clubes
         where c.Ciudad == "MADRID"
         from f in DatosFutbol.Futbolistas
         where f.CodigoClub == c.Codigo &&
               f.Posicion == Posicion.Delantero
   select new { Jugador = f.Nombre, Club = c.Nombre };
```

Observe que ha sido necesario asignar nombres explícitos a las propiedades del tipo anónimo resultante, debido a que en caso contrario éste resultaría con dos propiedades llamadas `Nombre`, lo cual es obviamente inaceptable.

La sentencia anterior podría ser traducida por el compilador a:

```
var s8 = DatosFutbol.Clubes.
         Where(c => c.Ciudad == "MADRID").
         SelectMany(c =>
            DatosFutbol.Futbolistas.
            Where(f => f.CodigoClub == c.Codigo &&
                  f.Posicion == Posicion.Delantero).
            Select(f => new {
                  Jugador = f.Nombre, Club = c.Nombre }));
```

pero también (utilizando la tercera sobrecarga de `SelectMany()`, que comentaremos a continuación) a:

```
var s8a = DatosFutbol.Clubes.
          Where(c => c.Ciudad == "MADRID").
          SelectMany(c =>
             DatosFutbol.Futbolistas.
             Where(f => f.CodigoClub == c.Codigo &&
                        f.Posicion == Posicion.Delantero),
             (c, coll) => new {
                Jugador = coll.Nombre, Club = c.Nombre });
```

Este mismo ejemplo nos permite comprender la diferencia entre `Select()` y `SelectMany()`. Suponiendo que el nombre del tipo anónimo que genera el compilador para las tuplas `(nombreJugador, nombreClub)` es `TPar`, el tipo de la consulta en su estado actual será `IEnumerable<TPar>`. Si pusiéramos un `Select()` en lugar de `SelectMany()`, entonces el tipo del resultado sería `IEnumerable<IEnumerable<TPar>>`. `SelectMany()` "aplana" la secuencia resultante. Al lector familiarizado con la programación lógica o funcional `SelectMany()` le traerá a la mente la típica función **flatten** (aplanado) generalmente disponible (o fácilmente programable de manera recursiva) en esos sistemas.

En la segunda variante de `SelectMany()` se suministra adicionalmente a la función de selección el índice del elemento de la secuencia de entrada, de manera similar a como hemos visto para otros operadores. Mientras tanto, la tercera sobrecarga introduce un parámetro adicional, otra expresión lambda que nos permite un tratamiento posterior conjunto del elemento de la secuencia original y cada uno de los elementos de la secuencia interna para variar el tipo de los elementos de la secuencia resultante. Por ejemplo, el siguiente código extiende el anterior, combinando la información del club y del jugador en una cadena de caracteres:

```
var s9 = DatosFutbol.Clubes.
         Where(c => c.Ciudad == "MADRID").
         SelectMany(
            // segundo argumento - selección de colección
            c =>
            DatosFutbol.Futbolistas.
            Where(f => f.CodigoClub == c.Codigo &&
                       f.Posicion == Posicion.Delantero).
            Select(f => new {
               Jugador = f.Nombre, Club = c.Nombre }),
            // tercer argumento - selección de resultado
            (x, coll) =>
            coll.Jugador + " (" + x.Nombre + ", " +
               x.Ciudad + ")");
```

10.2.5. Operadores de ordenación

La familia de operadores `OrderBy/ThenBy` permite ordenar una secuencia de acuerdo a una o más claves de ordenación.

```
// OrderBy()
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IComparer<TKey> comparer);
// OrderByDescending()
public static IOrderedEnumerable<TSource>
     OrderByDescending<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource>
     OrderByDescending<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IComparer<TKey> comparer);
// ThenBy()
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
   this IOrderedEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
   this IOrderedEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IComparer<TKey> comparer);
// ThenByDescending()
```

Continúa

```
public static IOrderedEnumerable<TSource>
     ThenByDescending<TSource, TKey>(
   this IOrderedEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource>
     ThenByDescending<TSource, TKey>(
   this IOrderedEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IComparer<TKey> comparer);
```

Los operadores de esta familia pueden ser combinados para ordenar una secuencia según múltiples criterios. La combinación debe comenzar siempre por una aplicación de `OrderBy()` u `OrderByDescending()`, que establece el criterio de ordenación primario; a continuación, se pueden realizar llamadas a `ThenBy()` o `ThenByDescending()` para establecer criterios adicionales. Observe que los dos primeros operadores producen una secuencia de tipo `IOrderedEnumerable<T>` (heredera de `IEnumerable<T>`, por lo que pueden participar de una "cadena de producción LINQ"). Los operadores `ThenBy()` y `ThenByDescending()`, por su parte, tienen como primer parámetro un `IOrderedEnumerable<T>`, y por tanto el compilador sólo permitirá su utilización a continuación de otro operador de la familia. ¡La belleza de la orientación a objetos!

Cada criterio de ordenación se define mediante:

- Una función de selección que permitirá calcular la clave de ordenación asociada a cada elemento de la secuencia de entrada.
- Una dirección de ordenación. Los operadores `OrderBy()` y `ThenBy()` producen una ordenación ascendente, mientras que `OrderByDescending()` y `ThenByDescending()` establecen una ordenación descendente.
- Una función de comparación opcional para comparar los valores de claves. Si no se especifica ninguna, se utilizará el comparador por defecto para el tipo de datos de la clave.

Cuando la secuencia devuelta por uno de estos operadores es enumerada, se enumera la secuencia de entrada, recolectando todos sus elementos; luego se evalúa la clave de ordenación para cada uno de los elementos, se ordenan los elementos de acuerdo con los valores de claves obtenidos y se producen los elementos en el orden resultante. Estos operadores llevan a cabo una **ordenación estable:** esto es, si las claves correspondientes a dos elementos de la secuencia de entrada son iguales, el orden relativo de esos elementos se mantiene.

Las cláusulas `orderby` en las expresiones de consulta de C# 3.0 se traducen en llamadas a `OrderBy()`, `OrderByDescending()`, `ThenBy()` y `ThenByDescending()`. Por ejemplo, la siguiente consulta:

```
var s10 = from f in DatosFutbol.Futbolistas
          orderby f.CodigoClub, f.Edad descending
          select new { f.Nombre, f.CodigoClub, f.Edad };
```

es traducida por el compilador a:

```
var s11 = DatosFutbol.Futbolistas.
   OrderBy(f => f.CodigoClub).
   ThenByDescending(f => f.Edad).
   Select(f => new { f.Nombre, f.CodigoClub, f.Edad });
```

Como ejemplo de la segunda sobrecarga que ofrecen estos métodos, considere la siguiente consulta, que ordena los futbolistas por su nombre sin distinguir mayúsculas de minúsculas:

```
var s12 = DatosFutbol.Futbolistas.
   OrderBy(f => f.Nombre,
      StringComparer.CurrentCultureIgnoreCase).
   Select(f => f.Nombre);
```

También se podría utilizar un comparador creado a medida:

```
class ComparadorPrimeraLetra : IComparer<string>
{
   public   int Compare(string a, string b)
   {
      return a == null ?  (b == null ? 0   : -1)
                       : (b == null ? +1 :
         string.Compare(a[0].ToString(), b[0].ToString()));
   }
}
```

10.2.6. El operador GroupBy()

El operador estándar `GroupBy()` permite agrupar los elementos de una secuencia. Ofrece cuatro sobrecargas, de las cuales la primera y la tercera forman parte del patrón LINQ:

```
public static IEnumerable<IGrouping<TKey, TSource>>
     GroupBy<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static IEnumerable<IGrouping<TKey, TSource>>
     GroupBy<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IEqualityComparer<TKey> comparer);
public static IEnumerable<IGrouping<TKey, TElement>>
```

Continúa

```
      GroupBy<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector);
public static IEnumerable<IGrouping<TKey, TElement>>
      GroupBy<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector,
   IEqualityComparer<TKey> comparer);
```

La interfaz `IGrouping<TKey, TElement>`, devuelta por cualquiera de los cuatro operadores anteriores, se define de la siguiente forma:

```
public interface IGrouping<TKey, TElement> :
   IEnumerable<TElement>
{
   TKey Key { get; }
}
```

El parámetro `keySelector` representa a una función que calcula, para cada elemento de la secuencia de entrada, el valor de la clave de agrupación, que determinará a qué grupo pertenece el elemento en cuestión. Por otra parte, `elementSelector` representa una función de mapeado que determina qué objeto se asociará a un grupo en representación de cada elemento de la secuencia de entrada. Si no se especifica, el propio elemento de la secuencia de entrada será colocado en el grupo correspondiente.
Cuando la secuencia devuelta por `GroupBy()` es enumerada, ésta enumera la secuencia de entrada y evalúa las funciones de selección de clave y de resultado (caso de que esté presente) una vez para cada uno de los elementos de la secuencia de entrada. Una vez que esa información ha sido recopilada, se produce una secuencia de objetos que implementan `IGrouping<TKey, TElement>`, cada uno de los cuales contiene a un grupo de elementos con un mismo valor de la clave de selección. Los grupos se producen en el mismo orden en que sus valores de claves van siendo encontrados en la secuencia de entrada, y los elementos dentro de cada grupo en el orden en que aparecen en la secuencia de entrada. Si se especifica el parámetro `comparer`, el método comparador en cuestión es utilizado para determinar la igualdad de los valores de las claves.

Por ejemplo, la siguiente consulta agrupa los futbolistas según sus edades:

```
IEnumerable<IGrouping<int?, string>> s14 =
   from f in DatosFutbol.Futbolistas
      group f.Nombre by f.Edad;

// cuántos de cada edad
foreach (var s in s14)
   Console.WriteLine(s.Key + " - " + s.Count());
```

Observe cómo en el bucle **foreach** hacemos uso de la propiedad `Key` de los grupos, así como del operador de consulta estándar `Count()`, que veremos un poco más adelante.

El defecto de la consulta anterior es que los grupos se crean según la primera aparición de un futbolista con cada edad. Para poner un poco de orden a ese resumen, pudimos haber escrito la consulta así:

```
IEnumerable<IGrouping<int?, string>> s14 =
   from f in DatosFutbol.Futbolistas
      orderby f.Edad
      group f.Nombre by f.Edad;
```

o así:

```
IEnumerable<IGrouping<int?, string>> s14 =
   from f in DatosFutbol.Futbolistas
      group f.Nombre by f.Edad into g
      orderby g.Count() descending
      select g;
```

Se deja al lector la interpretación de estas consultas, además de la extracción de posibles conclusiones sobre las edades de los futbolistas de la Liga Española.

El compilador de C# 3.0 traducirá nuestra expresión de consulta original a una llamada a `GroupBy()`:

```
IEnumerable<IGrouping<int?, string>> s15 =
   DatosFutbol.Futbolistas.GroupBy(
      f => f.Edad, f => f.Nombre);
```

Observe que las expresiones de selección de clave y elemento aparecen en el orden inverso con relación a la expresión de consulta.

10.2.7. El operador Join()

El operador `Join()` ejecuta un encuentro interno de dos secuencias en base a la coincidencia de claves obtenidas para cada uno de los elementos de ambas secuencias. Ofrece dos variantes, la primera de las cuales forma parte del patrón LINQ:

```
public static IEnumerable<TResult>
      Join<TOuter, TInner, TKey, TResult>(
   this IEnumerable<TOuter> outer,
```

Continúa

```
   IEnumerable<TInner> inner,
   Func<TOuter, TKey> outerKeySelector,
   Func<TInner, TKey> innerKeySelector,
   Func<TOuter, TInner, TResult> resultSelector);
public static IEnumerable<TResult>
     Join<TOuter, TInner, TKey, TResult>(
   this IEnumerable<TOuter> outer,
   IEnumerable<TInner> inner,
   Func<TOuter, TKey> outerKeySelector,
   Func<TInner, TKey> innerKeySelector,
   Func<TOuter, TInner, TResult> resultSelector,
   IEqualityComparer<TKey> comparer);
```

Los argumentos `outerKeySelector` e `innerKeySelector` son funciones que extraen los valores de la clave de encuentro para cada uno de los elementos de la secuencia de entrada (`outer`) y la segunda secuencia (`inner`), respectivamente. Por su parte, `resultSelector` es la función encargada de generar los elementos de la secuencia resultante a partir de cada par de elementos de las secuencias cuyos valores de clave coinciden.

Cuando la secuencia devuelta por `Join()` es enumerada, ésta inicialmente enumera la secuencia interna (`inner`) y evalúa la función de extracción de clave para cada uno de sus elementos, agrupando éstos según sus valores de claves en una tabla hash. Una vez que se ha recorrido toda esa secuencia, se enumera la secuencia de entrada, evaluando la función de extracción de clave correspondiente para cada uno de sus elementos. Para cada elemento de la secuencia de entrada, se busca en la tabla hash si su valor de clave coincide con el de alguno(s) de los elementos de la segunda secuencia. Para cada coincidencia, se evalúa la función `resultSelector` a partir de los elementos cuyas claves coinciden, y se produce el elemento correspondiente en la secuencia de salida.

El operador `Join()` preserva el orden de los elementos de la secuencia de entrada, y para cada elemento de ésta, el orden de los elementos coincidentes de la segunda secuencia.

El compilador de C# 3.0 traduce las cláusulas **`join`** de las expresiones de consulta en llamadas a `Join()`. Por ejemplo, la siguiente expresión, que produce una secuencia de cadenas que reflejan los nombres de los países en los que han nacido futbolistas que juegan en la Liga Española y los nombres de éstos:

```
var s16 = from p in DatosFutbol.Paises
          join f in DatosFutbol.Futbolistas
          on p.Codigo equals f.CodigoPaisNac
          select p.Nombre + " - " + f.Nombre;
```

es traducida a:

```
var s17 = DatosFutbol.Paises.
   Join(DatosFutbol.Futbolistas,
      p => p.Codigo,
      f => f.CodigoPaisNac,
      (p, f) => p.Nombre + " - " + f.Nombre);
```

En ciertos casos, el compilador tiene que sacarse un truco de la manga a la hora de transformar las expresiones de consulta. Por ejemplo, si hubiésemos solicitado ordenar la secuencia de cadenas por los nombres de países y jugadores, la expresión de consulta habría sido:

```
var s16 = from p in DatosFutbol.Paises
          join f in DatosFutbol.Futbolistas
          on p.Codigo equals f.CodigoPaisNac
          orderby p.Nombre, f.Nombre   // ordenación
          select p.Nombre + " - " + f.Nombre;
```

El compilador compilará esta sentencia de la siguiente forma:

```
var s17 = DatosFutbol.Paises.
   Join(DatosFutbol.Futbolistas,
      p => p.Codigo,
      f => f.CodigoPaisNac,
      (p, f) => new { p, f }).   // entidad "artificial"
   OrderBy(x => x.p.Nombre).
   ThenBy(x => x.f.Nombre).
   Select(x => x.p.Nombre + " - " + x.f.Nombre);
```

De acuerdo con los términos que se manejan habitualmente en el mundo de las bases de datos relacionales, el operador `Join()` implementa un **encuentro interno** basado en la igualdad de claves. No existen operadores de consulta especiales para implementar encuentros externos por la izquierda o por la derecha, pero éstos pueden implementarse fácilmente mediante el operador `GroupJoin()` que veremos a continuación.

10.2.8. El operador GroupJoin()

El operador `GroupJoin()` ejecuta un encuentro agrupado de dos secuencias en base a la coincidencia de claves obtenidas para cada uno de los elementos de ambas secuencias. Ofrece dos variantes, la primera de las cuales forma parte del patrón LINQ:

```
public static IEnumerable<TResult>
     GroupJoin<TOuter, TInner, TKey, TResult>(
   this IEnumerable<TOuter> outer,
```

Continúa

```
   IEnumerable<TInner> inner,
   Func<TOuter, TKey> outerKeySelector,
   Func<TInner, TKey> innerKeySelector,
   Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);
public static IEnumerable<TResult>
     GroupJoin<TOuter, TInner, TKey, TResult>(
   this IEnumerable<TOuter> outer,
   IEnumerable<TInner> inner,
   Func<TOuter, TKey> outerKeySelector,
   Func<TInner, TKey> innerKeySelector,
   Func<TOuter, IEnumerable<TInner>, TResult> resultSelector,
   IEqualityComparer<TKey> comparer);
```

Los argumentos `outerKeySelector` e `innerKeySelector` son funciones que extraen los valores de la clave de encuentro para cada uno de los elementos de la secuencia de entrada (`outer`) y la segunda secuencia (`inner`), respectivamente, mientras que `resultSelector` es la función encargada de generar los elementos de la secuencia resultante a partir de cada par de elementos de las secuencias cuyos valores de clave coinciden.

Cuando la secuencia devuelta por `GroupJoin()` es enumerada, ésta inicialmente enumera la secuencia interna (`inner`) y evalúa la función de extracción de clave para cada uno de sus elementos, agrupando éstos según sus valores de claves en una tabla hash. Una vez que se ha recorrido toda esa secuencia, se enumera la secuencia de entrada, evaluando la función de extracción de clave correspondiente para cada uno de sus elementos. Para cada elemento de la secuencia de entrada, se busca en la tabla hash si su valor de clave coincide con el de alguno(s) de los elementos de la secuencia de entrada, y se evalúa la función `resultSelector` a partir del elemento de la secuencia de entrada y la secuencia (posiblemente nula) de los elementos de la secuencia interna cuyos valores de clave coinciden con el valor de la clave del elemento de la secuencia de entrada, produciendo ese resultado.

El operador `GroupJoin()` preserva el orden de los elementos de la secuencia de entrada, y para cada elemento de ésta, el orden de los elementos coincidentes de la segunda secuencia.

El compilador de C# 3.0 traduce las cláusulas `join...into` de las expresiones de consulta en llamadas a `GroupJoin()`. Por ejemplo, la siguiente consulta produce una secuencia de pares (país, cantidad de jugadores nacidos en ese país), donde la cantidad de jugadores será cero en caso de que un país no tenga jugadores que lo representen:

```
var s18 = from p in DatosFutbol.Paises
          join f in DatosFutbol.Futbolistas
            on p.Codigo equals f.CodigoPaisNac into pares
          select new {
            Pais = p.Nombre, Cantidad = pares.Count() };
```

El código anterior se traduce a:

```
var s19 = DatosFutbol.Paises.
   GroupJoin(DatosFutbol.Futbolistas,
      p => p.Codigo,
      f => f.CodigoPaisNac,
      (p, pares) =>
         new { Pais = p.Nombre, Cantidad =  pares.Count() } );
```

El operador `GroupJoin()` produce resultados jerárquicos, emparejando elementos de la secuencia de entrada con secuencias de elementos coincidentes de la segunda secuencia, y como tal no tiene equivalente en términos de las operaciones comunes sobre bases de datos relacionales.

Los encuentros externos por la izquierda y por la derecha del álgebra relacional pueden simularse utilizando encuentros agrupados y otros operadores de consulta estándar. Por ejemplo, la siguiente sentencia implementa un encuentro externo por la izquierda entre `Paises` y `Futbolistas`:

```
// encuentro interno por la izquierda
var s20 = from p in DatosFutbol.Paises
          join f in DatosFutbol.Futbolistas
            on p.Codigo equals f.CodigoPaisNac into pares
          from par in pares.DefaultIfEmpty(null)
          select new {
            Pais = p.Nombre,
            Jugador = (par == null ? "(Ninguno)" : par.Nombre)
          };
```

La cláusula `from` inmediatamente después del encuentro implica una iteración por la secuencia resultante del mismo para aplanarla; pero previamente se aplica a la secuencia el operador estándar `DefaultIfEmpty()`, que produce su misma secuencia de entrada, a excepción de que cuando uno de los elementos de esa secuencia es una secuencia vacía la sustituye por el valor predeterminado que especifiquemos como parámetro. Este valor predeterminado debe ser del tipo de los elementos de la secuencia, y generalmente **`null`** es una opción adecuada.

10.3. OPERADORES DE PARTICIÓN

Los operadores de consulta que se describen a partir de este punto no tienen equivalente sintáctico en C# 3.0, y para hacer uso de ellos se deberá utilizar la sintaxis explícita de llamada a métodos.

Los operadores de partición tienen como objetivo producir un subconjunto de elementos consecutivos de la secuencia de entrada. Específicamente, `Take()` y `Skip()` podrían utilizarse para tareas de paginación de resultados.

10.3.1. El operador Take()

El operador `Take()` produce la cantidad de elementos especificada de la secuencia de entrada, e ignora el resto de la secuencia. Se presenta en una única variante:

```
public static IEnumerable<TSource> Take<TSource>(
   this IEnumerable<TSource> source,
   int count);
```

Cuando el objeto devuelto por `Take()` es enumerado, éste enumera a su vez su secuencia de entrada y produce la cantidad de elementos indicada por el parámetro `count` (o menos, en caso de que se alcance antes el final de la secuencia).

La siguiente expresión de consulta devuelve una secuencia con los cinco equipos más avejentados (con una media de edad más alta) de la Liga:

```
var s21 = (from f in DatosFutbol.Futbolistas
           group f.Edad by f.CodigoClub into g
           orderby g.Average() descending
           select new {
              Club = g.Key, Media = g.Average() }).Take(5);
```

10.3.2. El operador Skip()

El operador `Skip()` ignora la cantidad de elementos indicada por el parámetro `count` y produce el resto de la secuencia. Su firma es:

```
public static IEnumerable<TSource> Skip<TSource>(
   this IEnumerable<TSource> source,
   int count);
```

Cuando el objeto devuelto por `Skip()` es enumerado, éste enumera a su vez su secuencia de entrada, ignorando la cantidad de elementos indicada por el parámetro `count` y produciendo el resto de la secuencia.

Los operadores `Take()` y `Skip()` son funcionalmente complementarios: dada cualquier secuencia `s`, la concatenación de `s.Take(n)` y `s.Skip(n)` produce la secuencia `s`.

La siguiente sentencia devuelve los códigos y medias de edad de los equipos de la Liga, exceptuando los cinco primeros:

```
var s22 = (from f in DatosFutbol.Futbolistas
           group f.Edad by f.CodigoClub into g
           orderby g.Average() descending
           select new {
              Club = g.Key, Media = g.Average() }).Skip(5);
```

10.3.3. El operador TakeWhile()

El operador `TakeWhile()` produce los elementos de la secuencia de entrada mientras que éstos cumplen cierta condición. Tan pronto se encuentra un elemento en la secuencia de entrada que no satisface el predicado, la enumeración termina y se ignora el resto de la secuencia. Se presenta en dos variantes:

```
public static IEnumerable<TSource> TakeWhile<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
public static IEnumerable<TSource> TakeWhile<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, int, bool> predicate);
```

Cuando el objeto devuelto por `TakeWhile()` es enumerado, éste enumera a su vez su secuencia de entrada y comprueba para cada uno de los elementos la condición indicada por el predicado. Mientras no se alcance el final de la secuencia y se cumpla la condición, el operador producirá los elementos de la secuencia de entrada. Tan pronto se encuentre un elemento para el que el predicado devuelva `false`, o se alcance el final de la secuencia, la enumeración se detendrá.

La siguiente expresión de consulta devuelve una secuencia con los equipos de la Liga cuya media de edad es mayor o igual a 25 años:

```
var s23 = (from f in DatosFutbol.Futbolistas
           group f.Edad by f.CodigoClub into g
           orderby g.Average() descending
           select new { Club = g.Key, Media = g.Average() }).
              TakeWhile(g => g.Media >= 25);
```

En la segunda variante se envía adicionalmente al predicado lógico el índice (posición en la secuencia) del elemento a comprobar.

10.3.4. El operador SkipWhile()

El operador `SkipWhile()` descarta los elementos de la secuencia de entrada mientras que éstos cumplen cierta condición. Tan pronto se encuentra un elemento en la secuencia de entrada que no satisface el predicado, se comienza a producir los elementos de la secuencia. Ofrece dos variantes:

```
public static IEnumerable<TSource> SkipWhile<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
public static IEnumerable<TSource> SkipWhile<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, int, bool> predicate);
```

Cuando el objeto devuelto por `SkipWhile()` es enumerado, éste enumera a su vez su secuencia de entrada y comprueba para cada uno de los elementos la condición

indicada por el predicado. Mientras no se alcance el final de la secuencia y se cumpla la condición, el operador ignorará los elementos de la secuencia de entrada. Tan pronto se encuentre un elemento para el que el predicado devuelva `false`, se comenzará a enumerar los elementos hasta el final de la secuencia.

La siguiente expresión de consulta devuelve una secuencia con los equipos de la Liga cuya media de edad es inferior a 25 años:

```
var s24 = (from f in DatosFutbol.Futbolistas
           group f.Edad by f.CodigoClub into g
           orderby g.Average() descending
           select new {
              Club = g.Key, Media = g.Average() }).
           SkipWhile(g => g.Media >= 25);
```

10.4. OPERADORES CONJUNTUALES

Los operadores conjuntuales permiten modelar las operaciones del álgebra de conjuntos (unión, intersección, diferencia) sobre secuencias.

10.4.1. El operador Distinct()

El operador `Distinct()` elimina los elementos duplicados de una secuencia. Se presenta en dos variantes:

```
public static IEnumerable<TSource> Distinct<TSource>(
   this IEnumerable<TSource> source);
public static IEnumerable<TSource> Distinct<TSource>(
   this IEnumerable<TSource> source,
   IEqualityComparer<TSource> comparer);
```

Cuando el objeto devuelto por `Distinct()` es enumerado, éste enumera su secuencia de entrada, produciendo cada elemento que no haya sido producido anteriormente. Si se suministra un objeto comparador, éste se utiliza para comparar los elementos en lugar del comparador predeterminado para el tipo de los elementos de la secuencia.

La siguiente expresión de consulta devuelve una secuencia de todos los países representados en la Liga Española:

```
var s25 = (from f in DatosFutbol.Futbolistas
           select f.CodigoPaisNac).
           Distinct().
           Join(DatosFutbol.Paises,
              c => c,      // identidad
              p => p.Codigo,
              (c, p) => p.Nombre).
        OrderBy(n => n);
```

Observe cómo una vez que nos pasamos a la codificación explícita de operadores, luego hay que seguir el juego… Si comprende cómo funciona la sentencia anterior, está usted preparado para empeños mayores.

10.4.2. El operador Union()

El operador `Union()` produce la unión conjuntual de dos secuencias. Ofrece dos sobrecargas:

```
public static IEnumerable<TSource> Union<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second);
public static IEnumerable<TSource> Union<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second,
   IEqualityComparer<TSource> comparer);
```

Cuando el objeto devuelto por el operador `Union()` es enumerado, éste enumera todos los elementos de la primera y segunda secuencias (la secuencia de entrada y la otra), en ese orden, produciendo los elementos que aún no hubieran sido producidos anteriormente. Si se suministra un objeto comparador, éste se utiliza para comparar los elementos en lugar del comparador predeterminado para el tipo de los elementos de ambas secuencias (que debe ser el mismo).

Por ejemplo, en el siguiente fragmento de código la variable `enAlguno`, al ser enumerada, producirá una secuencia con los países que están representados por algún futbolista en el Real Madrid o el Fútbol Club Barcelona:

```
var mad = JugadoresDeEquipo("RMA");
var bcn = JugadoresDeEquipo("BAR");

var enAlguno = mad.Union(bcn).OrderBy(x => x);
```

Este código asume que las siguientes definiciones están en vigor:

```
// tipo delegado a método que devuelve una secuencia
delegate IEnumerable<string> ListaCadenas(string x);

// método que produce una secuencia de los
// distintos países representados en un club determinado
static ListaCadenas JugadoresDeEquipo =
   (codigo =>
      (from f in DatosFutbol.Futbolistas
      where f.CodigoClub == codigo
      select f.CodigoPaisNac).Distinct());
```

10.4.3. El operador Intersect()

El operador `Intersect()` produce la intersección conjuntual de dos secuencias. Ofrece dos sobrecargas:

```
public static IEnumerable<TSource> Intersect<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second);
public static IEnumerable<TSource> Intersect<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second,
   IEqualityComparer<TSource> comparer);
```

Cuando el objeto devuelto por el operador `Intersect()` es enumerado, éste enumera la primera secuencia (la de entrada), recolectando los elementos diferentes de esa secuencia. A continuación, enumera la segunda secuencia, marcando los elementos que están presentes en ambas secuencias. Finalmente, produce los elementos comunes a ambas secuencias, en el orden en que los fue recolectando. Si se suministra un objeto comparador, éste se utiliza para comparar los elementos en lugar del comparador predeterminado para el tipo de los elementos de ambas secuencias (que debe ser el mismo).

Por ejemplo, en el siguiente fragmento de código la variable `enAmbos`, al ser enumerada, producirá una secuencia con los países que están representados por algún futbolista en el Real Madrid y en el Fútbol Club Barcelona:

```
var enAmbos = mad.Intersect(bcn).OrderBy(x => x);
```

Este código asume que están en vigor las definiciones dadas en el epígrafe dedicado al operador `Union()`.

10.4.4. El operador Except()

El operador `Except()` produce la diferencia conjuntual de dos secuencias. Ofrece dos sobrecargas:

```
public static IEnumerable<TSource> Except<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second);
public static IEnumerable<TSource> Except<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second,
   IEqualityComparer<TSource> comparer);
```

Cuando el objeto devuelto por el operador `Except()` es enumerado, éste enumera la primera secuencia (la de entrada), recolectando los elementos diferentes de esa

secuencia. A continuación, enumera la segunda secuencia, eliminando los elementos que están presentes en la primera secuencia. Finalmente, produce los elementos que quedan de la primera secuencia, en el orden en que los fue recolectando. Si se suministra un objeto comparador, éste se utiliza para comparar los elementos en lugar del comparador predeterminado para el tipo de los elementos de ambas secuencias (que debe ser el mismo).

Por ejemplo, en el siguiente fragmento de código la variable `soloMadrid`, al ser enumerada, producirá una secuencia con los países que están representados por algún futbolista en el Real Madrid y en el Fútbol Club Barcelona:

```
var soloMadrid = mad.Except(bcn).OrderBy(x => x);
```

Este código asume que están en vigor las definiciones dadas en el epígrafe dedicado al operador `Union()`.

10.5. OPERADORES DE CONVERSIÓN

Los operadores de conversión se encargan de diferentes tareas relacionadas con la conversión de un tipo de datos a otro de secuencias como un todo o de los elementos de las mismas.

10.5.1. El operador ToArray()

El operador `ToArray()` enumera la secuencia de entrada y crea un array a partir de ella. Su firma es:

```
public static TSource[] ToArray<TSource>(
   this IEnumerable<TSource> source);
```

La siguiente expresión de consulta devuelve un array con los nombres de todos los equipos en los que juegan jugadores franceses:

```
string[] s26 =
   (from f in DatosFutbol.Futbolistas
   where f.CodigoPaisNac == "FR"
   join c in DatosFutbol.Clubes
      on f.CodigoClub equals c.Codigo
   select c.Nombre).Distinct().ToArray();
```

10.5.2. El operador ToList()

El operador `ToList()` enumera la secuencia de entrada y crea una colección `List<TSource>` a partir de ella. Su firma es:

```
public static TList<TSource> ToList<TSource>(
   this IEnumerable<TSource> source);
```

La siguiente expresión de consulta crea una lista de cadenas con los nombres de todos los equipos en los que juegan jugadores brasileños:

```
List<string> s27 =
   (from f in DatosFutbol.Futbolistas
   where f.CodigoPaisNac == "BR"
   join c in DatosFutbol.Clubes
      on f.CodigoClub equals c.Codigo
   select c.Nombre).Distinct().ToList();
```

10.5.3. El operador ToDictionary()

El operador `ToDictionary()` crea un `Dictionary<TKey,TElement>` a partir de una secuencia. Ofrece las siguientes sobrecargas:

```
public static Dictionary<TKey, TSource>
     ToDictionary<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static Dictionary<TKey, TSource>
     ToDictionary<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IEqualityComparer<TKey> comparer);
public static Dictionary<TKey, TElement>
     ToDictionary<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector);
public static Dictionary<TKey, TElement>
     ToDictionary<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector,
   IEqualityComparer<TKey> comparer);
```

El operador `ToDictionary()` enumera la secuencia de entrada y evalúa las funciones `keySelector` y `elementSelector` para cada elemento, para obtener la clave y el valor asociadas a ese elemento. Los pares (clave, valor) resultantes se devuelven en un diccionario `Dictionary<TKey, TElement>`. Si no se especifica una función de cálculo del valor, entonces el valor de cada elemento es simplemente el propio elemento. Si la función de cálculo de claves produce un mismo valor para dos elementos diferentes, se lanza una excepción. Si se suministra un objeto comparador, éste se utiliza para comparar los valores de las claves en lugar del comparador predeterminado.

La siguiente consulta crea un objeto `Dictionary<int, int>` que mapea cada año a la cantidad de futbolistas de la Liga nacidos en ese año:

```
Dictionary<int, Futbolista> año_cantJugadores =
   (from f in DatosFutbol.Futbolistas
   group f by f.FechaNac.Value.Year).
   OrderBy(g => g.Key).
   ToDictionary(
      x => x.Key,
      y => y.Count());
```

10.5.4. El operador ToLookup()

El operador `ToLookup()` crea un objeto que implementa la interfaz `ILookup<TKey,TElement>` a partir de una secuencia. Ofrece las siguientes sobrecargas:

```
public static ILookup<TKey, TSource>
     ToLookup<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static ILookup<TKey, TSource>
     ToLookup<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IEqualityComparer<TKey> comparer);
public static ILookup<TKey, TElement>
     ToLookup<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector);
public static ILookup<TKey, TElement>
     ToLookup<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector,
   IEqualityComparer<TKey> comparer);
```

La interfaz `System.Linq.ILookup` está definida en `System.Core.dll` de la siguiente forma:

```
public interface ILookup<TKey, TElement> :
   IEnumerable<IGrouping<TKey, TElement>>, IEnumerable
{
   int Count { get; }
   IEnumerable<TElement> this[TKey key] { get; }
   bool Contains(TKey key);
}
```

La implementación del operador `ToLookup()` de LINQ to Objects produce un objeto de la clase `Lookup<TKey, TElement>`, que implementa la interfaz `ILookup<TKey, TElement>`. Esta clase representa a un diccionario uno-a-muchos que mapea claves a secuencias de valores, a diferencia de `Dictionary<TKey, TElement>`, que implementa un diccionario uno-a-uno que mapea claves a valores únicos. El tipo `Lookup<TKey, TElement>` se utiliza en las implementaciones de los operadores `Join()`, `GroupJoin()` y `GroupBy()`.

El operador `ToLookup()` enumera la secuencia de entrada y evalúa las funciones `keySelector` y `elementSelector` para cada elemento, para obtener la clave y el valor asociadas a ese elemento. Los pares (clave, valor) resultantes se devuelven en un `Lookup<TKey, TElement>`. Si no se especifica una función de cálculo del valor, entonces el valor de cada elemento es simplemente el propio elemento. Si se suministra un objeto comparador, éste se utiliza para comparar los valores de las claves en lugar del comparador predeterminado.

El siguiente ejemplo construye un objeto `Lookup<int, Futbolista>` que mapea los años a secuencias de futbolistas nacidos ese año:

```
ILookup<int, Futbolista> año_Jugadores =
   DatosFutbol.Futbolistas.ToLookup(
      f => f.FechaNac.Value.Year,
      f => f);
```

Una vez creado ese objeto, podremos consultarlo de esta manera:

```
int nacidos1977 = año_Jugadores[1977].Count();
```

10.5.5. El operador AsEnumerable()

El operador `AsEnumerable()` simplemente devuelve su argumento de entrada, cambiando el tipo estático de éste a `IEnumerable<TSource>`.

```
public static IEnumerable<TSource> AsEnumerable<TSource>(
   this IEnumerable<TSource> source);
```

Este operador no tiene efecto alguno sobre la ejecución. Su único objetivo es indicar al compilador que el tipo de una secuencia es `IEnumerable<TSource>` y no otro, de modo que éste elija la implementación estándar de los operadores de consulta antes que otro más específico.

Por ejemplo, una cierta clase `MiColeccion<T>` podría tener una implementación específica del patrón LINQ (en el próximo capítulo veremos cómo hacerlo), en cuyo caso, la utilización de operadores como `Where()`, `Select()`, etc. provocaría llamadas a los métodos específicos definidos para esa clase. Si en cierto momento se deseara ejecutar los operadores de consulta estándar sobre objetos de ese tipo

en vez de los métodos especializados, el operador `AsEnumerable()` sería la vía para lograrlo.

Por ejemplo, si tuviéramos una referencia `miColeccion` a un objeto de tipo `MiColeccion<T>` y quisiéramos aplicarle el operador de consulta estándar `Where()` en lugar del que se ha definido específicamente para ese tipo, podríamos hacer:

```
Random rand = new Random();
// ...
MiColeccion subConjunto =
   miColeccion.AsEnumerable().Where(rand.Next(5) >= 3);
```

10.5.6. El operador Cast<T>()

El operador `Cast<T>()` convierte los elementos de una secuencia a un tipo dado. Su firma es:

```
public static IEnumerable<T> Cast<TResult>(
   this IEnumerable source);
```

Cuando el objeto devuelto por `Cast<T>()` es enumerado, éste enumera su secuencia de entrada y produce cada uno de los elementos de ésta convertidos al tipo `T`. Si algún elemento no se puede convertir, se producirá una excepción de tipo `InvalidCastException`.

El operador `Cast<T>()` se puede utilizar para aplicar los operadores de consulta estándar a colecciones no genéricas. Por ejemplo, si cierto código creado para .NET 1.1 utiliza el tipo no genérico `ArrayList` para implementar una colección de objetos de un mismo tipo, el operador `Cast<T>()` se podrá utilizar para suministrar la información de tipos necesaria:

```
ArrayList a = new ArrayList();
a.Add("Uno");
a.Add("Dos");
a.Add("Tres");
var consultaArrayList = from s in a.Cast<string>()
                        where s.Length == 3
                        select s;
foreach (var s in consultaArrayList)
   Console.WriteLine(s);
```

En las expresiones de consulta de C# 3.0, una variable de iteración explícitamente tipada genera una llamada a `Cast<T>()`. Por ejemplo, la siguiente consulta:

```
var s28 = from Futbolista f in DatosFutbol.Futbolistas
          orderby f.CodigoClub, f.Edad descending
          select new { f.Nombre, f.CodigoClub, f.Edad };
```

se traduce a:

```
var s29 = DatosFutbol.Futbolistas.
          Cast<Futbolista>().
          Select(f => new { f.Nombre, f.CodigoClub, f.Edad });
```

10.5.7. El operador OfType<T>()

El operador `OfType()` filtra los elementos de una secuencia en base a un tipo.

```
public static IEnumerable<T> OfType<TResult>(
   this IEnumerable source);
```

Cuando el objeto devuelto por `OfType<T>()` es enumerado, éste enumera su secuencia de entrada, produciendo aquellos elementos que sean de tipo `T`. Más exactamente, para cada elemento de la secuencia de entrada se comprueba si la condición `(e is T)` es `true`, y en caso afirmativo se produce `(T)e`.

Por ejemplo, si tenemos una lista de objetos `personas`, estáticamente de tipo `Persona`, pero en la que algunos elementos son de la clase `Empleado`, descendiente de `Persona`, podemos crear una secuencia que devolverá sólo los empleados así:

```
IEnumerable<Empleado> empleados = personas.OfType<Empleado>();
```

El operador `OfType<T>()` se puede utilizar también para aplicar los operadores de consulta estándar a colecciones no genéricas. Por ejemplo, si cierto código creado para .NET 1.1 utiliza el tipo no genérico `ArrayList` para implementar una colección de objetos de diferentes tipos, el operador `OfType<T>()` se podrá utilizar para filtrar los elementos de un tipo específico:

```
ArrayList b = new ArrayList();
b.Add("Uno");
b.Add(2);
b.Add("Tres");

var consultaArrayList2 = from s in b.OfType<string>()
                         where s.Length == 3
                         select s;
```

10.6. OPERADORES DE GENERACIÓN DE SECUENCIAS

Los operadores de generación de secuencias producen una nueva secuencia, sin necesitar de la existencia previa de otra secuencia anterior.

10.6.1. El operador Range()

El operador `Range()` produce una secuencia de números enteros. Recibe dos parámetros: el valor inicial y la cantidad de elementos a generar:

```
public static IEnumerable<int> Range(int start, int count);
```

Cuando el objeto devuelto por `Range()` es enumerado, produce `count` números enteros consecutivos a partir del valor de `start`.

El siguiente ejemplo produce un array con los cuadrados de los números del 1 al 100:

```
int[] cuadrados = Enumerable.Range(1, 100).
   Select(x => x * x).
   ToArray();
```

Observe la necesidad de indicar el nombre de la clase en la que los operadores de consulta estándar residen, `Enumerable`.

10.6.2. El operador Repeat<T>()

El operador `Repeat<T>()` produce una secuencia en la que un mismo elemento se repite una cierta cantidad de veces. Recibe como parámetros el valor a repetir (de tipo `T`) y la cantidad de repeticiones:

```
public static IEnumerable<T> Repeat<T>(
   T element, int count);
```

Cuando el objeto devuelto por `Repeat<T>` es enumerado, produce `element` un total de `count` veces.

El siguiente ejemplo construye un array de 500 elementos inicializados con el valor 1:

```
int[] arr = Enumerable.Repeat(1, 500).ToArray();
```

Gracias a la inferencia de tipos que ofrece el compilador, no es necesario especificar el tipo de los elementos de la secuencia.

10.6.3. El operador Empty<T>()

El operador `Empty<T>()` produce una secuencia vacía de elementos de un tipo dado.

```
public static IEnumerable<T> Empty<T>();
```

Cuando el objeto devuelto por `Empty<T>()` es enumerado, simplemente no produce nada.

La siguiente sentencia inicializa una secuencia vacía de futbolistas:

```
IEnumerable<Futbolista> ninguno =
   Enumerable.Empty<Futbolista>();
```

10.7. OTROS OPERADORES DE TRANSFORMACIÓN DE SECUENCIAS

En esta categoría hemos agrupado a diversos operadores que reciben una o más secuencias de entrada y realizan una transformación sobre ellas.

10.7.1. El operador Concat()

El operador `Concat()` concatena dos secuencias:

```
public static IEnumerable<TSource> Concat<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second);
```

Cuando el objeto devuelto por `Concat()` es enumerado, éste enumera la secuencia de entrada, produciendo cada uno de sus elementos, y luego enumera la segunda secuencia, produciendo cada uno de sus elementos.

El siguiente ejemplo produce una secuencia con los futbolistas del Real Madrid, el Atlético de Madrid y el Getafe (los tres equipos madrileños en la Liga):

```
IEnumerable<Futbolista> madrileños =
   (from f in DatosFutbol.Futbolistas
   where f.CodigoClub == "RMA" select f).
   Concat(
      (from f in DatosFutbol.Futbolistas
       where f.CodigoClub == "ATM" select f)).
   Concat(
      (from f in DatosFutbol.Futbolistas
       where f.CodigoClub == "GET" select f));
```

10.7.2. El operador Reverse()

El operador `Reverse()` invierte el orden de los elementos de una secuencia.

```
public static IEnumerable<TSource> Reverse<TSource>(
   this IEnumerable<TSource> source);
```

Cuando el objeto devuelto por `Reverse()` es enumerado, éste enumera la secuencia de entrada, recolectando todos los elementos y, entonces, produce los elementos de la secuencia en orden inverso.

La siguiente expresión producirá la secuencia de números naturales del 10 hasta el 1:

```
IEnumerable<int> inv = Enumerable.Repeat(1, 10).Reverse();
```

10.8. CUANTIFICADORES

Los cuantificadores son operadores que producen resultados lógicos (booleanos) en base al análisis de los elementos de una secuencia.

10.8.1. El operador Any()

El cuantificador existencial `Any()` comprueba si algún elemento de una secuencia satisface cierta condición. Se presenta en dos variantes:

```
public static bool Any<TSource>(
   this IEnumerable<TSource> source);
public static bool Any<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `Any()` enumera la secuencia de entrada y devuelve `true` si alguno de los elementos de ésta satisfacen el predicado lógico suministrado. Si no se suministra un predicado, el operador `Any()` simplemente devuelve `true` si la secuencia de entrada contiene algún elemento.

`Any()` opera en régimen de cortocircuito: la enumeración de la secuencia de entrada se termina tan pronto como se conoce el resultado final a devolver. En este caso, tan pronto se encuentre un elemento que satisfaga el predicado, la enumeración terminará (y el resultado será `true`).

El siguiente ejemplo comprueba si en la Liga Española hay algún futbolista cubano:

```
bool b1 = DatosFutbol.Futbolistas.Any(
   f => f.CodigoPaisNac == "CU");
```

Desgraciadamente, la consulta anterior devolverá `false` después de recorrer los 400+ jugadores de la Liga.

10.8.2. El operador All()

El cuantificador universal `All()` comprueba si todos los elementos de una secuencia satisfacen cierta condición. Ofrece una única variante:

```
public static bool All<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `All()` enumera la secuencia de entrada y devuelve `true` si todos los elementos de ésta satisfacen el predicado lógico suministrado. `All()` devuelve `true` si la secuencia de entrada es vacía, lo cual es consistente con la lógica de predicados y el lenguaje SQL.

`All()` opera en régimen de cortocircuito: la enumeración de la secuencia de entrada se termina tan pronto como se conoce el resultado final a devolver. En este caso, tan pronto se encuentre un elemento que no satisfaga el predicado, la enumeración terminará (y el resultado será `false`).

El siguiente ejemplo comprueba si todos los jugadores de la Liga Española tienen menos de 40 años:

```
bool b2 = DatosFutbol.Futbolistas.All(f => f.Edad < 40);
```

10.8.3. El operador Contains()

El operador `Contains()` comprueba si una secuencia contiene un elemento dado. Ofrece dos sobrecargas:

```
public static bool Contains<TSource>(
   this IEnumerable<TSource> source,
   TSource value);
public static bool Contains<TSource>(
   this IEnumerable<TSource> source,
   TSource value,
   IEqualityComparer<TSource> comparer);
```

`Contains()` primero comprueba si la secuencia de entrada implementa la interfaz `ICollection<TSource>`. En ese caso, para obtener el resultado se utiliza el método `Contains()` de la implementación en el tipo de la secuencia de la interfaz `ICollection<TSource>`. En caso contrario, el operador enumerará la secuencia de entrada para determinar si contiene un elemento con el valor buscado. La enumeración de la secuencia terminará tan pronto se encuentre un elemento con el valor buscado.

10.8.4. El operador SequenceEqual()

El operador `SequenceEqual()` comprueba si dos secuencias son iguales. Ofrece dos variantes:

```
public static bool SequenceEqual<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second);
public static bool SequenceEqual<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second,
   IEqualityComparer<TSource> comparer);
```

`SequenceEqual()` enumera tanto la secuencia de entrada como la otra en paralelo y compara los elementos correspondientes. Si se suministra un comparador, éste es utilizado; en caso contrario, se utiliza el comparador predeterminado del tipo de los elementos de las secuencias. El operador devuelve `true` si todos los elementos correspondientes son iguales y las dos secuencias tienen la misma longitud; en caso contrario, devuelve `false`.

10.9. OPERADORES DE ELEMENTOS

Los operadores de este grupo devuelven un único elemento de una secuencia.

10.9.1. El operador First()

El operador `First()` devuelve el primer elemento de una secuencia, o el primero que satisface un predicado lógico:

```
public static TSource First<TSource>(
   this IEnumerable<TSource> source);
public static TSource First<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `First()` enumera la secuencia de entrada y devuelve el primer elemento de la misma para el que el predicado especificado devuelve `true`. Si no se suministra un predicado, el operador simplemente devuelve el primer elemento de la secuencia. Si ningún elemento de la secuencia satisface el predicado o la secuencia es vacía, se lanzará una excepción `InvalidOperationException`.

El siguiente ejemplo devuelve el primer futbolista de la lista original que tiene más de 35 años:

```
Futbolista f1 = DatosFutbol.Futbolistas.
   First(f => f.Edad > 35);
```

Para evitar que se produzca una excepción en caso de que no exista ningún elemento que cumpla la condición especificada, utilice el operador `FirstOrDefault()`.

10.9.2. El operador FirstOrDefault()

El operador `FirstOrDefault()` devuelve el primer elemento de una secuencia, o el primero que satisface un predicado lógico. Si no se encuentra ningún elemento, el operador devuelve el valor predeterminado del tipo de la secuencia.

```
public static TSource FirstOrDefault<TSource>(
   this IEnumerable<TSource> source);
```

Continúa

```
public static TSource FirstOrDefault<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `FirstOrDefault()` enumera la secuencia de entrada y devuelve el primer elemento de la misma para el que el predicado especificado devuelve `true`. Si no se suministra un predicado, el operador simplemente devuelve el primer elemento de la secuencia. Si ningún elemento de la secuencia satisface el predicado o la secuencia es vacía, se devuelve **`default`**`(TSource)`, el valor predeterminado del tipo (sobre el que hemos hablado en el Capítulo 2). El valor predeterminado para los tipos referencia y los tipos anulables es **`null`.**

10.9.3. El operador Last()

El operador `Last()` devuelve el último elemento de una secuencia, o el último que satisface un predicado lógico:

```
public static TSource Last<TSource>(
   this IEnumerable<TSource> source);
public static TSource Last<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `Last()` enumera la secuencia de entrada y devuelve el último elemento de la misma para el que el predicado especificado devuelve `true`. Si no se suministra un predicado, el operador simplemente devuelve el último elemento de la secuencia. Si ningún elemento de la secuencia satisface el predicado o la secuencia es vacía, se lanzará una excepción `InvalidOperationException`.

10.9.4. El operador LastOrDefault()

El operador `LastOrDefault()` devuelve el último elemento de una secuencia, o el último que satisface un predicado lógico. Si no se encuentra ningún elemento, el operador devuelve el valor predeterminado del tipo de la secuencia.

```
public static TSource LastOrDefault<TSource>(
   this IEnumerable<TSource> source);
public static TSource LastOrDefault<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `LastOrDefault()` enumera la secuencia de entrada y devuelve el último elemento de la misma para el que el predicado especificado devuelve `true`. Si no se suministra un predicado, el operador simplemente devuelve el último elemento de la secuencia. Si ningún elemento de la secuencia satisface el predicado o

la secuencia es vacía, se devuelve **`default`**`(TSource)`, el valor predeterminado del tipo. El valor predeterminado para los tipos referencia y los tipos anulables es **`null`.**

10.9.5. El operador Single()

El operador `Single()` devuelve el único elemento de una secuencia.

```
public static TSource Single<TSource>(
   this IEnumerable<TSource> source);
public static TSource Single<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `Single()` enumera la secuencia de entrada y devuelve el único elemento para el que el predicado especificado se cumple. Si no se especifica un predicado, el operador simplemente devuelve el único elemento de la secuencia.

Si en la secuencia de entrada no hay ningún elemento que satisfaga la condición o hay más de uno, se lanzará una excepción `InvalidOperationException`.

El siguiente ejemplo devuelve el único futbolista conocido como "Raúl":

```
Futbolista f2 = DatosFutbol.Futbolistas.
   Single(f => f.Nombre == "RAUL");
```

Para evitar que se produzca una excepción en caso de que no exista ningún elemento que cumpla la condición especificada, utilice el operador `SingleOrDefault()`.

10.9.6. El operador SingleOrDefault()

El operador `SingleOrDefault()` devuelve el único elemento de una secuencia. Si no se encuentra ningún elemento, el operador devuelve el valor predeterminado del tipo de la secuencia.

```
public static TSource Single<TSource>(
   this IEnumerable<TSource> source);
public static TSource Single<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `SingleOrDefault()` enumera la secuencia de entrada y devuelve el único elemento para el que el predicado especificado se cumple. Si no se especifica un predicado, el operador simplemente devuelve el único elemento de la secuencia.

Si en la secuencia de entrada hay más de un elemento que satisfaga la condición, se lanzará una excepción `InvalidOperationException`. Si ningún elemento de la secuencia satisface el predicado o la secuencia es vacía, se devuelve **`default`**`(TSource)`, el valor predeterminado del tipo. El valor predeterminado para los tipos referencia y los tipos anulables es **`null`.**

10.9.7. El operador ElementAt()

El operador `ElementAt()` devuelve el elemento situado en una posición específica de la secuencia de entrada.

```
public static TSource ElementAt<TSource>(
   this IEnumerable<TSource> source,
   int index);
```

El operador `ElementAt()` primero comprueba si la secuencia de entrada implementa `IList<TSource>`. En ese caso, se utilizará la implementación en la clase de la secuencia de entrada de `IList<TSource>` para obtener el elemento situado en la posición `index`. En caso contrario, el operador enumerará la secuencia de entrada hasta que `index` elementos hayan sido saltados, y el elemento encontrado en esa posición en la secuencia será devuelto. Si `index` es mayor o igual que la cantidad de elementos en la secuencia, se producirá una excepción de tipo `ArgumentOutOfRangeException`.

La siguiente consulta devuelve el tercer equipo con más jugadores mayores de 30 años en la Liga:

```
var s30 = (from f in DatosFutbol.Futbolistas
           where f.Edad > 30
           group f by f.CodigoClub into grupos
           orderby grupos.Count() descending
           select new { grupos.Key, Cant = grupos.Count() }).
          ElementAt(2).Key;
```

10.9.8. El operador ElementAtOrDefault()

El operador `ElementAt()` devuelve el elemento situado en una posición específica de la secuencia de entrada, o el valor predeterminado del tipo de la secuencia si el índice está fuera de rango.

```
public static TSource ElementAtOrDefault<TSource>(
   this IEnumerable<TSource> source,
   int index);
```

El operador `ElementAtOrDefault()` primero comprueba si la secuencia de entrada implementa `IList<TSource>`. En ese caso, se utilizará la implementación en la clase de la secuencia de entrada de `IList<TSource>` para obtener el elemento situado en la posición `index`. En caso contrario, el operador enumerará la secuencia de entrada hasta que `index` elementos hayan sido saltados, y el elemento encontrado en esa posición en la secuencia será devuelto. Si `index` es mayor o igual que la cantidad de elementos en la secuencia, se devolverá **`default`**`(TSource)`. El valor predeterminado para los tipos referencia y los tipos anulables es **`null`.**

10.9.9. El operador DefaultIfEmpty()

El operador `DefaultIfEmpty()` suministra un valor predeterminado para una secuencia vacía. Ofrece dos variantes:

```
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
   this IEnumerable<TSource> source);
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
   this IEnumerable<TSource> source,
   TSource defaultValue);
```

Cuando el objeto devuelto por `DefaultIfEmpty()` es enumerado, éste enumera la secuencia de entrada y produce sus elementos. Si la secuencia de entrada es vacía, se produce un único elemento con el valor predeterminado `defaultValue`. Si este valor no se suministra, se devolverá **`default`**`(TSource)` en lugar de la secuencia vacía. El valor predeterminado para los tipos referencia y los tipos anulables es **`null`.**

Como hemos visto antes en el capítulo, el operador `DefaultIfEmpty()` puede utilizarse en combinación con un encuentro agrupado para obtener un encuentro externo por la izquierda.

10.10. AGREGADOS

Los agregados (operadores de consolidación) permiten calcular toda una serie de valores estadísticos escalares en base a los elementos de una secuencia.

10.10.1. Los operadores Count() y LongCount()

El operador `Count()` cuenta la cantidad de elementos en la secuencia de entrada. Ofrece dos variantes:

```
public static int Count<TSource>(
   this IEnumerable<TSource> source);
public static int Count<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

El operador `Count()` sin un predicado primero comprueba si la secuencia de entrada implementa `ICollection<TSource>`. En tal caso, la implementación de `ICollection<TSource>` en la secuencia es utilizada para obtener la cantidad de elementos. En caso contrario, la secuencia de entrada es enumerada hasta su fin para contar la cantidad de elementos.

El operador `Count()` con un predicado enumera la secuencia de entrada y cuenta la cantidad de elementos que satisfacen el predicado suministrado.

Para ambas sobrecargas de `Count()`, se producirá una excepción de tipo `OverflowException` si la cantidad excede **`int.`**`MaxValue`. En este caso es donde cobra sentido el uso del operador `LongCount()`:

```
public static long LongCount<TSource>(
   this IEnumerable<TSource> source);
public static long LongCount<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate);
```

Ambas variantes funcionan del mismo modo que sus homólogas en `Count()`, pero en el caso de `LongCount()` para contar los elementos se utiliza un entero largo.

La siguiente consulta devuelve la cantidad de futbolistas alemanes que toman parte en la Liga Española:

```
int s31 = (from f in DatosFutbol.Futbolistas
           where f.CodigoPaisNac == "DE"
           select f).Count();
```

10.10.2. Los operadores Max() y Min()

Los operadores `Max()` y `Min()` permiten determinar el máximo y el mínimo valor, respectivamente, de una secuencia de elementos.

```
public static tipo_Numérico Max(
   this IEnumerable<tipo_Numérico> source);
public static TSource Max<TSource>(
   this IEnumerable<TSource> source);
public static tipo_Numérico Max<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, tipo_Numérico> selector);
public static TResult Max<TSource, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, TResult> selector);

public static tipo_Numérico Min(
   this IEnumerable<tipo_Numérico> source);
public static TSource Min<TSource>(
   this IEnumerable<TSource> source);
public static tipo_Numérico Min<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, tipo_Numérico> selector);
public static TResult Min<TSource, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, TResult> selector);
```

El **tipo _ Numérico** puede ser cualquiera de los tipos: **int, int?, long, long?, float, float?, double, double?, decimal** y **decimal?**

El operador `Max()` enumera la secuencia de entrada, llamando para cada elemento a la función de selección y obteniendo el máximo de los valores resultantes. `Min()` funciona de la misma manera, pero obteniendo el mínimo. Si no se especifica una función de selección, se determina el máximo (mínimo) de los propios elementos de la secuencia. Los valores se comparan utilizando su implementación de la interfaz `IComparable<TSource>`, o en caso de que los valores no implementen esa interfaz, de la interfaz no genérica `IComparable`. Las implementaciones para los tipos numéricos son optimizaciones de los operadores genéricos más generales.

Si el tipo de los elementos es un tipo valor no anulable y la secuencia de entrada es vacía, se lanzará una excepción `InvalidOperationException`. Si el tipo es un tipo referencia o un tipo valor anulable y la secuencia de entrada es vacía o contiene únicamente valores nulos, entonces el resultado será **null.**

La siguiente consulta produce una secuencia con los códigos de los clubes de la Liga y las edades máximas de sus plantillas:

```
var s32 = (from f in DatosFutbol.Futbolistas
           group f.Edad by f.CodigoClub into grupos
           select new { grupos.Key, Max = grupos.Max() });
```

El siguiente ejemplo muestra cómo determinar la máxima cantidad de veces que se repite una palabra cualquiera en un texto:

```
string letra =
   // "Man in the wilderness", STYX 1976
   @"Sometimes I feel like a man in the wilderness
      Like a lonely soldier of a war
      Sent away to die never quite knowing why
      Sometimes it makes no sense at all";

var palabras = letra.Split(new char[] { ' ', '\t', '\r', '\n' },
      StringSplitOptions.RemoveEmptyEntries);

var maxRepeticiones =
      (from p in palabras
       group p by p.ToUpper() into pares
       select new { pares.Key, Cant = pares.Count() }).
   Max(p => p.Cant);
```

10.10.3. El operador Sum()

El operador `Sum()` calcula la suma de una secuencia de valores numéricos.

```
public static tipo_Numérico Sum(
   this IEnumerable<tipo_Numérico> source);
public static tipo_Numérico Sum<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, tipo_Numérico> selector);
```

El **tipo_Numérico** debe ser alguno de los tipos: **int**, **int?**, **long**, **long?**, **float**, **float?**, **double**, **double?**, **decimal** y **decimal?**

El operador `Sum()` enumera la secuencia de entrada, llamando a la función de selección para cada elemento y calculando la suma de los valores resultantes. Si no se especifica una función de selección, se calcula la suma de los propios elementos. Para los tipos **int**, **int?**, **long**, **long?**, **decimal** y **decimal?**, si un resultado intermedio a la hora de calcular la suma no puede ser representado utilizando ese tipo, se producirá una excepción `OverflowException`. Para **float**, **float?**, **double** y **double?**, se devolverá un valor infinito (positivo o negativo) si un resultado intermedio es demasiado grande para representarse utilizando un **double**.

El operador `Sum()` devuelve cero para una secuencia vacía. Además, ignora los valores **null** (algo posible en los casos en que el tipo de los elementos sea un tipo anulable).

A continuación, se muestra un ejemplo en el que se calcula la suma de los cuadrados de un array de números:

```
double[] arr3 = { 2.75, 4.15, 6.25 };

var sumaCuadrados = arr3.Sum(x => x * x);
```

10.10.4. El operador Average()

El operador `Average()` calcula la media de una secuencia de valores numéricos.

```
public static tipo_Resultado Average(
   this IEnumerable<tipo_Numérico> source);
public static tipo_Resultado Average<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, tipo_Numérico> selector);
```

El **tipo_Numérico** debe ser alguno de los tipos: **int**, **int?**, **long**, **long?**, **float**, **float?**, **double**, **double?**, **decimal** y **decimal?** Cuando el **tipo_Numérico** es **int** o **long**, **tipo_Resultado** es **double**. Cuando **tipo_Numérico** es **int?** o **long?**, **tipo_Resultado** es **double?** En el resto de los casos, **tipo_Resultado** será el mismo **tipo_Numérico**.

El operador `Average()` enumera la secuencia de entrada, llamando a la función de selección para cada elemento y calculando la media de los valores resultantes.

Si no se especifica una función de selección, se calcula la media de los propios elementos. Para los tipos **int**, **int?**, **long** y **long?**, si un resultado intermedio a la hora de calcular la suma no puede ser representado en un **long**, se producirá una excepción `OverflowException`. Para los tipos **decimal** y **decimal?**, si un resultado intermedio a la hora de calcular la suma no puede ser representado en un **decimal**, se producirá una excepción `OverflowException`.

El operador `Average()`para **int?**, **long?**, **float?**, **double?** y **decimal?** devuelve **null** si la secuencia de entrada es vacía o contiene únicamente valores nulos. Para el resto de los tipos, se produce una excepción `InvalidOperationException` si la secuencia de entrada es vacía.

La siguiente consulta (con subconsulta) produce una secuencia con los nombres de los clubes de fútbol y la media de edad de sus jugadores:

```
var s33 = (from c in DatosFutbol.Clubes
           orderby c.Nombre
           select new {
            c.Nombre,
            MediaEdad =
               (from f in DatosFutbol.Futbolistas
                where f.CodigoClub == c.Codigo
                select f.Edad).Average() });
```

10.10.5. El operador Aggregate()

El operador `Aggregate()` aplica una función de consolidación a lo largo de una secuencia. Ofrece tres sobrecargas:

```
public static TSource Aggregate<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, TSource, TSource> func);
public static TAccumulate Aggregate<TSource, TAccumulate>(
   this IEnumerable<TSource> source,
   TAccumulate seed,
   Func<TAccumulate, TSource, TAccumulate> func);
public static TResult Aggregate<TSource, TAccumulate, TResult>(
   this IEnumerable<TSource> source,
   TAccumulate seed,
   Func<TAccumulate, TSource, TAccumulate> func,
   Func<TAccumulate, TResult> resultSelector);
```

Las variantes de `Aggregate()` con un valor de semilla (`seed`) comienzan asignando el valor de ésta a un acumulador interno. Entonces enumeran la secuencia de entrada, calculando repetidamente el próximo valor del acumulador llamando a la función especificada (`func`) con el valor actual del acumulador como primer argumento y el elemento actual de la secuencia de entrada como segundo argumento. La

variante que incluye un selector de resultado (`resultSelector`) pasa a esta función el valor final del acumulador y devuelve el valor resultante. La variante que no incluye un selector de resultado simplemente devuelve el valor final del acumulador.

La variante del operador `Aggregate()` que no tiene una semilla utiliza el primer elemento de la secuencia de entrada como valor de la semilla, y a partir del segundo elemento de la secuencia se comporta tal y como se ha descrito anteriormente. Si la secuencia de entrada es vacía, esta variante de `Aggregate()` lanza una excepción `InvalidOperationException`.

El siguiente ejemplo muestra una nueva manera de calcular el factorial de un número positivo en C# 3.0:

```
public static int Factorial(int n)
{
   Debug.Assert(n > 0);
   return Enumerable.Range(1, n).
      Aggregate(
         1,
         (x, i) => x * i);
}
```

CAPÍTULO 11

El patrón LINQ

En este capítulo estudiaremos las diferentes vías por las cuales un desarrollador puede crear su "proveedor LINQ", entendiendo como tal a cualquier mecanismo que haga posible aplicar la sintaxis de las expresiones de consulta a un tipo de datos dado. Después de presentar varios casos relativamente sencillos, formalizaremos la definición del **patrón LINQ** (que ya le será familiar, dado que lo hemos mencionado con bastante frecuencia en los capítulos anteriores), el mecanismo general que los creadores de la tecnología proponen implementar a los fabricantes de extensiones de LINQ. Por último, presentaremos una interfaz heredera de `IEnumerable<T>` llamada `IQueryable<T>`, que en ciertos casos ofrece mayores posibilidades que su ancestro para la implementación de tales extensiones.

11.1. ACERCANDO LINQ A NUEVOS TIPOS DE DATOS

En muchos casos, para poder aplicar las consultas integradas a los objetos de un cierto tipo (o a los objetos que gestiona una cierta clase) basta con definir uno o más **puntos de entrada** a partir de los cuales la tecnología básica incorporada en .NET Framework (LINQ to Objects) pueda comenzar a ser aplicada. Éste ha sido, por ejemplo, el enfoque aplicado en LINQ to XML (Capítulo 12), donde se ha definido un conjunto de iteradores (implementados como métodos extensores) que producen subconjuntos de los nodos que componen un documento XML como secuencias `IEnumerable<T>`; una vez se llama a cualquiera de esos métodos, se obtiene un generador de nodos XML al que se pueden aplicar las implementaciones estándar de los operadores `Where()`, `OrderBy()`, etc.

Considere, por ejemplo, los mecanismos de navegación por el sistema de ficheros que estudiamos en el Capítulo 3. En principio, nada nos impide aplicar la sintaxis de expresiones de consulta al resultado de la llamada a `Directory.GetFiles()`: es un array, y como tal implementa `IEnumerable<T>`. Pero es un array de cadenas, y nuestras posibilidades de selección, ordenación, etc. estarán limitadas a operar sobre los nombres de ficheros:

DIRECTORIO DEL CODIGO: EJEMPLO11 _ 01

```
var c1 =
   from f in System.IO.Directory.GetFiles(
      "C:\\Users\\Octavio\\Pictures", "*.jpg",
      SearchOption.AllDirectories)
   where f.ToUpper().Contains("DIANA")
   orderby f
   select f.ToUpper();
```

La segunda deficiencia de este mecanismo es que, como ya hemos mencionado, no hace uso de la evaluación demorada: todos los nombres de ficheros son recolectados de antemano por `GetFiles()`.

El uso del método `GetFiles()` de la clase `System.IO.DirectoryInfo` (que no es estático como el de `Directory`), nos daría más opciones a la hora de expresar consultas sobre el contenido de un sistema de ficheros:

```
DirectoryInfo dirInfo =
   new System.IO.DirectoryInfo("C:\\Users\\Octavio\\Pictures");

var c2 =
   from f in dirInfo.GetFiles(".jpg",
      SearchOption.AllDirectories)
   where f.Length > 10240
   orderby f.CreationTime descending
   select f.FullName.ToUpper();
```

pero seguiría adoleciendo de la deficiencia de que la llamada es "golosa" (*greedy*) en vez de perezosa.

A diferencia de las anteriores, la clase `FileSystemEnumerator` presentada en el Capítulo 3 resuelve con solvencia las dos limitaciones planteadas: genera una secuencia de objetos de tipo `FileInfo` que incluyen toda la información relevante sobre los ficheros (tamaño, atributos, etc.) y funciona como un iterador, produciendo esos objetos en la medida que van siendo necesitados.

Según mi humilde opinión, se podría perfectamente encapsular esta clase dentro de un ensamblado y promoverla con un nombre que suene bien como **LINQ to FileSystem.** Tal vez lo único que faltaría sería crear un método extensor para la clase `DirectoryInfo`, de modo que pudiera utilizarse como un método más de ésta:

```
public static partial class Extensiones
{
   public static IEnumerable<FileInfo> GetFileInfos(
```

Continúa

```
      this DirectoryInfo dir,
      string fileTypesToMatch,
      bool includeSubDirs)
   {
      using (FileSystemEnumerator fse =
         new FileSystemEnumerator(
            dir.FullName,
            fileTypesToMatch,
            includeSubDirs))
      {
         foreach (FileInfo fi in fse.Matches())
            yield return fi;
      }
   }
}
```

Una vez puesto en ámbito el espacio de nombres al que pertenece la clase que define el método extensor:

```
using PlainConcepts.Linq;
```

este método podrá utilizarse como uno más de la clase `DirectoryInfo`:

```
var c3 =
   from f in dirInfo.GetFileInfos("*.jpg", true)
   where f.Length > 10240
   orderby f.CreationTime descending
   select f.FullName.ToUpper();

foreach (var c in c3)
   Console.WriteLine(c);
```

11.1.1. LINQ to Pipes

Como segundo ejemplo de "activación para LINQ" de clases propias o de la librería de .NET Framework, presentaremos la implementación de una extensión que permitirá aplicar la sintaxis de operadores de consulta a los datos que se reciben a través de una canalización anónima o con nombre.

Las canalizaciones (*pipes*) son un mecanismo de comunicación entre procesos que se utiliza desde hace mucho en el mundo DOS/Windows, pero sólo con la aparición de .NET Framework 3.5 tenemos la posibilidad de utilizarlos en nuestras aplicaciones .NET sin tener que recurrir a llamadas a la plataforma. Las canalizaciones pueden ser anónimas, concebidas para la comunicación entre un proceso padre y un proceso hijo que residen en la misma máquina, o con nombre, que ofrecen mucha más funcionalidad y pueden ser utilizadas a través de una red. Específicamente, las canalizaciones con nombre ofrecen comunicación orientada a mensajes.

El siguiente programa muestra un ejemplo de un servidor que envía mensajes a través de una canalización llamada `CS3`:

DIRECTORIO DEL CODIGO: EJEMPLO11 _ 02

```
class Program
{
   static void Main(string[] args)
   {
      // *** SERVIDOR
      UTF8Encoding encoding = new UTF8Encoding();

      using (NamedPipeServerStream pipeStream =
         new NamedPipeServerStream("CS3", PipeDirection.Out))
      {
         pipeStream.WaitForConnection();
         // envío de mensajes
         for (int i = 0; i < 100; i++)
         {
            string msg = i.ToString();
            byte[] bytes = encoding.GetBytes(msg);
            pipeStream.Write(bytes, 0, bytes.Length);
         }
      }
   }
}
```

El cliente, por su parte, compone los mensajes que van llegando a su cola:

```
// *** CLIENTE
Decoder decoder = Encoding.UTF8.GetDecoder();
Byte[] bytes = new Byte[10];
Char[] chars = new Char[10];

using (NamedPipeClientStream pipeStream =
   new NamedPipeClientStream(".", "CS3", PipeDirection.InOut))
   // recepción de mensajes:
   // "." representa al equipo local
   // "CS3" es el nombre de la canalización
{
   pipeStream.Connect();
   pipeStream.ReadMode = PipeTransmissionMode.Message;

   int numBytes;
   do
   {
```

Continúa

```
      string message = "";
      do
      {
         numBytes = pipeStream.Read(bytes, 0, bytes.Length);
         int numChars =
            decoder.GetChars(bytes, 0, numBytes, chars, 0);
         message += new String(chars, 0, numChars);
      } while (!pipeStream.IsMessageComplete);
      decoder.Reset();
      Console.WriteLine(message);
   } while (numBytes != 0);
```

¿Y si quisiéramos poder leer en el proceso cliente las líneas, filtrarlas, ordenarlas, etc.? Nada más fácil que crear un método extensor para el flujo de canalización:

```
public static partial class Extensions
{
   public static IEnumerable<string> GetMessages(
      this NamedPipeClientStream pipeStream)
      {
      Decoder decoder = Encoding.UTF8.GetDecoder();
      Byte[] bytes = new Byte[10];
      Char[] chars = new Char[10];

      pipeStream.Connect();
      pipeStream.ReadMode = PipeTransmissionMode.Message;

      int numBytes;
      do
      {
         string message = "";
         do
         {
            numBytes = pipeStream.Read(bytes, 0,
               bytes.Length);
            int numChars = decoder.GetChars(bytes, 0,
               numBytes, chars, 0);
            message += new String(chars, 0, numChars);
         } while (!pipeStream.IsMessageComplete);
         decoder.Reset();
         // *** producir el mensaje
         yield return message;
      } while (numBytes != 0);
   }
}
```

Disponiendo de esa "puerta de entrada" al mundo de las consultas integradas, podremos tratar los datos que llegan de las canalizaciones así:

```
using (NamedPipeClientStream pipeStream =
   new NamedPipeClientStream(".", "CS3",
      PipeDirection.InOut))
{
   var entrada = from s in pipeStream.GetMessages()
                 where string.Compare(s, "3") < 0
                 orderby s
                 select s;
   foreach (var s in entrada)
      Console.WriteLine(s);
}
```

11.2. EL PATRÓN DE EXPRESIONES DE CONSULTA

En situaciones más complejas, es posible que el enfoque presentado anteriormente no sea suficiente, y se necesite ofrecer una implementación personalizada de los operadores de consulta para un cierto tipo de datos, generalmente (pero no necesariamente) una colección de un tipo específico. Ya hemos estudiado cómo LINQ se basa en una arquitectura abierta y 100% programable, lo que permite a un desarrollador experimentado crear sus propias versiones de los operadores de consulta para un tipo cualquiera.

El patrón de expresiones de consulta (**patrón LINQ**) no es más que un conjunto de prototipos de clases con sus respectivos métodos que la especificación de C# 3.0 propone implementar a los desarrolladores que necesiten crear su propio proveedor de expresiones de consulta. No es más que una recomendación a fin de cuentas, porque la flexibilidad que ofrece 3.0 a este respecto hace posible implementar el soporte para las expresiones de consulta mediante clases o interfaces genéricas o no genéricas, mediante métodos de instancia o métodos extensores… (y aún hay más).

El patrón recomendado para una clase genérica `C<T>` que dé soporte al patrón de expresiones de consulta es el siguiente:

```
class C
{
   public C<T> Cast<T>();
}

class C<T>
{
   public C<T> Where(Func<T, bool> predicate);
   public C<U> Select<U>(Func<T, U> selector);
```

Continúa

```
   public C<V> SelectMany<U, V>(Func<T, C<U>> selector,
      Func<T, U, V> resultSelector);
   public C<V> Join<U, K, V>(C<U> inner,
      Func<T, K> outerKeySelector,
      Func<U, K> innerKeySelector,
      Func<T, U, V> resultSelector);
   public C<V> GroupJoin<U, K, V>(C<U> inner,
      Func<T, K> outerKeySelector,
      Func<U, K> innerKeySelector,
      Func<T, C<U>, V> resultSelector);
   public O<T> OrderBy<K>(Func<T, K> keySelector);
   public O<T> OrderByDescending<K>(Func<T, K> keySelector);
   public C<G<K, T>> GroupBy<K>(Func<T, K> keySelector);
   public C<G<K, E>> GroupBy<K, E>(Func<T, K> keySelector,
      Func<T, E> elementSelector);
}

class O<T> : C<T>
{
   public O<T> ThenBy<K>(Func<T, K> keySelector);
   public O<T> ThenByDescending<K>(Func<T, K> keySelector);
}

class G<K, T> : C<T>
{
   public K Key { get; }
}
```

Sobre este conjunto de clases debemos hacer las siguientes observaciones:

- El patrón sugiere implementar una variante no genérica de la clase `C`, y dotarla del operador de consulta estándar `Cast<T>()`, a través del cual se hará posible la utilización sobre esa variante no genérica de la sintaxis de expresiones de consulta.
- Los operadores `OrderBy()` y `OrderByDescending()` deben producir instancias de una clase `O<T>` descendiente de `C<T>`, en la que solamente es necesario definir los operadores `ThenBy()` y `ThenByDescending()`.
- La clase `G<K, T>` se producirá como resultado de la agrupación de los elementos de la colección original `C<T>` por una clave de tipo `K`.

A continuación, se presenta una clase `ClaseLinq<T>` que hereda de `List<T>` e implementa el patrón de expresiones de consulta completo, intentando imitar el funcionamiento de los operadores de consulta estándar. Observe que en el código de prueba de esta clase que acompaña al libro no se pone en ámbito la implementación predeterminada de LINQ to Objects; de esta manera, las consultas integradas contra colecciones de tipo `ClaseLinq<T>` utilizarán nuestras definiciones, mientras que el intento de utilizar la sintaxis de consultas contra secuencias de cualquier otro tipo producirá un error de compilación.

Un buen ejercicio que le ayudará a reforzar todos sus conocimientos sobre C# 3.0 y LINQ es intentar implementar esta clase sin consultar el libro, para luego comparar su versión con la mía. Tenga en cuenta que para reducir el tamaño del listado, he suprimido los controles de posibles argumentos nulos en llamadas a los métodos.

DIRECTORIO DEL CODIGO: EJEMPLO11 _ 03

```
namespace PlainConcepts.LinqProviders
{
   class ClaseLinq<T>: List<T>
   {
      public ClaseLinq<T> Where(Func<T, bool> predicate)
      {
         ClaseLinq<T> result = new ClaseLinq<T>();
         foreach (T t in this)
            if (predicate(t))
               result.Add(t);
         return result;
      }

      public ClaseLinq<U> Select<U>(Func<T, U> selector)
      {
         ClaseLinq<U> result = new ClaseLinq<U>();
         foreach (T t in this)
            result.Add(selector(t));
         return result;
      }

      public ClaseLinq<V> SelectMany<U, V>(
         Func<T, ClaseLinq<U>> selector,
         Func<T, U, V> resultSelector)
      {
         ClaseLinq<V> result = new ClaseLinq<V>();
         foreach (T t in this)
         {
            ClaseLinq<U> lista2 = selector(t);
            foreach (U u in lista2)
               result.Add(resultSelector(t, u));
         }
         return result;
      }

      public ClaseLinq<V> Join<U, K, V>(ClaseLinq<U> inner,
         Func<T, K> outerKeySelector,
         Func<U, K> innerKeySelector,
         Func<T, U, V> resultSelector)
```

Continúa

```
{
   Dictionary<K, List<U>> dict =
      new Dictionary<K,List<U>>();
   foreach (U u in inner)
   {
      K clave2 = innerKeySelector(u);
      if (dict.ContainsKey(clave2))
         dict[clave2].Add(u);
      else
      {
         List<U> list2 = new List<U>();
         list2.Add(u);
         dict.Add(clave2, list2);
      }
   }
   ClaseLinq<V> result = new ClaseLinq<V>();
   foreach (T t in this)
   {
      K clave1 = outerKeySelector(t);
      if (dict.ContainsKey(clave1))
         foreach (U u in dict[clave1])
            result.Add(resultSelector(t, u));
   }
   return result;
}

public ClaseLinq<V> GroupJoin<U, K, V>(ClaseLinq<U> inner,
   Func<T, K> outerKeySelector,
   Func<U, K> innerKeySelector,
   Func<T, ClaseLinq<U>, V> resultSelector)
{
   Dictionary<K, ClaseLinq<U>> dict =
      new Dictionary<K, ClaseLinq<U>>();
   foreach (U u in inner)
{
   K clave2 = innerKeySelector(u);
   if (dict.ContainsKey(clave2))
      dict[clave2].Add(u);
   else
   {
      ClaseLinq<U> list2 = new ClaseLinq<U>();
      list2.Add(u);
      dict.Add(clave2, list2);
   }
}
ClaseLinq<V> result = new ClaseLinq<V>();
foreach (T t in this)
```

Continúa

```
      {
         K clave1 = outerKeySelector(t);
         if (dict.ContainsKey(clave1))
            result.Add(resultSelector(t, dict[clave1]));
         else
            result.Add(resultSelector(t,
               new ClaseLinq<U>()));
      }
      return result;
   }

   public ClaseLinqOrdenada<T> OrderBy<K>(
      Func<T, K> keySelector)
   {
      ClaseLinqOrdenada<T> result =
         new ClaseLinqOrdenada<T>();
      foreach (T t in this)
         result.Add(t);
      // ordenar
      for (int i = 0; i < result.Count - 1; i++)
         for (int j = i + 1; j < result.Count; j++)
            if (Comparer<K>.Default.Compare(
               keySelector(result[i]),
               keySelector(result[j])) > 0)
            {
               T tmp = result[i];
               result[i] = result[j];
               result[j] = tmp;
            }
      // guardar relación entre cada elemento y su predecesor
      // esta info la utilizarán los ThenBy() subsiguientes
      if (result.Count > 0)
      {
         result.igualAlAnterior.Add(false);
         for (int i = 1; i < result.Count; i++)
            result.igualAlAnterior.Add(
               Comparer<K>.Default.Compare(
                  keySelector(result[i]),
                  keySelector(result[i - 1])) == 0);
      }
   return result;
   }

   public ClaseLinqOrdenada<T> OrderByDescending<K>(
      Func<T, K> keySelector)
   {
      ClaseLinqOrdenada<T> result =
```

Continúa

```
      new ClaseLinqOrdenada<T>();
   foreach (T t in this)
      result.Add(t);
   // ordenar
   for (int i = 0; i < result.Count - 1; i++)
      for (int j = i + 1; j < result.Count; j++)
         if (Comparer<K>.Default.Compare(
            keySelector(result[i]),
            keySelector(result[j])) < 0)
         {
            T tmp = result[i];
            result[i] = result[j];
            result[j] = tmp;
         }
   // guardar relación entre cada elemento y su predecesor
   // esta info la utilizarán los ThenBy() subsiguientes
   if (result.Count > 0)
   {
      result.igualAlAnterior.Add(false);
      for (int i = 1; i < result.Count; i++)
         result.igualAlAnterior.Add(
            Comparer<K>.Default.Compare(
               keySelector(result[i]),
               keySelector(result[i - 1])) == 0);
   }
   return result;
}

public ClaseLinq<GrupoLinq<K, T>> GroupBy<K>(
   Func<T, K> keySelector)
{
   Dictionary<K, ClaseLinq<T>> dict =
      new Dictionary<K, ClaseLinq<T>>();
   foreach (T t in this)
   {
      K clave = keySelector(t);
      if (dict.ContainsKey(clave))
         dict[clave].Add(t);
      else
      {
         ClaseLinq<T> list2 = new ClaseLinq<T>();
         list2.Add(t);
         dict.Add(clave, list2);
      }
   }
   ClaseLinq<GrupoLinq<K, T>> result =
      new ClaseLinq<GrupoLinq<K, T>>();
```

Continúa

```
      foreach (K k in dict.Keys)
      {
         GrupoLinq<K, T> g = new GrupoLinq<K, T>(k);
         foreach (T t in dict[k])
            g.Add(t);
         result.Add(g);
      }
      return result;
   }

   public ClaseLinq<GrupoLinq<K, E>> GroupBy<K, E>(
      Func<T, K> keySelector,
      Func<T, E> elementSelector)
   {
      Dictionary<K, ClaseLinq<E>> dict =
         new Dictionary<K, ClaseLinq<E>>();
      foreach (T t in this)
      {
         E elem = elementSelector(t);
         K clave = keySelector(t);
         if (dict.ContainsKey(clave))
            dict[clave].Add(elem);
         else
         {
            ClaseLinq<E> list2 = new ClaseLinq<E>();
            list2.Add(elem);
            dict.Add(clave, list2);
         }
      }
      ClaseLinq<GrupoLinq<K, E>> result =
         new ClaseLinq<GrupoLinq<K, E>>();
      foreach (K k in dict.Keys)
      {
         GrupoLinq<K, E> g = new GrupoLinq<K, E>(k);
         foreach (E e in dict[k])
              g.Add(e);
         result.Add(g);
      }
      return result;
      }
   }

   class ClaseLinqOrdenada<T> : ClaseLinq<T>
   {
      internal List<bool> igualAlAnterior = new List<bool>();

      public ClaseLinqOrdenada<T> ThenBy<K>(
```

Continúa

```
      Func<T, K> keySelector)
   {
      ClaseLinqOrdenada<T> result =
         new ClaseLinqOrdenada<T>();
      for (int i = 0; i < this.Count; i++)
      {
         result.Add(this[i]);
         result.igualAlAnterior.Add(this.igualAlAnterior[i]);
      }

      if (result.Count > 0)
      {
         int i = 0;
         while (i < result.Count)
         {
            int j = i + 1;
            if (j < result.Count &&
               result.igualAlAnterior[j])
            {
               while (j < result.Count &&
                  result.igualAlAnterior[j])
               {
                  int res = Comparer<K>.Default.Compare(
                    keySelector(result[i]),
                    keySelector(result[j]));
                  if (res > 0)
                  {
                    T tmp = result[i];
                    result[i] = result[j];
                    result[j] = tmp;
                  }
                  j++;
               }
               result.igualAlAnterior[i + 1] =
                  Comparer<K>.Default.Compare(
                    keySelector(result[i]),
                    keySelector(result[i + 1])) == 0;
               }
            i++;
         }
      }

      return result;
   }

   public ClaseLinqOrdenada<T> ThenByDescending<K>(
      Func<T, K> keySelector)
```

Continúa

```
    {
      ClaseLinqOrdenada<T> result =
        new ClaseLinqOrdenada<T>();
      for (int i = 0; i < this.Count; i++)
      {
        result.Add(this[i]);
        result.igualAlAnterior.Add(this.igualAlAnterior[i]);
      }

      if (result.Count > 0)
      {
        int i = 0;
        while (i < result.Count)
        {
          int j = i + 1;
          if (j < result.Count &&
            result.igualAlAnterior[j])
          {
            while (j < result.Count &&
              result.igualAlAnterior[j])
            {
              int res = Comparer<K>.Default.Compare(
                keySelector(result[i]),
                keySelector(result[j]));
              if (res < 0)
              {
                T tmp = result[i];
                result[i] = result[j];
                result[j] = tmp;
              }
              j++;
            }
            result.igualAlAnterior[i + 1] =
              Comparer<K>.Default.Compare(
                keySelector(result[i]),
                keySelector(result[i + 1])) == 0;
          }
          i++;
        }
      }

      return result;
    }
  }

  class GrupoLinq<K, T> : ClaseLinq<T>
  {
```

Continúa

```
      private K key;

      public K Key
      {
         get
         {
            return key;
         }
      }

      public GrupoLinq(K k)
      {
         key = k;
      }
   }
}
```

Otro ejercicio interesante consiste en sustituir las implementaciones (correctas) que se ofrecen en esta clase por otras con resultados no tan correctos. Por ejemplo, ¿qué pasaría si sustituimos el método `Where()` que ofrece `ClaseLinq<T>` por el siguiente?

```
private int cont = 0;

public ClaseLinq<T> Where(Func<T, bool> predicate)
{
   ClaseLinq<T> result = new ClaseLinq<T>();
   foreach (T t in this)
      if (predicate(t))
      {
         if (cont % 3 == 0) result.Add(t);
         cont++;
      }
   return result;
}
```

11.3. UNA IMPLEMENTACIÓN ALTERNATIVA A LINQ TO OBJECTS

A diferencia del proveedor anterior, que habilita la sintaxis de consultas para una clase específica, el que se presenta a continución se basa simplemente en la definición de una clase estática que implementa las firmas del patrón LINQ como métodos extensores de la propia interfaz `IEnumerable<T>`. Debido a esto, sólo podrá utilizarse como implementación alternativa a la estándar: si se importan simultáneamente `System`.

`Linq` y `PlainConcepts.LinqProviders`, el compilador se quejará de que las llamadas a los métodos del patrón LINQ son ambiguas.

La implementación que presentamos aquí se apoya en la estándar, añadiendo simplemente la capacidad de registrar las llamadas a cada uno de los métodos extensores en un fichero de bitácora representado por una propiedad estática llamada `Log`. Por esta razón, la clase que contiene los métodos extensores se ha denominado `LoggingEnumerable`. Esta clase podría ser útil para conocer qué métodos del patrón LINQ van a ser llamados para una expresión de consulta concreta. A continuación presentamos su código:

```
namespace PlainConcepts.LinqProviders
{
   using PlainConcepts.LinqProviders;

   public static class LoggingEnumerable
   {
      // *** propiedad adicional

      public static TextWriter Log
      {
         set { log = value; }
      }
      private static TextWriter log = null;

      // *** métodos extensores para IEnumerable<T>

      public static IEnumerable<T> Where<T>(
         this IEnumerable<T> self,
         Func<T, bool> predicate)
      {
         if (log != null) log.WriteLine("WHERE/2");
         return Enumerable.Where(self, predicate);
      }

      public static IEnumerable<U> Select<T, U>(
         this IEnumerable<T> self,
         Func<T, U> selector)
      {
         if (log != null) log.WriteLine("SELECT/2");
         return Enumerable.Select(self, selector);
      }

      public static IEnumerable<V> SelectMany<T, U, V>(
         this IEnumerable<T> self,
         Func<T, IEnumerable<U>> selector,
         Func<T, U, V> resultSelector)
```

Continúa

```
{
   if (log != null) log.WriteLine("SELECTMANY/3");
   return Enumerable.SelectMany(self, selector,
      resultSelector);
}

public static IEnumerable<V> Join<T, U, K, V>(
   this IEnumerable<T> self,
   IEnumerable<U> inner,
   Func<T, K> outerKeySelector,
   Func<U, K> innerKeySelector,
   Func<T, U, V> resultSelector)
{
   if (log != null) log.WriteLine("JOIN/5");
   return Enumerable.Join(self, inner,
      outerKeySelector, innerKeySelector, resultSelector);
}

public static IEnumerable<V> GroupJoin<T, U, K, V>(
   this IEnumerable<T> self,
   IEnumerable<U> inner,
   Func<T, K> outerKeySelector,
   Func<U, K> innerKeySelector,
   Func<T, IEnumerable<U>, V> resultSelector)
{
   if (log != null) log.WriteLine("GROUPJOIN/5");
   return Enumerable.GroupJoin(self, inner,
      outerKeySelector, innerKeySelector, resultSelector);
}

public static IOrderedEnumerable<T> OrderBy<T, K>(
   this IEnumerable<T> self,
   Func<T, K> keySelector)
{
   if (log != null) log.WriteLine("ORDERBY/2");
   return Enumerable.OrderBy(self, keySelector);
}

public static IOrderedEnumerable<T> OrderByDescending<T, K>(
   this IEnumerable<T> self,
   Func<T, K> keySelector)
{
   if (log != null) log.WriteLine("ORDERBYDESCENDING/2");
   return Enumerable.OrderByDescending(self, keySelector);
}

public static IOrderedEnumerable<T> ThenBy<T, K>(
   this IOrderedEnumerable<T> self,
   Func<T, K> keySelector)
```

Continúa

```
      {
         if (log != null) log.WriteLine("THENBY/2");
         return Enumerable.ThenBy(self, keySelector);
      }

      public static IOrderedEnumerable<T> ThenByDescending<T, K>(
         this IOrderedEnumerable<T> self,
         Func<T, K> keySelector)
      {
         if (log != null) log.WriteLine("THENBYDESCENDING/2");
         return Enumerable.ThenByDescending(self, keySelector);
      }

      public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
         this IEnumerable<T> self,
         Func<T, K> keySelector)
      {
         if (log != null) log.WriteLine("GROUPBY/2");
         return Enumerable.GroupBy(self, keySelector);
      }

      public static IEnumerable<IGrouping<K, E>> GroupBy<T, K,
E>(
         this IEnumerable<T> self,
         Func<T, K> keySelector,
         Func<T, E> elementSelector)
      {
         if (log != null) log.WriteLine("GROUPBY/3");
         return Enumerable.GroupBy(self, keySelector,
            elementSelector);
      }
   }
}
```

Una vez que tengamos esa clase a nuestro alcance, podremos ponerla en ámbito en una aplicación cliente y probarla con varias consultas integradas:

```
LoggingEnumerable.Log = Console.Out;   // salida a la consola

string[] nombres = {
   "Kerry", "Steve", "Phil", "Dave", "Rich", "Robbie" };

var erres = from n in nombres
            let pos = n.IndexOf('e')
            where pos >= 0
```

Continúa

```
            orderby pos, n
            select n.ToUpper();

foreach (string erre in erres)
   Console.WriteLine(erre);

int[] numeros = { 27, 43, 52, 87, 99, 45, 72, 29, 61, 58, 94 };

ParDigito[] digitos =
   {
      new ParDigito { Digito = 0, Nombre = "Cero" },
      new ParDigito { Digito = 1, Nombre = "Uno" },
      new ParDigito { Digito = 2, Nombre = "Dos" },
      new ParDigito { Digito = 3, Nombre = "Tres" },
      new ParDigito { Digito = 4, Nombre = "Cuatro" },
      new ParDigito { Digito = 5, Nombre = "Cinco" },
      new ParDigito { Digito = 6, Nombre = "Seis" },
      new ParDigito { Digito = 7, Nombre = "Siete" },
      new ParDigito { Digito = 8, Nombre = "Ocho" },
      new ParDigito { Digito = 9, Nombre = "Nueve" },
   };

var grupos =from n in numeros
            let terminacion = n % 10
            join d in digitos on terminacion equals d.Digito
            orderby d.Digito
            group n by d.Nombre;

foreach (var g in grupos)
{
   Console.WriteLine(g.Key);
   foreach (var elem in g)
      Console.WriteLine("    " + elem);
}
```

Los resultados que se producirán en la consola son los siguientes:

```
SELECT/2
WHERE/2
ORDERBY/2
THENBY/2
SELECT/2
KERRY
STEVE
DAVE
ROBBIE
```

Continúa

```
SELECT/2
JOIN/5
ORDERBY/2
GROUPBY/3
Uno
   61
Dos
   52
   72
Tres
   43
Cuatro
   94
Cinco
   45
Siete
   27
   87
Ocho
   58
Nueve
   99
   29
```

11.4. LA INTERFAZ IQUERYABLE<T>

Tan pronto se pensó en implementar una extensión de LINQ para consultar bases de datos relacionales, se hizo evidente que el mecanismo basado en `IEnumerable<T>` no era el más adecuado. Cualquier implementación de los operadores de consulta que estuviera basada en la navegación de múltiples cursores, incluso utilizando evaluación demorada, sería desastrosa desde el punto de vista del rendimiento. Los creadores de la tecnología se dieron cuenta de que la única vía correcta de implementar las consultas integradas contra un almacén relacional consiste en ir "capturando" en cada paso las especificaciones de cada una de las cláusulas `where`, `orderby`, etc. para solamente después analizada toda la cadena de llamadas a métodos extensores componer una única sentencia SQL a enviar al motor de base de datos.

Para este fin se definió una nueva interfaz, `IQueryable<T>`, que hereda de `IEnumerable<T>` pero viene dotada de un conjunto de métodos extensores (cuya implementación básica está contenida en la clase estática `System.Linq.Queryable`, del mismo modo que los métodos extensores para `IEnumerable<T>` se implementan en `System.Linq.Enumerable`) que funcionan de un modo totalmente diferente a como lo hacen los operadores de LINQ to Objects, y que es más adecuado para el caso en que la fuente de datos a la que se desea acceder sea "remota", como ocurre en LINQ to SQL. En ese sentido, podríamos considerar a `IQueryable<T>` un proveedor LINQ personalizado, aunque ello no es totalmente correcto, dado que el compilador tiene un cierto conocimiento interno sobre esta interfaz.

La clase `Queryable` implementa prácticamente los mismos operadores de consulta que `Enumerable`, con una diferencia bastante lógica: en lugar de un operador `AsEnumerable()`, `Queryable` ofrece el operador `AsQueryable()`. Además, están ausentes los operadores `Range()` y `Repeat()`, que no tienen una contrapartida natural en SQL.

El recurso de C# 3.0 sobre el que se apoya la funcionalidad de los métodos extensores de `IQueryable<T>` son los **árboles de expresiones.** En el caso de las expresiones de consulta de LINQ to SQL y otros proveedores basados en esta interfaz, las implementaciones de los operadores de consulta no reciben, conjuntamente con la secuencia de entrada, delegados a los que llamar según se estime oportuno, sino árboles de expresiones que reflejan lo que debe hacer el operador; esta vía permite que los implementadores una flexibilidad mucho mayor a la hora de devolver la secuencia de salida más conveniente.

¿Cómo funciona este mecanismo? Recordemos (Capítulo 8) que los árboles de expresiones permiten almacenar las expresiones lambda como estructuras de datos que pueden ser manipuladas según la necesidad, para que luego en el momento oportuno alguien las convierta en código IL listo para su ejecución. Recordemos también del Capítulo 9 que la transformación de una expresión de consulta en una secuencia de llamadas a métodos es un mecanismo basado en la **reescritura del código fuente,** y que las expresiones lambda (que es precisamente lo que acompaña a los operadores `where`, `select`, etc. en las expresiones de consulta) pueden ser transformadas según el contexto en delegados (código), que es lo que ocurre en el caso de LINQ to Objects, o en árboles de expresiones, que es lo que hace el compilador cuando la secuencia de origen implementa `IQueryable<T>`.

11.4.1. Definiciones

La interfaz `System.Linq.IQueryable<T>` está definida dentro de `System.Core.dll` de la siguiente forma:

```
public interface IQueryable<T> : IEnumerable<T>, IQueryable,
   IEnumerable
{
}
```

Ya hemos dicho que `IQueryable<T>` hereda de `IEnumerable<T>`, que es a su vez heredera de `IEnumerable`; adicionalmente, hereda también de una interfaz no genérica `IQueryable`, que tiene la siguiente apariencia:

```
public interface IQueryable : IEnumerable
{
   // Propiedades
   Type ElementType { get; }
   Expression Expression { get; }
   IQueryProvider Provider { get; }
}
```

De estas tres propiedades, la más compleja e importante es la última, que devuelve un **proveedor de consultas:** el objeto que en última instancia determina cómo se van transformando las secuencias para reflejar la consulta a ejecutar y cómo y cuándo ésta se conecta a la fuente de datos subyacente para producir los resultados. La interfaz `IQueryProvider` se define así:

```
public interface IQueryProvider
{
   // Métodos
   IQueryable CreateQuery(Expression e);
   IQueryable<TElement> CreateQuery<TElement>(Expression e);

   object Execute(Expression e);
   TResult Execute<TResult>(Expression e);
}
```

Está claro que el ejercicio de implementación de un proveedor basado en `IQueryable<T>` tiene sus complicaciones, pues tendremos que implementar las tres propiedades de `IQueryable`, una de las cuales nos exige definir una clase que implemente `IQueryProvider`; además del mecanismo de enumeración dictado por `IEnumerable<T>`, que es el que garantiza, a fin de cuentas, la posibilidad de recorrer los resultados.

11.4.2. Ejemplo básico

Veamos inicialmente un ejemplo muy sencillo de cómo implementar esta interfaz para hacer posibles las consultas integradas sobre un tipo de colección determinado. Para ello, supongamos que disponemos de la siguiente clase (una versión simplificada de la clase `Persona` de la Introducción.

DIRECTORIO DEL CODIGO: EJEMPLO11 _ 04

```
public class Persona
{
   public Persona(string nombre, int edad)
   {
      Nombre = nombre;
      Edad = edad;
   }
   public string Nombre { get; set; }
   public int Edad { get; set; }
}
```

Al estilo de como lo hemos hecho anteriormente, vamos a implementar nuestro tipo "consultable" como heredero de `List<T>`, con lo que evitaremos programar explícitamente el mecanismo de enumeración de elementos, del que las listas ya

disponen. Para más simplicidad, haremos que los operadores de consulta implementen directamente el filtrado, ordenación y selección en lugar de ir "acumulando" el trabajo para hacerlo al final.

Inicialmente, definiremos una clase que llamaremos `Proveedor` y será la que implementará `IQueryProvider`. Generaremos automáticamente código para sus métodos, con la ayuda de Visual Studio 2008:

```
class Proveedor : IQueryProvider
{
   #region IQueryProvider Members

   public IQueryable IQueryable.CreateQuery(Expression e)
   {
      throw new Exception("No implementado.");
   }

   public IQueryable<TElement> IQueryable<TElement>.
      CreateQuery<TElement>(Expression e)
   {
      throw new Exception("No implementado.");
   }

   public object IQueryable.Execute(Expression expression)
   {
      throw new Exception("No implementado.");
   }

   public TResult IQueryable<T>.Execute<TResult>(Expression e)
   {
      throw new Exception("No implementado.");
   }

   #endregion
}
```

También generaremos los cuerpos de los métodos de las interfaces `IQueryable` e `IQueryable<T>` sobre nuestro tipo de colección. En el caso de la propiedad `Provider`, haremos que devuelva un nuevo objeto de la clase `Proveedor`:

```
class ClaseLinqQueryable<T> : List<T>, IOrderedQueryable<T>
{
   #region IQueryable Members

   public Type IQueryable.ElementType
   {
      get { throw new Exception("No implementado."); }
```

Continúa

```
    }

    public Expression IQueryable.Expression
    {
        get { throw new Exception("No implementado."); }
    }

    public IQueryProvider Provider
    {
        get { return new Proveedor(); }
    }

    #endregion
}
```

Si ahora en el cuerpo de `Main()` hacemos lo siguiente:

```
// colección de prueba
var src = new ClaseLinqQueryable<Persona>();
src.Add(new Persona("Judith", 37));
src.Add(new Persona("Gustavo", 45));
src.Add(new Persona("Irina", 41));
src.Add(new Persona("Diana", 12));

// consulta sobre la colección anterior
var res = from p in src where p.Edad > 25 select p.Nombre;
```

el programa compilará correctamente, aunque por supuesto fallará en ejecución tan pronto se produzca la primera llamada a alguno de esos métodos realmente sin implementar. Recuerde de los fundamentos de LINQ que la consulta anterior se traducirá a:

```
var res = src.Where(p => p.Edad > 25).Select(p => p.Nombre);
```

Pues bien, debido a que `src` implementa `IQueryable<T>`, el compilador elegirá traducir las funciones lambda a árboles de expresiones en lugar de delegados, resultando algo así:

```
ParameterExpression par = Expression.Parameter(
    typeof(Persona), "p");

/* p => p.Edad > 25 */
Expression<Func<Persona, bool>> predicado =
    Expression.Lambda<Func<Persona,bool>>(
        /* > */
```

Continúa

```
      Expression.GreaterThan(
         /* p.Edad (int) */
         Expression.Property(par,
            typeof(Persona).GetProperty("Edad")),
         /* 25 */
         Expression.Constant(25)
      ),
      /* par */
      new ParameterExpression[] { par }
   );

/* p => p.Nombre */
Expression<Func<Persona, string>> selección =
   Expression.Lambda<Func<Persona, string>>(
      /* p.Nombre */
      Expression.Property(par,
         typeof(Persona).GetProperty("Nombre")),
      /* p */
      new ParameterExpression[] { par }
   );
```

Por supuesto, reescribirá la expresión de consulta en llamadas a métodos extensores:

```
var res = src.Where(predicado).
   Select(selección);
```

Y como `src` implementa `IQueryable<T>` y nosotros no hemos suministrado versiones propias de estos operadores, utilizará los métodos extensores definidos en la clase `System.Linq.Queryable`:

```
var tmp = Queryable.Where(src, predicado).
var res = Queryable.Select(tmp, selección);
```

11.4.3. Implementación de IQueryable<T>

¿Qué significan los métodos y propiedades de las interfaces `IQueryable` e `IQueryable<T>` y cómo debemos implementarlos para que funcionen de la manera adecuada? Veamos:

- La propiedad `ElementType` debe devolver el tipo de los elementos de la secuencia. La implementación más natural sería:

```
Type IQueryable.ElementType
{
   get { return typeof(T); }
}
```

- La propiedad `Expression` debe devolver el árbol de expresión asociado al objeto `IQueryable<T>`. En particular, desde dentro de los operadores de consulta implementados en `Queryable` se hace una llamada a esta propiedad para obtener los árboles de expresión correspondientes a los operandos de los operadores de consulta. Una implementación bastante típica podría ser:

```
Expression IQueryable.Expression
{
   get { return Expression.Constant(this); }
}
```

  En este caso el objeto, al ser consultado, responderá "éste soy yo, y se me puede considerar una expresión constante".
- La propiedad `Provider`, como hemos dicho antes, debe devolver un objeto que implemente `IQueryProvider`.

11.5. QUÉ HACE EL PROVEEDOR DE CONSULTAS

En este punto de la historia, hay que explicar qué hacen las implementaciones predeterminadas en `IQueryable<T>` de los operadores de consulta. Básicamente, todos estos operadores **generan un árbol de expresión** en el que ellos mismos se representan en forma de un nodo de tipo `MethodCall` (llamada a método) que tiene como argumentos a los argumentos originales de la llamada al operador. Por ejemplo, en el caso de nuestra expresión de consulta de prueba, el método `Where()` construirá el siguiente árbol de expresión, en el que el primer argumento de la llamada es la colección original, y el segundo, el predicado a comprobar:

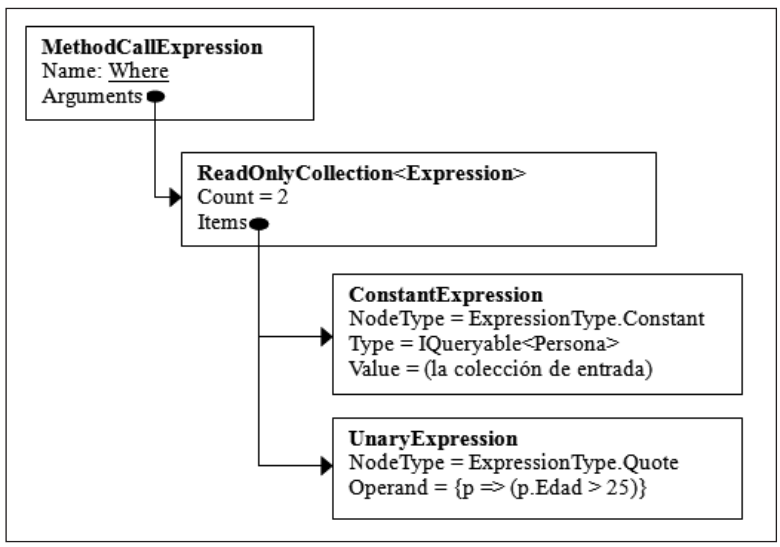


Figura 11.1.

A continuación, los operadores solicitan a su secuencia de entrada un proveedor, y llaman método `CreateQuery()` de éste, pasándole como parámetro el árbol de expresión recién construido. A partir de este árbol de expresión "de entrada", `CreateQuery()` debe producir una secuencia "de salida", en la que se registre de alguna manera el trabajo a realizar para evaluar (inmediatamente o en un momento posterior) el resultado que el operador de consulta está solicitando.

El mismo proceso se repetirá en el caso de nuestro segundo operador de consulta, sólo que en este caso se trata del método `Select()`, el primer argumento será árbol resultante de la llamada precedente a `Where()`, y el segundo argumento, la expresión lambda que permite seleccionar el nombre de la persona.

Si no se ha ido realizando "sobre la marcha", la evaluación del resultado de la consulta se debe implementar cuando el método `CreateQuery()` del proveedor reciba un árbol cuyo nodo raíz es una llamada a `Select()`, ya que éste es uno de los elementos "terminales" en la sintaxis de las expresiones de consulta (el otro es `GroupBy()`, como vimos en el Capítulo 9).

En el caso de los operadores de consulta que producen resultado escalares, como `Sum()` o `Count()`, que también son elementos terminales en las cadenas de llamadas a operadores, éstos llaman al método `Execute()` del proveedor para que se encargue del cálculo correspondiente.

11.5.1. El método CreateQuery()

Como hemos mencionado antes, el método `CreateQuery()` del proveedor de consultas es el "caballo de batalla" que será llamado para procesar cada uno de los operadores presentes en la expresión de consulta reescrita. A través de su único parámetro, `CreateQuery()` recibirá un árbol de expresión que refleja todos los detalles de la llamada que se está realizando.

A continuación, presentamos un ejemplo de implementación de `CreateQuery()` para nuestro tipo de colección personalizada. Es una implementación directa, que ejecuta cada operación (filtro, selección y ordenación) directamente sobre la colección de entrada y devuelve una colección de salida (como nodo de tipo `ConstantExpression`), en lugar de ir transformando paso a paso un árbol de expresión, tarea que dejaremos para la próxima sección. Este ejemplo le permitirá comprender qué parámetros envía `Queryable` a `CreateQuery()` para cada uno de los operadores presentes en la expresión de consulta.

```
IQueryable<TElement> IQueryProvider.
   CreateQuery<TElement>(Expression e)
{
   Console.WriteLine("CREATE QUERY: " + e);

   MethodCallExpression mc = (MethodCallExpression)e;
   switch (mc.Method.Name)
   {
```

Continúa

```
            case "Where":
               // filtrar los elementos de la colección de entrada
               // que satisfacen la condición
               ClaseLinqQueryable<TElement> objeto1 =
                  (ClaseLinqQueryable<TElement>)
((mc.Arguments[0] as ConstantExpression).Value);
               LambdaExpression expr1 =
                  (mc.Arguments[1] as UnaryExpression).Operand
                    as LambdaExpression;
               Func<TElement, bool> predicado =
                  (Func<TElement, bool>)expr1.Compile();
               ClaseLinqQueryable<TElement> resultadoW =
                 new ClaseLinqQueryable<TElement>();
               foreach (TElement t in objeto1)
               {
                  if (predicado(t))
                     resultadoW.Add(t);
               }
               return resultadoW;
            case "Select":
               // transformar los elementos de la colección de entrada
               ClaseLinqQueryable<Persona> objeto2 =
                  (ClaseLinqQueryable<Persona>)
((mc.Arguments[0] as ConstantExpression).Value);
               LambdaExpression expr2 =
                  (mc.Arguments[1] as UnaryExpression).Operand
                  as LambdaExpression;
               Func<Persona, TElement> trans =
                  (Func<Persona, TElement>)expr2.Compile();
               ClaseLinqQueryable<TElement> resultadoS =
                  new ClaseLinqQueryable<TElement>();
               foreach (Persona p in objeto2)
               {
                  resultadoS.Add(trans(p));
               }
               return resultadoS;
            case "OrderBy":
               // ordenar los elementos de la colección de entrada
               // según el criterio especificado
               ClaseLinqQueryable<TElement> objeto3 =
                  (ClaseLinqQueryable<TElement>)
                     ((mc.Arguments[0] as ConstantExpression).Value);
               LambdaExpression expr3 =
                  (mc.Arguments[1] as UnaryExpression).Operand
                    as LambdaExpression;
               Type tipo = expr3.Body.Type;

               Delegate criterio = expr3.Compile();
               // copiar la colección original
```

Continúa

```
            ClaseLinqQueryable<TElement> resultado0 =
               new ClaseLinqQueryable<TElement>();
            for (int i = 0; i < objeto3.Count; i++)
               resultado0.Add(objeto3[i]);
            // ordenar
            for (int i = 0; i < resultado0.Count - 1; i++)
               for (int j = i + 1; j < resultado0.Count; j++)
            {
               IComparable c1 =
            (IComparable)criterio.DynamicInvoke(resultado0[i]);
               IComparable c2 =
            (IComparable)criterio.DynamicInvoke(resultado0[j]);
               if (c1.CompareTo(c2) > 0)
            {
               TElement tmp = resultado0[i];
               resultado0[i] = resultado0[j];
               resultado0[j] = tmp;
            }
         }
      return resultado0;
   default:
      throw new NotImplementedException("NO IMPLEMENTADO");
  }
}
```

Una observación a tener en cuenta: nuestra colección implementa no sólo `IQueryable<T>`, sino una interfaz heredera de ésta que no añade ningún método adicional, `IOrderedQueryable<T>`. Es el único requisito a cumplir en el caso de que se desee que nuestra clase dé soporte a las operaciones de ordenación. Si se incluye una cláusula **`orderby`** en una expresión de consulta sin haber "marcado" la colección subyacente como `IOrderedQueryable<T>`, se producirá una excepción en tiempo de ejecución.

11.5.2. El método Execute()

Ya hemos mencionado antes también que el método `Execute()` del proveedor de consultas es llamado cuando es necesario evaluar un operador de consulta que produce un resultado escalar. A continuación, se muestra un ejemplo de implementación del operador `Count()`:

```
public T Execute<T>(Expression e)
{
   Console.WriteLine("EXECUTE: " + e);

   MethodCallExpression mc = (MethodCallExpression)e;
```

Continúa

```
   switch (mc.Method.Name)
   {
      case "Count":
         // contar cuántos (se ignora el posible argumento)
         IEnumerable obj =
            (IEnumerable)
            ((mc.Arguments[0] as ConstantExpression).Value);
         int n = 0;
         foreach (var x in obj)
            n++;
         return (T)Convert.ChangeType(n, typeof(T));
      default:
         throw new NotImplementedException("NO IMPLEMENTADO");
      }
   }
```

Este método podríamos utilizarlo luego así:

```
var xx = (from x in src
          where x.Nombre.Contains('D')
          select x).Count();
Console.WriteLine(xx);
```

11.6. UN EJEMPLO REAL: LINQ TO TFS

La implementación de `IQueryable<T>` que hemos suministrado antes no hace uso de las posibilidades de transformación paulatina de secuencias a partir de árboles de expresiones en las que se apoya, por ejemplo, LINQ to SQL. Como ejemplo de una implementación más "apropiada" en este sentido, presentaremos ahora el esqueleto de una extensión de LINQ que hemos llamado **LINQ to TFS** (el nombre, así como la idea original, se deben a mi colega y amigo Unai Zorrilla), y que permite ejecutar consultas integradas contra un repositorio de Team Foundation Server en el que se almacena, entre otros datos, la información relativa a los diferentes elementos que componen un proyecto de desarrollo de software y la historia de su evolución en el tiempo.

11.6.1. Presentación de las API de TFS

Visual Studio 2008 ofrece al desarrollador avanzado numerosas vías para la extensión y personalización del entorno de desarrollo. Para hacer más factibles esas posibilidades, el Kit de Desarrollo de Software de Visual Studio (Visual Studio SDK) ofrece diversas API de extensibilidad mediante las cuales se hace posible no solo añadir al entorno nuevos asistentes u opciones de menú, sino también acceder a toda la información que Visual Studio gestiona internamente. Un conjunto de

tales librerías lo conforman las API de interacción con Team Foundation Server, el servidor que sirve como *back end* a la edición Team System de Visual Studio y que es el encargado de gestionar toda la información relativa a los proyectos de desarrollo compartidos por un equipo. El estudio de estas API sobrepasa con creces los objetivos de este libro, y aquí nos limitaremos a describir los conceptos mínimos necesarios para comprender la implementación de la extensión de LINQ que nos ocupa.

Un concepto que es necesario introducir para comprender de qué trata esta extensión es el de **elemento de trabajo** (*work item*). En los proyectos de TFS, un elemento de trabajo es un registro que TFS utiliza para controlar la asignación a usuarios y el estado de cumplimiento de un cierto trabajo que forma parte de un proyecto. Entre los posibles tipos de elementos están las tareas, peticiones de cambios, revisiones, requisitos y errores. Cada elemento de trabajo tiene diferentes propiedades, como la fecha de creación (`CreationDate`), su creador (`CreatedBy`), y en particular una clave primaria (`Id`) que lo identifica únicamente. Aunque las herramientas visuales integradas en Visual Studio Team System (el llamado Team Foundation Client) permiten consultar esa información de una manera bastante cómoda y completa, un desarrollador de extensiones de Visual Studio podría querer en cierto momento recuperar un subconjunto específico de esos datos para mostrarlos de manera personalizada. Para facilitar esto, Microsoft suministra el llamado **Work Item Query Language** (WIQL), inspirado en SQL pero orientado específicamente a consultar los elementos de trabajo almacenados en un repositorio TFS.

La sintaxis de WIQL es, *grosso modo,* la siguiente:

```
SELECT <ListaExpresiones> FROM <TablaOVista>
WHERE <Condiciones> ORDER BY <ListaCampos>
```

Un ejemplo podría ser:

```
SELECT Id, Title, [Created By], [Created Date]
FROM WorkItems WHERE [Created By] CONTAINS 'Marco Amoedo'
ORDER BY [Created Date]
```

El sencillo proveedor que presentaremos aquí nos permitirá consultar el repositorio sin necesidad de escribir las sentencias WIQL: nuestro proveedor se encargará precisamente de generarlas. Para poder centrarnos en los conceptos, la funcionalidad que presentamos es mínima; en particular, implementaremos únicamente la ejecución de consultas de tipo "SELECT *", donde se recuperan elementos de trabajo completos (lo cual implica, como hemos visto en el Capítulo 10, que la llamada final al método `Select()` se optimizará). El lector interesado podrá obtener una implementación más amplia, con todo el código fuente, en Codeplex (**www.codeplex.com**), bajo un proyecto con el nombre "LINQ to TFS".

11.6.2. Ejemplo básico de consulta

Para ejecutar consultas WIQL, antes que nada es necesario agregar a los proyectos referencias a los siguientes ensamblados: `Microsoft.TeamFoundation.dll`, `Microsoft.TeamFoundation.Client.dll` y `Microsoft.TeamFoundation.WorkItemTracking.Client.dll`. Cada uno de estos ensamblados contiene clases pertenecientes a un espacio de nombres del mismo nombre, que conviene importar para mayor comodidad.

En el código, lo primero que hay que hacer es conectarse al servidor TFS y obtener una referencia a su `WorkItemStore` (almacén de elementos de trabajo), para luego emitir las consultas contra ese almacén. El siguiente ejemplo muestra la mecánica tradicional:

DIRECTORIO DEL CODIGO: EJEMPLO11 _ 05

```
   // *** VIA ESTANDAR

   // Nos conectamos al TFS
   TeamFoundationServer server = TeamFoundationServerFactory.
      GetServer("http://tfs.plainconcepts.com:8147/");
   // Obtenemos una referencia a su WorkItemStore
   WorkItemStore workItemStore = (WorkItemStore)server.
      GetService(typeof(WorkItemStore));
   // La consulta
   string query =
      @"SELECT Id, Title, [Created By], [Created Date] FROM WorkItems
WHERE [System.TeamProject] = 'Lebab'
   AND [Created By] = 'Marco Amoedo'
ORDER BY [Created Date]";
   // Ejecutamos la consulta
   WorkItemCollection collection = workItemStore.Query(query);
   // Recuperamos los resultados
   Console.WriteLine("* TFS API *");
   foreach (WorkItem i in collection)
      Console.WriteLine(i.Id + " | " + i.Title + " | " + i.CreatedDate
+ " | " + i.CreatedBy);
```

Como puede verse, la API de TFS ya suministra un tipo `WorkItem` para representar a los elementos de trabajo, y los nombres de columnas que se utilizan en la sentencia WIQL (por ejemplo, `Created Date`) no coinciden exactamente con los nombres de las propiedades de esta clase, sino que existe un mecanismo de mapeado de unas en otras.

11.6.3. Un proveedor básico

Nuestro plan de acción para habilitar las consultas integradas sobre la API de elementos de trabajo será la siguiente:

- En una clase estática, definiremos un método extensor para el almacén de elementos de trabajo (la clase `WorkItemStore`), al que llamaremos

`ToQueryable()`. Esta será nuestra "puerta de entrada" para la ejecución de consultas integradas.

- El método anterior devolverá un objeto de una clase propia que implementa `IOrderedQueryable<WorkItem>`; esto nos permitirá programar consultas como la del siguiente ejemplo (observe la duplicidad de cláusulas **`where`**, algo que en SQL es ilegal, pero no en la sintaxis de C# 3.0):

```
// *** VIA LINQ

var q = from i in workItemStore.ToQueryable()
        where i.Project.Name == "Lebab"
        where i.CreatedBy == "Marco Amoedo"
        orderby i.CreatedDate
        select i;
Console.WriteLine("* LINQ TO TFS *");
foreach (var i in q)
   Console.WriteLine(i.Id + " | " + i.Title + " | " +
      i.CreatedDate + " | " + i.CreatedBy);
```

- La clase `QueryableItemStore<WorkItem>`, que implementará la interfaz `IQueryable<WorkItem>`, se encargará (con la ayuda del correspondiente proveedor de consultas) de transformar las secuencias de entrada de las llamadas a `Where()` y `OrderBy()` en otros árboles de expresiones que reflejen esas operaciones de filtrado y ordenación; el árbol de expresiones solamente se transformará en sentencia WIQL y se enviará al servidor TFS cuando se inicie la enumeración de los resultados.

11.6.4. La puerta de entrada a LINQ

No es nada difícil añadir el método extensor necesario a la clase `WorkItemStore`:

```
namespace PlainConcepts.LinqToTfs
{
   public static class QueryableStore
   {
      public static QueryableItemStore<WorkItem>
         ToQueryable(this WorkItemStore wis)
      {
         return new QueryableItemStore<WorkItem>(wis);
      }
   }

   // ...
}
```

Naturalmente, pasaremos al constructor de la clase `QueryableItemStore` la URL correspondiente al servidor TFS con el que deberá interactuar. El constructor la guardará en un campo privado:

```
public class QueryableItemStore<WorkItem> : IOrderedQueryable<
WorkItem>
{
   private WorkItemStore _wis = null;

   public QueryableItemStore(WorkItemStore wis)
   {
      _wis = wis;
   }

   // ...
}
```

11.6.5. La implementación de IQueryable<WorkItem>

A continuación, se presenta la implementación de la interfaz `IOrderedQueryable<WorkItem>`. Esta clase se apoya ciegamente en otra, su proveedor de consultas, que es la que suministra el método `CreateQuery()` y a la que hemos llamado `QueryableItemProvider<WorkItem>`.

Como `IQueryable<T>` hereda de `IEnumerable<T>`, aquí es necesario implementar el mecanismo de enumeración que irá devolviendo los resultados de la consulta. En este método `GetEnumerator()` , que aplazaremos hasta más adelante, será donde construiremos la sentencia WIQL, la enviaremos al servidor TFS y produciremos los resultados.

```
public class QueryableItemStore<WorkItem> : IOrderedQueryable<
WorkItem>
{
   private WorkItemStore _wis = null;

   public QueryableItemStore(WorkItemStore wis)
   {
      _wis = wis;
   }

   #region State

   private List<Expression> whereConditions =
      new List<Expression>();
   internal List<Expression> WhereConditions
   {
      get { return whereConditions; }
   }
```

Continúa

```
    private List<OrderCriteria> orderFields =
       new List<OrderCriteria>();
    internal List<OrderCriteria> OrderFields
    {
       get { return orderFields; }
    }

    #endregion

    #region IQueryable Members

    public Type ElementType
    {
       get { return typeof(WorkItem); }
    }

    public Expression Expression
    {
       get { return Expression.Constant(this); }
    }

    public IQueryProvider Provider
    {
       get { return new QueryableItemProvider<WorkItem>(); }
    }

    #endregion

    #region IEnumerable<WorkItem> Members

    private IEnumerator<WorkItem> getEnumerator()
    {
       // ¡Aquí estará la acción!
    }

    public IEnumerator<WorkItem> GetEnumerator()
    {
       return getEnumerator();
    }

    #endregion

    #region IEnumerable Members

    System.Collections.IEnumerator
       System.Collections.IEnumerable.GetEnumerator()
    {
       return getEnumerator();
    }

    #endregion
}
```

Lo más interesante aquí es la presencia de las propiedades `WhereConditions` y `OrderFields`; no queremos que se ejecute ninguna acción en el momento, sino demorarlo todo hasta el final; por eso almacenamos las condiciones de filtro y los criterios de ordenación en sendas listas privadas, que irán pasando de un resultado intermedio a otro hasta que llegue el momento oportuno.

11.6.6. El proveedor de consultas

Llega ahora el momento de implementar el proveedor de consultas, y en particular su método `CreateQuery()`, el corazón de la clase. Habiendo visto antes los campos declarados en la clase `QueryableItemStore`, probablemente ya imagine cómo funciona el método en el caso de llamadas a `Where()` y `OrderBy()` o `ThenBy()`: almacenando en el campo correspondiente de la secuencia de entrada el criterio de filtro u ordenación y devolviendo la misma secuencia:

```
private IQueryable<WorkItem> createQuery<WorkItem>(Expression e)
{
   MethodCallExpression mc = e as MethodCallExpression;
   switch (mc.Method.Name)
      {
         case "Where":
            {
               QueryableItemStore<WorkItem> entrada =
                  (mc.Arguments[0] as ConstantExpression).Value
                    as QueryableItemStore<WorkItem>;

               LambdaExpression expr =
                  (mc.Arguments[1] as UnaryExpression).Operand
                    as LambdaExpression;

               entrada.WhereConditions.Add(expr);
               return entrada;
            }
         case "OrderBy":
         case "ThenBy":
         case "OrderByDescending":
         case "ThenByDescending":
            {
               QueryableItemStore<WorkItem> entrada =
                  (mc.Arguments[0] as ConstantExpression).Value
                    as QueryableItemStore<WorkItem>;

               LambdaExpression expr1 =
                  (mc.Arguments[1] as UnaryExpression).Operand
                    as LambdaExpression;
               string fName =
```

Continúa

```
                (expr1.Body as MemberExpression).Member.Name;

            entrada.OrderFields.Add(
               new OrderCriteria {
                  FieldName = fName,
                  Ascending =
                    ! mc.Method.Name.Contains("Descending")
               });
            return entrada;
      }
   case "Select":
      // No se utiliza aquí
      throw new NotImplementedException();
   default:
      throw new NotImplementedException();
   }
}
```

11.6.7. El mecanismo de enumeración

Lo realmente interesante viene cuando toca implementar la enumeración de los resultados de la consulta: aquí es cuando hay que generar la sentencia WIQL, enviarla a TFS y recoger los resultados. Lo más interesante está en la transformación de los árboles de expresiones correspondientes a los criterios de filtro, donde hay que echar mano a algoritmos de recorrido de árboles de expresiones como los que vimos en el Capítulo 8. Por otra parte, generar la cadena de caracteres apropiada a partir de nuestra lista de criterios de ordenación es bastante trivial. A continuación, se presenta el código del método enumerador:

```
private IEnumerator<WorkItem> getEnumerator()
{
   var entrada =
      (this.Expression as ConstantExpression).Value
        as QueryableItemStore<WorkItem>;

   // * generar consulta *
   StringBuilder sb = new StringBuilder("SELECT [Id] FROM
WorkItems");
   if (entrada.WhereConditions.Count > 0)
   {
      sb.Append("\n WHERE ");
      foreach (Expression w in entrada.WhereConditions)
         sb.Append("(" + Utils.TranslateCondition(w) + ") AND ");
      sb.Remove(sb.Length - 5, 5);
   }
```

```
   if (entrada.OrderFields.Count > 0)
   {
      sb.Append("\n ORDER BY ");
      foreach (OrderCriteria o in entrada.OrderFields)
         sb.Append(Utils.TranslateOrderCriteria(o) + ", ");
      sb.Remove(sb.Length - 2, 2);
   }

   string WIQLSentence = sb.ToString();
   WorkItemCollection col = _wis.Query(WIQLSentence);
   foreach (WorkItem wi in col)
      yield return wi;
}
```

Hemos encapsulado en la clase estática `Utils` los métodos auxiliares que se encargan de transformar los filtros y criterios de ordenación en las cadenas que conforman la sentencia WIQL:

```
internal static class Utils
{
   private static Dictionary<string, string> translations =
      new Dictionary<string, string> {
         { "CreatedBy",           "[Created By]" },
         { "CreatedDate",         "[Created Date]" },
         { "Id",                  "[Id]" },
         { "Title",               "[Title]" },
         { "Project.Name",        "[System.TeamProject]" }
         // otros más...
      };

   public static string TranslateCondition(Expression condition)
   {
      if (condition == null)
         return "";
      switch (condition.NodeType)
      {
         case ExpressionType.Lambda:
            {
               Expression lambda =
                  (condition as LambdaExpression).Body;
               return TranslateCondition(lambda);
            }

         // lógicos
```

Continúa

```
        case ExpressionType.And:
        case ExpressionType.AndAlso:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " AND " + TranslateCondition(
              (condition as BinaryExpression).Right);
        case ExpressionType.Or:
        case ExpressionType.OrElse:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " OR " + TranslateCondition(
              (condition as BinaryExpression).Right);
        // relacionales
        case ExpressionType.Equal:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " = " + TranslateCondition(
              (condition as BinaryExpression).Right);
        case ExpressionType.NotEqual:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " <> " + TranslateCondition(
              (condition as BinaryExpression).Right);
        case ExpressionType.GreaterThan:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " > " + TranslateCondition(
              (condition as BinaryExpression).Right);
        case ExpressionType.GreaterThanOrEqual:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " >= " + TranslateCondition(
              (condition as BinaryExpression).Right);
        case ExpressionType.LessThan:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " < " + TranslateCondition(
              (condition as BinaryExpression).Right);
        case ExpressionType.LessThanOrEqual:
           return TranslateCondition(
              (condition as BinaryExpression).Left) +
              " <= " + TranslateCondition(
              (condition as BinaryExpression).Right);
        case ExpressionType.Constant:
           {
              ConstantExpression c =
                 condition as ConstantExpression;
```

Continúa

```
            if (c.Type == typeof(string))
               return "'" + c.Value.ToString() + "'";
            else
               return c.Value.ToString();
         }
      case ExpressionType.MemberAccess:
         {
            MemberExpression m =
               condition as MemberExpression;
         string name = m.ToString();
         string pName = name.Substring(
               name.IndexOf('.') + 1);
         return translations[pName];
         }
      default:
         throw new NotImplementedException();
   }
}

public static string TranslateOrderCriteria(OrderCriteria
criteria)
{
   return translations[criteria.FieldName] +
      (criteria.Ascending ? "" : " DESC");
}
}
```

¡Y eso es todo! Con estas clases a nuestra disposición, podremos recuperar información de un repositorio TFS utilizando consultas integradas, de una manera elegante a la vez que muy eficiente.

Espero que esta sección le haya permitido hacerse una idea sobre todo lo relacionado con la implementación de proveedores basados en `IQueryable<T>`.

Extensiones de LINQ

CAPÍTULO 12

LINQ to XML

Con este capítulo damos inicio a la última parte del libro, que trata sobre las extensiones de LINQ que Microsoft ha incorporado "de serie" en .NET Framework 3.5. Cualquiera de ellas (a excepción, tal vez, de LINQ to DataSet) podría merecer seguramente un libro entero; tales son las posibilidades que ofrecen, dado que esas tecnologías trascienden a la mera acción de consultar una fuente de datos determinada para permitir también la actualización de los datos y en general la manipulación, en el sentido más amplio de la palabra, de los contenedores en los que estos datos se almacenan: documentos XML y bases de datos relacionales. Por esta razón, nos limitaremos a exponer las características fundamentales de cada modelo, acompañándolas con algunos ejemplos que muestren las posibilidades que se abren ante nosotros.

12.1. PRESENTACIÓN

Antes de la aparición de .NET Framework 3.5 y Visual Studio 2008, la librería de clases de .NET Framework ya incluía diferentes mecanismos para la manipulación programática de documentos XML:

- Las clases `XmlTextReader` y `XmlTextWriter` (espacio de nombres `System.Xml`) posibilitan la lectura y escritura secuencial de documentos XML. El tratamiento secuencial puede ser una solución muy eficiente en ciertos casos, pero adolece de serias deficiencias cuando se requieren tratamientos más complejos, como pueden ser los relacionados con la selección o actualización de nodos en un documento.
- En los casos relacionados con la seriación o de-seriación de objetos hacia o desde documentos XML, se pueden utilizar las clases del espacio de nombres `System.Xml.Serialization`.
- Por último, las clases `XmlDocument`, `XmlNode` y otras (también en `System.Xml`) componen la implementación de un Modelo de Objetos de Documento (Document Object Model, DOM) que hace posible representar los documentos XML mediante árboles en la memoria operativa del sistema. Esta "visión simultánea" de todo el contenido del documento (siempre que las limitaciones de tamaño no lo impidan), combinada con las posibilidades que ofrece la API

construida alrededor de estas clases, garantiza una potencia considerable a la hora de realizar tratamientos complejos sobre documentos XML.

Pues bien, con la aparición de .NET 3.5 tenemos a nuestra disposición una API más para trabajar con documentos XML: la que forma parte de la implementación de LINQ to XML. Las clases que componen esta API se almacenan en el ensamblado `System.Xml.Linq.dll`; el nombre del espacio de nombres también es `System.Xml.Linq`.

Esta nueva librería de clases sigue las líneas generales de la tercera de las API mencionadas anteriormente; es decir, se trata de una implementación de un **modelo de objetos** alternativo para representar en memoria documentos XML, que ha sido desarrollada con los siguientes objetivos:

- Simplificar sustancialmente el modelo actual, a la vez que eliminar sus inconsistencias e inconveniencias.
- Aprovechar en la implementación de estas librerías las novedades que han sido incorporadas a C# y VB en fechas relativamente recientes, como los genéricos (Capítulo 2) o los tipos anulables (Capítulo 4).
- Ofrecer el soporte necesario para la implementación consistente de las consultas integradas en el lenguaje (LINQ) sobre documentos XML.

La siguiente figura muestra de manera resumida la jerarquía de clases las clases que lo componen:

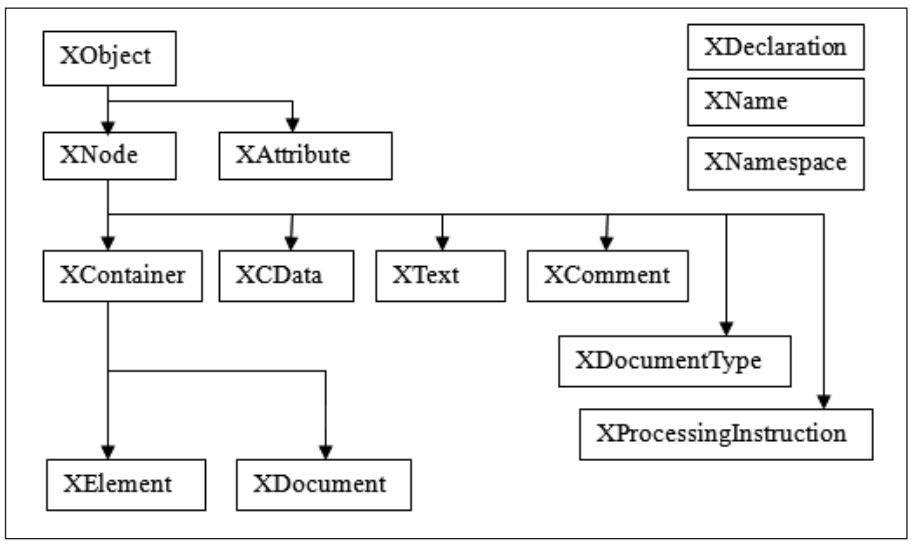


Figura 12.1.

`XObject` es la raíz de toda la jerarquía, y de ella heredan `XNode` (base para las clases que representan nodos del árbol de representación de un documento) y `XAttribute` (clase que representa a los atributos XML). De `XNode` desciende a su

vez `XContainer`, que sirve como ancestro común a `XDocument` y a `XElement`, de cuyas instancias se componen generalmente los árboles. Los demás tipos de nodos sólo pueden ser hojas situadas bajo un `XElement`. Tenga en cuenta también que aunque `XText` y `XCData` se exponen como parte de la jerarquía, deben considerarse en general como detalles de implementación interna: la API permite recuperar el texto situado dentro de un elemento como una cadena u otro valor de tipo simple.

En el código fuente que sigue a continuación presentamos un ejemplo de programación de una misma funcionalidad utilizando las API "tradicionales" de .NET y la que ofrece LINQ To XML. Se trata de la creación de un mismo documento XML a partir de una colección de objetos de la clase `Persona` que hemos venido utilizando a lo largo del libro (se define en la Introducción).

DIRECTORIO DEL CODIGO: EJEMPLO12_01

```
// los datos
static List<Persona> Generación = new List<Persona> {
   new Persona { Nombre = "Diana", Sexo = SexoPersona.Mujer,
                 FechaNac = new DateTime(1996, 2, 4) },
   new Persona { Nombre = "Dennis", Sexo = SexoPersona.Varón,
                 FechaNac = new DateTime(1983, 12, 27) },
   new Persona { Nombre = "Jennifer", Sexo = SexoPersona.Mujer,
                 FechaNac = new DateTime(1982, 8, 12) },
   new Persona { Nombre = "Claudia", Sexo = SexoPersona.Mujer,
                 FechaNac = new DateTime(1989, 7, 26) },
   new Persona { Nombre = "Amanda", Sexo = SexoPersona.Mujer,
                 FechaNac = new DateTime(1998, 10, 23) },
   new Persona { Nombre = "Adrián", Sexo = SexoPersona.Varón,
                 FechaNac = new DateTime(2005, 9, 29) }
};

private void button1_Click(object sender, EventArgs e)
{
   // *** API secuencial
   using (XmlTextWriter tw = new XmlTextWriter("Familia1.xml",
                                               Encoding.UTF8))
   {
      tw.Formatting = Formatting.Indented;
      tw.WriteStartDocument();
      tw.WriteStartElement("Familia");
      foreach (Persona p in Generación)
      {
         tw.WriteStartElement("Persona");
         tw.WriteElementString("Nombre", p.Nombre);
         if (p.Sexo.HasValue)
            tw.WriteElementString("Sexo",
```

Continúa

```
                p.Sexo.Value.ToString());
            if (p.FechaNac.HasValue)
                tw.WriteElementString("FechaNac",
                p.FechaNac.Value.ToString("yyyy-MM-dd"));
            tw.WriteEndElement();
        }
        tw.WriteEndElement();
    }
}

private void button2_Click(object sender, EventArgs e)
{
    // *** API DOM
    XmlDocument doc = new XmlDocument();
    doc.AppendChild(doc.CreateXmlDeclaration("1.0",
                            "utf-8", null));
    XmlElement raíz = doc.CreateElement("Familia");
    doc.AppendChild(raíz);
    foreach (Persona p in Generación)
    {
        XmlElement nodo = doc.CreateElement("Persona");
        XmlElement x = doc.CreateElement("Nombre");
        XmlText v = doc.CreateTextNode(p.Nombre);
        x.AppendChild(v);
        nodo.AppendChild(x);
        if (p.Sexo.HasValue)
        {
            x = doc.CreateElement("Sexo");
            v = doc.CreateTextNode(p.Sexo.Value.ToString());
            x.AppendChild(v);
            nodo.AppendChild(x);
        }
        if (p.FechaNac.HasValue)
        {
            x = doc.CreateElement("FechaNac");
            v = doc.CreateTextNode(p.FechaNac.
                Value.ToString("yyyy-MM-dd"));
            x.AppendChild(v);
            nodo.AppendChild(x);
        }
        raíz.AppendChild(nodo);
    }
    doc.Save("Familia2.xml");
}

private void button3_Click(object sender, EventArgs e)
{
    // *** API de LINQ To XML
```

Continúa

```
   XDocument doc = new XDocument(
                   new XDeclaration("1.0",
                          "utf-8", null));
   XElement raíz = new XElement("Familia");
   doc.Add(raíz);
   foreach (Persona p in Generación)
   {
      // construcción funcional
      XElement nodo =
         new XElement("Persona",
            new XElement("Nombre", p.Nombre));
      if (p.Sexo.HasValue)
      {
         nodo.Add(new XElement("Sexo", p.Sexo.Value));
         // *** equivale a:
         // nodo.Add(new XElement("Sexo",
         //       new XText(p.Sexo.Value.ToString())));
      }
      if (p.FechaNac.HasValue)
      {
         nodo.Add(new XElement("FechaNac",
            p.FechaNac.Value.ToString("yyyy-MM-dd")));
      }
      raíz.Add(nodo);
   }
   doc.Save("Familia3.xml");
}
```

El documento XML resultante de cualquier de estas tres transformaciones, que utilizaremos como datos de entrada a partir de la próxima sección, es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<Familia>
   <Persona>
      <Nombre>Diana</Nombre>
      <Sexo>Mujer</Sexo>
      <FechaNac>1996-02-04</FechaNac>
   </Persona>
   <Persona>
      <Nombre>Dennis</Nombre>
      <Sexo>Varón</Sexo>
      <FechaNac>1983-12-27</FechaNac>
   </Persona>
   <Persona>
      <Nombre>Jennifer</Nombre>
      <Sexo>Mujer</Sexo>
```

Continúa

```
      <FechaNac>1982-08-12</FechaNac>
   </Persona>
   <Persona>
      <Nombre>Claudia</Nombre>
      <Sexo>Mujer</Sexo>
      <FechaNac>1989-07-26</FechaNac>
   </Persona>
   <Persona>
      <Nombre>Amanda</Nombre>
      <Sexo>Mujer</Sexo>
      <FechaNac>1998-10-23</FechaNac>
   </Persona>
   <Persona>
      <Nombre>Adrián</Nombre>
      <Sexo>Varón</Sexo>
      <FechaNac>2005-09-29</FechaNac>
   </Persona>
</Familia>
```

En el fragmento de código en el que se hace uso de la nueva API que ofrece LINQ to XML se reflejan algunas de las principales características que la diferencian de sus predecesoras, como son:

- **Construcción funcional:** A diferencia de la manera típica "de abajo hacia arriba" en que se construyen los árboles XML al trabajar con el DOM tradicional, LINQ to XML promueve un estilo funcional de construcción que permite crear un árbol completo o una parte de él utilizando una única sentencia. Por ejemplo, observe la sentencia del código en la que se crea un fragmento de documento para asignarlo a la variable `nodo`.
- **Independencia del documento:** A diferencia de la implementación actual del DOM, en la que todos los nodos de un árbol XML deben ser creados en el contexto de un objeto `XmlDocument` (clase en la que se concentran los métodos `CreateXXX()` para crear los diferentes tipos de nodos), LINQ to XML permite crear los elementos y otros fragmentos directamente, sin depender de un documento (aunque sí existe una clase `XDocument`, como hemos visto antes, que puede utilizarse si se cree necesario o conveniente).
- El **texto como valor:** Generalmente, los elementos terminales de un árbol XML contienen valores tales como cadenas, números enteros o decimales, fechas, etc. LINQ to XML permite tratar los elementos y atributos que contienen valores de una manera natural, y no exige la creación explícita de nodos de texto (aunque en este sentido ya la clase `XmlDocument` ofrecía una vía "no estándar" para salvarse esa exigencia, mediante la propiedad `InnerText`). Pero aún hay más: la librería ofrece **operadores de conversión explícitos** desde todos los tipos predefinidos y otros de uso frecuente, como `DateTime` y `TimeSpan`, de modo que podamos escribir nuestro código de la manera más sencilla y cómoda posible. Por ejemplo, fíjese en la sentencia del ejemplo anterior en la que se construye el elemento asociado al sexo de la persona.

Más adelante presentaremos algunos ejemplos más de cómo llevar a cabo las tareas comunes de manipulación de documentos XML utilizando las facilidades que ofrece esta API. Pero como este libro trata sobre consultas integradas, nos centraremos primero en cómo utilizar LINQ to XML para ejecutar consultas contra (la representación en memoria de) documentos XML.

12.2. EXPRESIONES DE CONSULTA SOBRE DOCUMENTOS XML

En el capítulo anterior hemos analizado las diferentes vías mediante las cuales es posible "habilitar para LINQ" a clases o jerarquías enteras. Pues bien, en el caso de LINQ to XML se utiliza la misma implementación básica de los operadores de consulta estándar, lo cual no es de extrañar si se tiene en cuenta que en el fondo estamos hablando de colecciones de objetos en la memoria operativa del sistema. La mejor prueba de esto es que el código de los ejemplos a continuación deja de compilar si se comenta la sentencia de importación `using System.Linq;` al principio del código fuente.

Lo único que ha sido necesario hacer para posibilitar la ejecución de consultas integradas sobre esta familia de objetos es definir diversos "puntos de entrada" a partir de los cuales se puede "arrancar" una consulta. Frecuentemente, este punto de entrada para la ejecución de expresiones de consulta sobre documentos XML es el método `Elements()` definido en la clase `XContainer` (y por ello disponible para `XDocument` y `XElement`), que produce como resultado un `IEnumerable<XElement>` que genera una secuencia de los elementos descendientes del contenedor especificado. En la siguiente tabla se enumeran otros métodos de `XElement` que también producen `IEnumerable<T>` y, por lo tanto, pueden servir como origen para la ejecución de consultas.

Tabla 12.1. Métodos de XElement que devuelven IEnumerable<XElement> o IEnumerable<XNode>.

Heredados de XNode	
`Ancestors()`	Produce una secuencia `IEnumerable<XElement>` con todos los elementos situados por encima del nodo actual en el árbol del documento XML. Una segunda sobrecarga de este método permite filtrar por el nombre del elemento.
`ElementsBeforeSelf()`	Produce una secuencia con todos los elementos "hermanos" (situados a un mismo nivel del árbol) del elemento actual y que aparecen antes que él en el documento. Una segunda sobrecarga de este método permite filtrar por el nombre del elemento.

Continúa

Heredados de XNode	
`NodesBeforeSelf()`	Similar al anterior, pero produce no sólo los elementos, sino también los posibles nodos de otros tipos (por ejemplo `XComment`; consulte la jerarquía de clases) que estén presentes a ese mismo nivel en el documento.
`ElementsAfterSelf()`	Produce una secuencia con todos los elementos "hermanos" (situados a un mismo nivel del árbol) del elemento actual y que aparecen después de él en el documento. Una segunda sobrecarga de este método permite filtrar por el nombre del elemento.
`NodesAfterSelf()`	Similar al anterior, pero produce no sólo los elementos, sino también los posibles nodos de otros tipos que estén presentes a ese mismo nivel en el documento.
Heredados de XContainer	
`Elements()`	Produce una secuencia `IEnumerable<XElement>` con los elementos descendientes del elemento especificado. Una segunda versión de este método permite indicar el nombre de los elementos en que estamos interesados.
`Nodes()`	Similar al anterior, pero produce no una secuencia de `XElement` sino de `XNode`; además de los elementos, se incluyen también en la secuencia los nodos de texto, `XComment`, etc.
`Descendants()`/ `DescendantNodes()`	Produce una secuencia "aplanada" con todos los elementos (nodos) situados por debajo del elemento (nodo) actual en el subárbol XML. Una segunda versión de `Descendants()` permite indicar el nombre de los elementos en que estamos interesados.
Propios	
`DescendantsAndSelf()`/ `DescendantNodesAnd-Self()`	Similares a `Descendants()` y `DescendantNodes()`, pero incluyen también al propio elemento (nodo) actual.
`AncestorsAndSelf()`	Similar a `Ancestors()`, pero incluye también al propio elemento actual.
`Attributes()`	Produce una secuencia `IEnumerable<XAttribute>` con los atributos XML asociados al elemento original. Una segunda sobrecarga permite indicar el nombre de los atributos.

A continuación, se muestra la ejecución de una expresión de consulta sobre el documento de ejemplo. Observe que la expresión se inicia sobre una secuencia `IEnumerable<XElement>`, pero produce una secuencia `IEnumerable<Persona>` que podría ser origen para una consulta subsiguiente. Esta uniformidad de LINQ a través de los diferentes orígenes de datos es crucial a la hora de, por ejemplo, implementar la interacción entre bases de datos y documentos XML.

```
private void button4_Click(object sender, EventArgs e)
{
   // carga del documento XML
   XElement doc = XElement.Load("Familia3.xml");
   // expresión de consulta
   var chicas = from x in doc.Elements()
      where (string) x.Element("Sexo") == "Mujer"
      orderby (string) x.Element("Nombre")
      select new Persona() { Nombre = (string) x.Element("Nombre"),
         Sexo = SexoPersona.Mujer };
   // ejecución
   foreach (var chica in chicas)
      MessageBox.Show(chica.Nombre);
}
```

Para recuperar el elemento o atributo de una instancia de `XElement` con el nombre especificado, esta clase ofrece dos métodos, `Element()` y `Attribute()`, respectivamente. Formalmente, el parámetro es del tipo `XName`; pero éste ofrece una conversión implícita desde **`string`**, y eso será generalmente lo que utilicemos. Los tipos de retorno de esos métodos, `XElement` y `XAttribute`, son quienes disponen de los operadores de conversión explícita a los que se hacía alusión en la sección anterior.

A continuación, se presenta otro ejemplo de consulta:

```
private void button10_Click(object sender, EventArgs e)
{
   // carga del documento XML
   XElement doc = XElement.Parse(
      @"<discografia>
         <album>
            <nombre>Boston</nombre>
            <fecha>1975</fecha>
         </album>
         <album>
```

Continúa

```
            <nombre>Don't look back</nombre>
            <fecha>1977</fecha>
         </album>
         </discografia>");
      lbxFechas.Items.Clear();
      var años =
         from x in doc.Elements("album")
         select (int) x.Element("fecha") ;
      foreach (var a in años)
         lbxFechas.Items.Add(a);
}
```

Observe la utilización del método estático `XElement.Parse()` para convertir la representación textual de un documento XML en un árbol de nodos en memoria.

12.3. OPERADORES DE CONSULTA ESPECÍFICOS DE LINQ TO XML

LINQ to XML define adicionalmente un conjunto de operadores propios, específicos para las secuencias de elementos XML, que han sido modelados siguiendo las líneas generales de los **ejes de XPath.** Estos operadores, que se enumeran la siguiente tabla, se han implementado como métodos extensores de `IEnumerable<XElement>` o `IEnumerable<T>` en la clase `System.Xml.Linq.Extensions`. En la práctica, esto implica que para ejecutar consultas integradas contra documentos XML hay que importar tanto `System.Linq` como `System.Xml.Linq`.

Tabla 12.2.

Operadores específicos de LINQ to XML	
`Elements()`	Produce los elementos hijos de cada `XElement` presente en una secuencia `IEnumerable<XElement>`. Una segunda versión de este método permite indicar el nombre de los elementos en que estamos interesados.
`Nodes()`	Similar al anterior, pero produce no una secuencia de `XElement` sino de `XNode`; además de los elementos, se incluyen también nodos de texto, `XComment`, etc.
`InDocumentOrder()`	Produce la misma secuencia de entrada, pero con los nodos ordenados según el orden en que aparecen en el documento.
`Descendants()`/ `DescendantNodes()`	Produce una secuencia "aplanada" con todos los elementos (nodos) situados por debajo del elemento (nodo) actual en el subárbol XML. Una segunda versión de `Descendants()` permite indicar el nombre de los elementos en que estamos interesados.

Continúa

Operadores específicos de LINQ to XML	
`DescendantsAndSelf()/ DescendantNodesAnd-Self()`	Similares a los anteriores, pero incluyen también al elemento (nodo) actual.
`Ancestors()`	Produce una secuencia "aplanada" con todos los elementos situados por encima del elemento actual en el subárbol XML. Una sobrecarga de este método permite indicar el nombre de los descendientes.
`AncestorsAndSelf()`	Similar al anterior, pero incluye también al elemento actual.
`Attributes()`	A partir de una secuencia `IEnumerable<XElement>`, produce una secuencia `IEnumerable<XAttribute>` con todos los atributos XML presentes en la secuencia de elementos original. Una sobrecarga de este método permite indicar el nombre de los atributos.

El ejemplo que se muestra a continuación extrae las fechas de nacimiento de las personas de los elementos XML correspondientes, utilizando el operador `Descendants()`. Generalmente, estos operadores específicos de LINQ to XML se aplican, como en este caso, sobre la secuencia resultante de ejecutar alguno de los métodos propios o heredados de la clase `XElement`.

```
private void button5_Click(object sender, EventArgs e)
{
   // carga del documento XML
   XElement doc = XElement.Load("Familia3.xml");
   // extracción de fechas de cumpleaños
   lbxFechas.Items.Clear();
   foreach (DateTime fechaRegalo in
        doc.Elements().Descendants("FechaNac"))
      lbxFechas.Items.Add(fechaRegalo.ToString("dd 'de' MMMM"));
}
```

12.4. BÚSQUEDAS EN DOCUMENTOS XML

Se supone que el mecanismo más cómodo para localizar un conjunto de nodos o elementos de un documento que cumplen ciertas condiciones son precisamente las consultas integradas. Observe (aunque parezca evidente) que el resultado de las consultas de LINQ to XML en las que selecciona el propio elemento de iteración, como por ejemplo en:

```
var js = from x in doc.Elements()
   where (int)x.Element("Edad") == 24
   select x;
```

es una secuencia de **referencias** a elementos o nodos del documento, que por ejemplo podremos utilizar para actualizar esos elementos o nodos (vea la próxima sección). Recuerde también que se puede hacer uso de todos los operadores de consulta estándar; por ejemplo, para localizar el elemento correspondiente a alguien llamado "Jennifer" (sabiendo que es único) podríamos utilizar la sentencia:

```
XElement j = doc.Elements("Persona").Single
             (x => (string)x.Element("Nombre") == "Jennifer");
```

12.4.1. Búsquedas XPath

Además de las consultas integradas, los árboles de LINQ to XML también pueden ser navegados mediante búsquedas XPath. Para ello, se ha añadido al espacio de nombres `System.Xml.XPath` una clase estática `Extensions` que ofrece un grupo de métodos extensores de `XNode`, entre los cuales se encuentran `XPathSelectElement()` y `XPathSelectElements()`, que funcionan de manera similar a los métodos `SelectSingleNode()` y `SelectNodes()` del DOM tradicional. Por ejemplo, el siguiente código produce una secuencia con los elementos que contienen los nombres de las personas:

```
private void button9_Click(object sender, EventArgs e)
{
   // carga del documento XML
   XElement doc = XElement.Load("Familia3.xml");

   // obtener nodos con nombres
   IEnumerable<XElement> nombres =
      doc.XPathSelectElements("//Persona//Nombre");
   foreach (var n in nombres)
      MessageBox.Show(n.Value);
}
```

12.5. INSERCIÓN, MODIFICACIÓN Y BORRADO DE NODOS

Para permitir la modificación de los documentos XML, las clases `XNode`, `XContainer` y `XElement` definen diversos métodos para añadir, reemplazar o borrar nodos de un documento. Mostraremos aquí brevemente ejemplos de las tres operaciones:

```
private void button8_Click(object sender, EventArgs e)
{
   // carga del documento XML
```

Continúa

```
   XElement doc = XElement.Load("Familia3.xml");

   // buscar un elemento concreto
   XElement j = doc.Elements("Persona").First(
      x => (string)x.Element("Nombre") == "Jennifer");

   if (j != null)
   {
      // inserción (en este caso detrás)
      j.AddAfterSelf(
         new XElement("Persona",
            new XElement("Nombre", "JOHN DOE"),
            new XElement("Sexo", SexoPersona.Varón),
            new XElement("FechaNac", new DateTime(1980, 1, 1))
         )
      );

      // modificación de elemento
      j.SetElementValue("Nombre", "Jenny");

      // borrado
      j.NextNode.Remove();
   }

   doc.Save("Familia5.xml");
}
```

Un detalle al que hay que prestar especial atención al trabajar con esta nueva API es cuando se realizan actualizaciones combinadas con consultas sobre documentos XML en memoria. Como las consultas se ejecutan en régimen diferido o bajo demanda (capítulos 3 y 9), sus resultados se obtendrán en la medida en que vayan siendo solicitados. Si entre medias se realizan actualizaciones sobre el árbol del documento, ¡éstas podrían potencialmente afectar a los resultados de las consultas en curso!

12.6. TRANSFORMACIÓN DE DOCUMENTOS

La transformación de documentos es un caso de uso muy frecuente cuando se trabaja con XML, y no queríamos obviar aquí un ejemplo de cómo la técnica de construcción funcional de documentos, combinada con las expresiones de consulta, potencia y simplifica en gran medida la aplicación de transformaciones.

Suponga que deseamos transformar nuestro documento de prueba en otro cuyo contenido es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<NombresEdades>
   <Hijo>
      <Nombre>Diana</Nombre>
      <Edad>11</Edad>
   </Hijo>
   <Hijo>
      <Nombre>Dennis</Nombre>
      <Edad>23</Edad>
   </Hijo>
   <Hijo>
      <Nombre>Jennifer</Nombre>
      <Edad>24</Edad>
   </Hijo>
   <Hijo>
      <Nombre>Claudia</Nombre>
      <Edad>17</Edad>
   </Hijo>
   <Hijo>
      <Nombre>Amanda</Nombre>
      <Edad>8</Edad>
   </Hijo>
   <Hijo>
      <Nombre>Adrián</Nombre>
      <Edad>1</Edad>
   </Hijo>
</NombresEdades>
```

El código necesario se presenta a continuación. La construcción funcional de documentos nos permite establecer desde el principio la forma general del resultado deseado, para luego rellenar los "huecos" del XML resultante mediante combinaciones de expresiones de consulta y aplicaciones de métodos.

```
private void button6_Click(object sender, EventArgs e)
{
     // carga del documento XML
     XElement doc = XElement.Load("Familia3.xml");
     // transformación del documento original
     XElement nuevo =
        new XElement("NombresEdades",
           from x in doc.Elements("Persona")
           select new XElement("Hijo",
              x.Element("Nombre"),
              new XElement("Edad",
           Persona._Edad((DateTime)x.Element("FechaNac"))
        )
     )
   );
   nuevo.Save("Familia4.xml");
}
```

Para que esto funcione, hemos incluido un método estático de conveniencia `_ Edad()` a la clase `Persona`.

12.6.1. Transformaciones XSLT

Al igual que ha ocurrido en el caso de XPath, se ha añadido una clase estática `Extensions` con métodos extensores de `XElement` al espacio de nombres `System.Xml.Xsl`, con el objetivo de hacer posible la aplicación de transformaciones XSLT a documentos de LINQ to XML. Estos métodos permiten utilizar las clases `XslTransform` y `XslCompiledTransform`, ya familiares a los programadores .NET, para ejecutar transformaciones XSLT sobre secuencias `IEnumerable<XElement>`.

12.7. CONCLUSIÓN

En este capítulo hemos presentado de una manera resumida las principales características que ofrece a los desarrolladores LINQ to XML. Dado el hecho de que, como hemos mostrado aquí, LINQ to XML incorpora una nueva API para el tratamiento de documentos XML, y que esa API es la base sobre la que se implementan las expresiones de consulta integradas en C# 3.0 y VB 9.0, además de varias extensiones directas de este último lenguaje, creemos altamente conveniente que los programadores .NET se familiaricen cuanto antes con esta tecnología.

CAPÍTULO 13

LINQ to DataSet

Otra de las áreas del desarrollo de aplicaciones .NET para las que se ha encontrado utilidad a LINQ es la relativa a los conjuntos de datos en memoria. **LINQ to DataSet** es otra de las extensiones de LINQ que incluye .NET Framework 3.5, y ha sido concebida para hacer posible utilizar la sintaxis de las expresiones de consulta tanto contra conjuntos de datos "básicos" o "no tipados" (instancias de la clase `DataSet`) como contra conjuntos de datos fuertemente tipados (instancias de clases que heredan de `DataSet` y están "personalizadas" para representar los datos provenientes de una base de datos específica).

Es evidente que una facilidad como ésta es de gran utilidad, por ejemplo, en todas las capas de las aplicaciones de múltiples capas, en los que la clase `DataSet` (como también la clase `DataTable` a partir de la versión 2.0 de .NET) frecuentemente juega un papel esencial como elemento de transmisión. En principio, donde parece más útil su aplicación es en las capas intermedia y de interfaz de usuario, puesto que para la capa de acceso a datos se utilizaría LINQ to SQL, API a la que dedicaremos el siguiente capítulo, o LINQ to Entities, de próxima aparición. Por supuesto, también es tremendamente importante desde el punto de vista conceptual el hecho de poder utilizar un mecanismo uniforme (en este caso la sintaxis de las expresiones de consulta) en todas las capas de una aplicación y contra diferentes fuentes de datos, cualquiera que sea su naturaleza.

Los ejemplos de este capítulo se apoyan en la base de datos sobre fútbol español que hemos expuesto en la Introducción. Volcaremos el contenido de esa base de datos a conjuntos de datos en memoria para manipular éstos mediante expresiones de consulta. Dado que este capítulo se centra específicamente en la ejecución de consultas sobre conjuntos de datos, ignoraremos en gran medida el mecanismo mediante el cual los datos provenientes de esa base de datos llegan a uno u otro conjunto de datos del que dispongamos en la aplicación.

13.1. PRESENTACIÓN

Podemos definir LINQ to DataSet como "la extensión de LINQ destinada a hacer posibles las consultas integradas en los lenguajes .NET sobre objetos de las clases `DataSet`, `DataTable` o derivadas de éstas". Se trata de otra tecnología "montada" sobre la base de los operadores de consulta estándar en su estado original; lo cual no es de extrañar, dado que en el fondo se trata de objetos en memoria. Aunque se trata, indudablemente, de una extensión de menor envergadura que sus hermanas LINQ

to XML, LINQ to SQL o LINQ to Entities, no por ello deja de tener la importancia conceptual y utilidad práctica que ya hemos comentado anteriormente.

Hay que destacar aquí la presencia de la clase `DataTable` como ciudadana de primera clase en esta ecuación; como seguramente conoce el lector, `DataTable` no era serializable en la versión 1 de .NET Framework, por lo que todo el protagonismo recaía necesariamente sobre `DataSet`; en la versión 2.0 esa limitación desapareció, y `DataTable` puede utilizarse ahora de manera independiente en contextos en los que antes debía ser encapsulada obligatoriamente dentro de un `DataSet`. Aunque a la tecnología en general se le denomina LINQ to DataSet, el lector se dará cuenta rápidamente que su verdadero protagonista (quienes sirven como "puerta de entrada" a LINQ y generadores de secuencias) son las tablas.

Este capítulo se divide en dos partes bien diferenciadas, dedicadas respectivamente a mostrar la ejecución de consultas LINQ y los mecanismos en que éstas se apoyan (en principio diferentes) para conjuntos de datos no tipados, por una parte, y tipados, por la otra. Como también seguramente conoce el lector, en la literatura sobre .NET generalmente se distingue entre esos dos tipos de conjuntos de datos: los conjuntos de datos no tipados son instancias de la propia clase `DataSet`, y el acceso a su contenido (por ejemplo, a cualquiera de las tablas que la componen, o a cualquiera de las filas de éstas) debe hacerse mediante las propiedades vectoriales de propósito general que esta clase ofrece (`Tables[]`, `Relations[]`, etc.); por el contrario, los conjuntos de datos tipados son instancias de clases herederas de `DataSet`, creadas automáticamente mediante la herramienta de línea de comandos **XSD.EXE** (posiblemente llamada desde dentro de Visual Studio), y que aumentan la interfaz de esta clase con propiedades, métodos, eventos y atributos específicos para la estructura de los datos con los que se va trabajar.

DIRECTORIO DEL CODIGO: EJEMPLO12 _ 01

La interfaz de usuario de la aplicación de ejemplo en tiempo de diseño es la siguiente:

Figura 13.2.

El conjunto de datos tipado y los adaptadores para cada una de las tablas han sido creados arrastrando desde la ventana de Orígenes de datos, mientras que el no conjunto de datos no tipado ha sido arrastrado directamente desde el Cuadro de herramientas. La propiedad `DataSource` de la rejilla apunta al objeto `BindingSource`. Durante la carga del formulario, primero se rellenan las tablas del conjunto tipado; y luego se hace apuntar el conjunto de datos no tipado `dataSet1` al conjunto de datos tipado (en definitiva, a los mismos datos); aquí, el hecho importante es que el tipo estático de la variable `dataSet1` es `DataSet`, y no podremos tratarlo igual que a `FUTBOL2006DataSet`, que incorpora múltiples propiedades, métodos y eventos adicionales:

```
private void Form1_Load(object sender, EventArgs e)
{
   clubTableAdapter.Fill(FUTBOL2006DataSet.Club);
   paisTableAdapter.Fill(FUTBOL2006DataSet.Pais);
   futbolistaTableAdapter.Fill(FUTBOL2006DataSet.Futbolista);
   //
   dataSet1 = FUTBOL2006DataSet;
}
```

Para mostrar las diferencias en el modelo de programación contra conjuntos de datos tipados y no tipados, implementaremos una misma consulta contra los dos conjuntos de datos disponibles: una consulta que combina datos de las tres tablas disponibles y produce una secuencia con los futbolistas extranjeros que tomaron parte en la Liga española 2006-2007, ordenados por país de nacimiento, club en el que jugaron y nombre. La idea es mostrar que las posibilidades que tenemos a nuestra disposición son exactamente las mismas que cuando consultamos cualquier otra fuente de datos habilitada para LINQ.

13.2. CONSULTAS CONTRA DATASET TIPADOS

La ejecución de consultas integradas contra conjuntos de datos tipados es totalmente natural, como comprenderá al ver el código a continuación. Observe, de paso, que los controles de datos pueden enlazarse a las expresiones de consulta a través de un `BindingSource`:

```
private void button2_Click(object sender, EventArgs e)
{
   var consulta =
      from futbolista in FUTBOL2006DataSet.Futbolista
      join club in FUTBOL2006DataSet.Club
         on futbolista.CodigoClub equals club.Codigo
```

Continúa

```
      join pais in FUTBOL2006DataSet.Pais
         on futbolista.CodigoPaisNacimiento equals pais.Codigo
      where pais.Codigo != "ES"
      orderby pais.Nombre, club.Nombre, futbolista.Nombre
      select new
         {
            Pais = pais.Nombre,
            Club = club.Nombre,
            futbolista.Nombre
         };
   bindingSource1.DataSource = consulta;
   dataGridView1.DataSource = bindingSource1;
}
```

La razón por las que las consultas integradas funcionan de un modo tan elegante sobre conjuntos de datos tipados hay que buscarla en el código generado por la utilidad XSD.EXE: las tablas tipadas que se generan (en nuestro ejemplo se llaman `ClubDataTable`, `PaisDataTable` y `FutbolistaDataTable`) heredan de una clase genérica llamada `TypedTableBase<T>`, que hereda de `DataTable` e implementa `IEnumerable<T>`, donde `T` es el tipo de fila correspondiente (`ClubDataRow`, `PaisDataRow` y `FutbolistaDataRow`). De esta manera, cualquiera de estas tablas tipadas puede verse como una secuencia de sus filas.

Hay que señalar que la clase `System.Data.TypedTableBase<T>`, así como las demás clases a las que haremos referencia cuando veamos cómo trabajar con conjuntos no tipados, están implementadas dentro de un nuevo ensamblado independiente llamado `System.Data.DataSetExtensions.dll`.

Con relación al enlace a datos, hay que hacer el mismo señalamiento que ya hicimos en el capítulo anterior con relación al comportamiento perezoso de las consultas: la modificación "por detrás" de un `DataSet` enlazado a datos puede afectar el resultado de la visualización. Para evitar efectos colaterales no deseados, "cachee" el resultado de la consulta:

```
bindingSource1.DataSource = consulta.ToList();
```

Se almacene o no previamente en caché el resultado de la consulta, una limitación importante de este enfoque del enlace a datos tiene que ver con la imposibilidad de actualizar el conjunto de datos subyacente a la consulta. Más adelante veremos cómo el método `AsDataView()` puede ayudarnos en ese sentido.

13.3. CONSULTAS CONTRA DATASET NO TIPADOS

Para hacer posible la ejecución de consultas contra conjuntos de datos no tipados, la próxima versión de .NET Framework incluirá nuevas clases estáticas que implementarán unos cuantos métodos extensores para las clases `DataTable` y `DataRow`. Estos

métodos forman parte de dos clases estáticas llamadas `DataTableExtensions` y `DataRowExtensions`, respectivamente, incluidas también en `System.Data.DataSetExtensions.dll`. El espacio de nombres al que han sido asociadas es `System.Data`, así que el simple hecho de importar este espacio (que es el mismo en el que residen de toda la vida `DataSet`, `DataTable`, etc.) hará que esos métodos extensores estén en ámbito.

El principal de los métodos que extienden la clase `DataTable` es `AsEnumerable()`, que constituye la "puerta de entrada" de esta clase al mundo LINQ. Este método, como su nombre indica, devuelve un objeto que implementa `IEnumerable<DataRow>`, exponiendo la tabla como una secuencia de sus filas. ¿Sería esto suficiente? En principio sí, pero tendría un inconveniente: a la hora de acceder a las columnas de cada fila, el hecho de que la propiedad `Item[]` de `DataRow` es de tipo **`object`** obligaría al programador a estar haciendo constantes conversiones explícitas. Por esta razón, se ha extendido también la clase `DataRow`, añadiéndole la familia de métodos extensores genéricos `Field<T>()`; a través del parámetro de tipo `T` el programador especifica el tipo del campo, mientras que a través de un parámetro "normal" indica el nombre o la posición del campo deseado.

El siguiente código ejemplifica perfectamente todo lo anterior.

```
private void button1_Click(object sender, EventArgs e)
{
   var consulta =
      from f in dataSet1.Tables["Futbolista"].AsEnumerable()
      join club in dataSet1.Tables["Club"].AsEnumerable()
         on f.Field<string>("CodigoClub") equals
            club.Field<string>("Codigo")
      join pais in dataSet1.Tables["Pais"].AsEnumerable()
         on f.Field<string>("CodigoPaisNacimiento") equals
            pais.Field<string>("Codigo")
      where pais.Field<string>("Codigo") != "ES"
      orderby pais.Field<string>("Nombre"),
         club.Field<string>("Nombre"), f.Field<string>("Nombre")
      select new
         {
            Pais = pais.Field<string>("Nombre"),
            Club = club.Field<string>("Nombre"),
            Jugador = f.Field<string>("Nombre")
         };
   bindingSource1.DataSource = consulta.ToList();
   dataGridView1.DataSource = bindingSource1;
}
```

Observe que en este esquema de utilización aún subsisten vestigios de "desajustes de impedancia": debemos indicar explícitamente los tipos de los campos, y escribir los nombres de las columnas, que no son comprobados por el compilador. Por supuesto, podríamos aplicar este enfoque también a conjuntos de datos tipados; pero

estaríamos desaprovechando toda la información de tipos disponible, y el resultado sería menos natural y menos eficiente de lo posible. Los conjuntos de datos tipados mejoran sustancialmente la experiencia de utilización de LINQ to DataSet.

13.4. DE VUELTA A UN DATATABLE

Debido a que con cierta frecuencia se deseará obtener el resultado de una consulta LINQ en forma de objeto `DataTable` (por ejemplo, para pasarlo hacia otra capa de la aplicación que espere un objeto de ese tipo), la clase que ofrece métodos extensores para `DataTable` en LINQ to DataSet incluye un método, `CopyToDataTable()`, que cumple con esa función:

```
private void button3_Click(object sender, EventArgs e)
{
   var consulta =
      from club in FUTBOL2006DataSet.Club
      orderby club.Ciudad
      select club;
   // convertir en tabla
   DataTable t = consulta.CopyToDataTable();
   //
   bindingSource1.DataSource = t;
   dataGridView1.DataSource = bindingSource1;
}
```

El método `CopyToDataTable()` ofrece dos sobrecargas: la que acabamos de mostrar, que crea un nuevo objeto `DataTable`, y una segunda, que permite incorporar el resultado de la consulta en una tabla ya existente, utilizando diferentes políticas de actualización.

13.5. ACTUALIZACIONES

La idea de los creadores de LINQ to DataSet (al igual que en los casos de LINQ to XML y LINQ to SQL) también fue siempre la de convertir a esta tecnología en una API de propósito general que permita no solamente recuperar información, sino también actualizarla. Por esta razón, entre los métodos extensores de la clase `DataRow` hay varias variantes de un método genérico `SetField<T>()` cuyo objetivo es permitir al programador actualizar el valor de un campo de la fila. Por ejemplo:

```
// actualización
var betis = consulta.First(c => c.Nombre.Contains("BETIS"));
betis.SetField<string>("Nombre", "BETIS manque pierda");
```

No es necesario llamar a `BeginEdit()` y `EndEdit()` o `CancelEdit()` para iniciar o finalizar el modo de edición; sí, es responsabilidad del programador llamar a `AcceptChanges()` para confirmar definitivamente los cambios realizados y que éstos no puedan ser revertidos mediante una llamada a `RejectChanges()`.

Aquí cabría repetir nuevamente la advertencia acerca de tener en cuenta el comportamiento diferido de los operadores de consulta cuando se realicen actualizaciones en combinación con la navegación de consultas.

13.6. EL MÉTODO ASDATAVIEW()

Para hacer posible un enlace a datos bidireccional con el conjunto de datos subyacente, garantizando que los cambios que se efectúen sobre los controles visuales se reflejarán en el `DataTable` correspondiente, LINQ to DataSet ofrece un operador de consulta especializado, `AsDataView()`, que hace posible representar como un objeto `DataView` de ADO.NET el conjunto de datos resultante de la ejecución de una consulta integrada. Por supuesto, esto nos permitirá hacer uso de todo lo que nos ofrece un `DataView`, principalmente ordenación y filtrado de las filas. El siguiente código muestra un ejemplo de un enlace a datos perfectamente funcional:

```
private DataView vista = null;

private void button4_Click(object sender, EventArgs e)
{
   var consulta =
      from club in FUTBOL2006DataSet.Club
         orderby club.Ciudad
         select club;

      vista = consulta.AsDataView();
      vista.RowFilter = "Ciudad LIKE 'M%'";
      vista.Sort = "Nombre";
      dataGridView1.DataSource = vista;
   }

   private void GrabarCambios()
   {
      clubTableAdapter.Update(FUTBOL2006DataSet.Club);
   }

private void button5_Click(object sender, EventArgs e)
{
   GrabarCambios();
}
```

Al ser creado, un `DataLinqView` "se engancha" a la tabla a la que la consulta está asociada y opera sobre ella. Eso sí, lo que sigue a la cláusula **`select`** en la expresión de consulta debe ser un registro completo de una tabla; la presencia de un tipo anónimo provocará una excepción.

13.7. CONCLUSIÓN

En este capítulo hemos mostrado las posibilidades que ofrece LINQ to DataSet, otra de las extensiones de LINQ incorporadas a .NET Framework 3.5, en este caso para la ejecución de consultas integradas contra conjuntos de datos en memoria.

CAPÍTULO 14

LINQ to SQL

A lo largo de los dos capítulos anteriores hemos presentado los fundamentos de dos tecnologías "aplicadas" de LINQ: **LINQ to XML,** para trabajar con documentos XML, y **LINQ to DataSet,** que permite operar sobre conjuntos de datos tanto tipados como no tipados. En este capítulo final nos centraremos en **LINQ to SQL,** una extensión para el acceso desde el propio lenguaje de programación a la información almacenada en bases de datos relacionales, que más que una librería destinada únicamente a permitir la ejecución de consultas contra bases de datos constituye un potente nuevo marco de trabajo para interactuar con éstas en un estilo orientado a objetos. Debo señalar que lo que presentaremos aquí es, necesariamente, poco más que una introducción; con todo el material relacionado con LINQ to SQL podría escribirse, sin lugar a dudas, un libro completo.

14.1. PRESENTACIÓN

Mucho se viene hablando desde hace tiempo del fenómeno conocido como "desajuste de impedancia" (*impedance mismatch*) que provocan las diferencias entre los modelos de programación que proponen los lenguajes de propósito general y los lenguajes de consulta sobre bases de datos relacionales, el tipo de almacén persistente más común que utilizan las aplicaciones de hoy día. Si bien el desarrollador piensa en términos de clases y objetos, propiedades, métodos y eventos (la POO basada en componentes es, desde hace ya bastante, el paradigma de programación predominante), a la hora de almacenar o recuperar esa información en una base de datos relacional debe traducir esos conceptos a tablas, filas y columnas, y utilizar un lenguaje "foráneo", SQL, para expresar las órdenes que desea enviar al motor de bases de datos. Para complicar la situación, estas sentencias (casi siempre dinámicas o parametrizadas, y dependientes de valores de propiedades de objetos o de variables de la aplicación) se generan dentro del código como cadenas de caracteres, para las que el entorno de desarrollo no puede ofrecer validación en tiempo de compilación y que, por tanto, pueden pasar a la fase de ejecución con errores que será necesario depurar y que conspiran contra la productividad del programador.

Para saltar la barrera que separa el mundo de la Programación Orientada a Objetos del mundo de la programación de bases de datos relacionales se han explorado varios caminos. Las bases de datos orientadas a objetos (**POET, Versant,** por mencionar algunas) realmente nunca llegaron a despegar para el gran público; tampoco lo hicieron API "puras" como los *bindings* del **ODMG.** Un enfoque que sí ha alcanzado una mayor cuota de éxito es el de los **mapeadores objeto-relacional** (Object-Relational Mappers, ORM); en esta categoría podemos encuadrar herramientas como **nHibernate**, **LLBLGen** o **TierDeveloper** (todas ellas disponibles para .NET Framework). Se trata de marcos de trabajo (generalmente una API acompañada de herramientas más o menos visuales de generación de código) que hacen posible en tiempo de ejecución **mapear**, o sea, establecer una correspondencia, entre objetos disponibles desde el lenguaje de programación OO y las filas de las diferentes tablas y vistas que componen la base de datos contra la que la aplicación trabajará.

Este último enfoque es al que se adhiere LINQ to SQL ya que, como decíamos antes, ofrece una API completa para la manipulación de datos relacionales, y no solamente las clases estáticas con los métodos extensores necesarios para implementar las expresiones de consulta. Como parte de LINQ to SQL se incluye un **ORM ligero** (*lightweight* ORM) que, si bien no incorpora todas las facilidades que pueden encontrarse en sistemas ORM comerciales como los antes mencionados, es en sí mismo más que suficiente para acometer las principales tareas comunes de recuperación y actualización de los datos relacionales. La siguiente imagen muestra el mapeado que se hace en LINQ to SQL entre los conceptos de la programación orientada a objetos y los conceptos de las bases de datos relacionales.

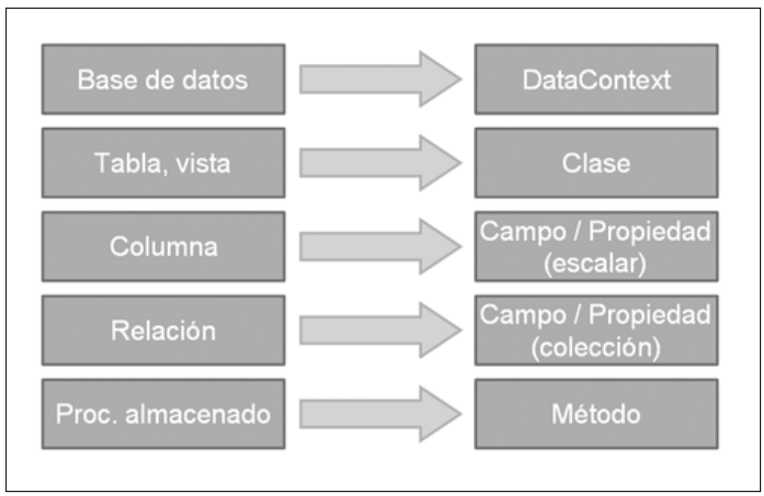


Figura 14.1.

Algo que es muy importante destacar es que todo este mecanismo se monta por encima de ADO.NET, como muestra la siguiente imagen:

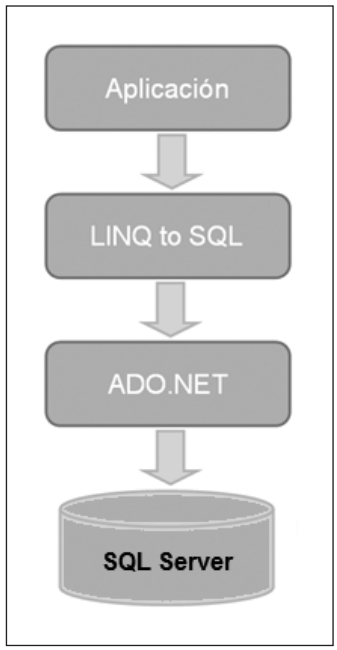


Figura 14.2.

Esto hace posible, por una parte, la "coexistencia pacífica" del nuevo código con el código ADO.NET creado para versiones anteriores; por otra, que podamos desde código LINQ to SQL obtener las referencias necesarias a los objetos ADO.NET correspondientes para actuar "a la vieja usanza" (por ejemplo, emitiendo directamente una sentencia SQL contra el almacén de datos) cuando cualquier razón (por ejemplo, el rendimiento) así lo aconseje; por último, facilitará extender LINQ to SQL a otras bases de datos para las que se disponga de proveedor ADO.NET (actualmente, LINQ to SQL sólo funciona para SQL Server 2000 y 2005).

Las clases y otros tipos que componen la API de LINQ to SQL se almacenan en el ensamblado `System.Data.Linq.dll`; el espacio de nombres principal es `System.Data.Linq`, dentro del cual se anidan otros varios espacios "subsidiarios".

14.2. CONTEXTOS DE DATOS Y CLASES DE ENTIDAD

El primer paso a dar para representar una base de datos mediante objetos es reflejar su estructura (metadatos) en un modelo de clases. El núcleo de este modelo lo componen las **clases de entidad** (*entity classes*): clases que permiten encapsular en un objeto los valores almacenados en las columnas de una fila de una cierta tabla de la base de datos.

La creación de este modelo puede realizarse de tres maneras diferentes: manualmente (con diferencia, la vía más difícil), escribiendo las clases nosotros mismos y adornándolas con los atributos necesarios; mediante una utilidad de línea de comandos

que se suministra con Visual Studio 2008 llamada **SQLMetal** (`SqlMetal.exe`), capaz de generar automáticamente todas las clases a partir de la base de datos; o mediante un diseñador visual integrado, que nos brinda la oportunidad de personalizar el modelo de manera visual, y que en el fondo llama tras las bambalinas en los momentos oportunos a la utilidad antes mencionada, que es su herramienta personalizada (*custom tool*). Para nuestro primer ejemplo utilizaremos el diseñador visual, que se despliega a través de la opción del proyecto **Agregar nuevo elemento | Clases de LINQ to SQL.** Esta opción presenta un lienzo de diseño en blanco sobre el que podremos arrastrar las tablas de nuestra base de datos desde el Explorador de servidores. Adicionalmente, se pueden agregar al modelo objetos de otros tipos; en el caso de nuestra base de datos de fútbol, además de las tres tablas `Club`, `Pais` y `Futbolista` arrastraremos también el procedimiento almacenado `InsertarFutbolista` (cuyo objetivo es insertar un nuevo futbolista en la base de datos) y la función definida por el usuario `FutbolistasDeUnPais` (que devuelve la cantidad de futbolistas del país indicado que juegan en la Liga española). La apariencia final del diseñador será la siguiente:

DIRECTORIO DEL CODIGO: EJEMPLO14 _ 01

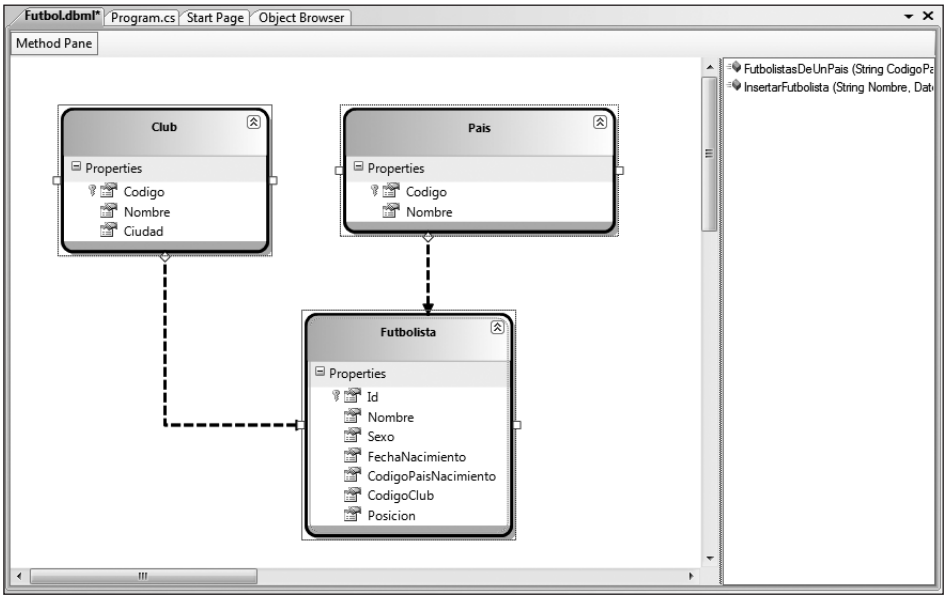


Figura 14.3.

Este diseño se almacena como parte del proyecto en un fichero con extensión **.dbml** (*database model*). Por supuesto, el objetivo final de este diseñador visual es el de generar código, de manera similar a como sucede, por ejemplo, con el diseñador de conjuntos de datos. Dedicaremos el próximo epígrafe a analizar el código fuente que se obtiene para nuestra base de datos.

14.2.1. El código generado por el diseñador

Si analiza el código generado automáticamente, lo primero que verá será una clase llamada `FutbolDataContext`, que hereda de la clase `DataContext` de `System.Data.Linq.dll`. Se trata de un **contexto de datos tipado,** una clase que especializa a la clase predeterminada `DataContext` para nuestra base de datos de ejemplo, del mismo modo que los conjuntos de datos tipados de ADO.NET son especializaciones de la clase `DataSet`. El contexto de datos es el canal a través del cual se interactúa desde LINQ to SQL con la base de datos. De hecho, incluye una propiedad `Connection`, de tipo `IDbConnection`, asociada a la conexión ADO.NET subyacente y que puede servir de "puente" entre ambos mundos. Adicionalmente, el contexto de datos se encarga de otras tareas esenciales, como el seguimiento de la identidad y estado actual de los objetos y la aplicación de las actualizaciones sobre el almacén relacional, como veremos más adelante. Del análisis del código fuente de la clase se desprenden otros dos hechos adicionales:

- El contexto de datos ofrece varios constructores que de un modo u otro obtienen la información relativa a la conexión ADO.NET que subyacerá al contexto, ya sea leyéndola del fichero de configuración de la aplicación o recibiéndola directamente como parámetro.
- El contexto de datos ofrece varias "puertas de entrada" para la ejecución de las consultas integradas en el lenguaje, a través de sus propiedades `Club`, `Futbolista` y `Pais`, que devuelven objetos del tipo `Table<T>`, donde `T` es el tipo de la clase de entidad correspondiente. Esta clase genérica implementa `IQueryable<T>`, de acuerdo con lo que necesita el mecanismo de consultas integradas para su funcionamiento.

> **Nota:**
>
> Si utiliza una versión en inglés de Visual Studio 2008, desactive la opción "Pluralization of names" en **Herramientas** | **Opciones** | **Herramientas de bases de datos** | **Diseñador O/R.** Esta opción convierte los nombres a plural al generar estas propiedades. La "pluralización" no va más allá de agregar una 's' al final del nombre original, cosa que en castellano no siempre funciona (ni en inglés tampoco :-).

- A continuación, el contexto de datos contiene métodos que podremos utilizar para llamar a los procedimientos almacenados y funciones definidas por el usuario que se hayan incluido en el modelo de la base de datos. La asociación de estos métodos y sus parámetros con las entidades de Transact SQL correspondientes se realiza a través de atributos. Más adelante presentaremos estos atributos con más detalle.
- Por último, el código generado incluye las definiciones de las clases de entidad, que por defecto reciben los mismos nombres de las tablas originales (estos nombres se pueden personalizar en el modelo mediante el Visor de propiedades). A continuación, presentamos un fragmento de la implementación de la clase `Pais`, sobre el que haremos luego varias observaciones.

```
[Table(Name="dbo.Pais")]
public partial class Pais : INotifyPropertyChanging,
                            INotifyPropertyChanged {

   private string _Codigo;
   private string _Nombre;

   #region Extensibility Method Definitions
   partial void OnLoaded();
   partial void OnValidate();
   partial void OnCreated();
   partial void OnCodigoChanging(string value);
   partial void OnCodigoChanged();
   partial void OnNombreChanging(string value);
   partial void OnNombreChanged();
   #endregion

   private EntitySet<Futbolista> _Futbolistas;

   public Pais() {
      OnCreated();
         this._Futbolistas = new EntitySet<Futbolista>(
            new Action<Futbolista>(this.attach_Futbolistas),
            new Action<Futbolista>(this.detach_Futbolistas));
   }

   [Column(Storage="_Codigo", Name="Codigo",
      DBType="Char(2) NOT NULL",
      IsPrimaryKey=true, CanBeNull=false)]
   public string Codigo {
      get {
         return this._Codigo;
      }
      set  {
           if ((this._Codigo != value)) {
               this.OnPropertyChanging("Codigo");
               this.SendPropertyChanging();
               this._Codigo = value;
               this.SendPropertyChanged("Codigo");
               this.OnPropertyChanged("Codigo");
         }
      }
   }

   [Column(Storage="_Nombre", Name="Nombre",
      DBType="VarChar(50) NOT NULL", CanBeNull=false)]
   public string Nombre {
      get {
         return this._Nombre;
```

Continúa

```
        }
        set  {
             if ((this._Nombre != value)) {
                 this.OnPropertyChanging("Nombre");
                 this.SendPropertyChanging();
                 this._Nombre = value;
                 this.SendPropertyChanged("Nombre");
                 this.OnPropertyChanged("Nombre");
             }
        }
    }

    [Association(Name="Pais_Futbolista",
        Storage="_Futbolistas",
        OtherKey="CodigoPaisNacimiento")]
    public EntitySet<Futbolista> Futbolistas {
        get  {
             return this._Futbolistas;
        }
        set  {
             this._Futbolistas.Assign(value);
        }
    }

    public event PropertyChangedEventHandler PropertyChanging;
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void SendPropertyChanging() {
        if ((this.PropertyChanging != null)) {
             this.PropertyChanging(emptyChangingEventArgs);
        }
    }

    protected virtual void SendPropertyChanged(string propName) {
        if ((this.PropertyChanged != null)) {
            this.PropertyChanged(this, new
             PropertyChangedEventArgs(propName));
        }
    }

    private void attach_Futbolista(Futbolista entity) {
        this.SendPropertyChanging();
        entity.Pais = this;
        this.SendPropertyChanged("Futbolistas");
    }

    private void detach_Futbolistas(Futbolista entity) {
        this.SendPropertyChanging();
        entity.Pais = null;
        this.SendPropertyChanged("Futbolistas");
    }
}
```

Las observaciones a hacer sobre el código generado para las clases de entidad son las siguientes:

- Como era de esperar, las clases incluyen una propiedad del tipo CLR adecuado para cada una de las columnas de la tabla de bases de datos correspondiente.
- Los métodos de acceso de escritura de cada una de estas propiedades incluyen "ganchos" a los que podemos "enchufarnos" para realizar las tareas que sean necesarias antes y/o después de un cambio en el valor de la propiedad. Estos "ganchos" se implementan mediante métodos parciales, la nueva característica de C# 3.0 que hemos presentado en el Capítulo 5.
- El mapeado de la clase y de cada una de las propiedades generadas a las entidades correspondientes de la base de datos se realiza mediante atributos. Por ejemplo, el atributo `System.Data.Linq.Table` asociado a la clase refleja el hecho de que la clase es una clase de entidad, y su parámetro `Name` indica el nombre de la tabla de la base de datos cuyas filas se representarán mediante objetos de esta clase. De la misma manera, el atributo `Column` permite asociar una propiedad de la clase de entidad con una columna de la tabla en la base de datos. Diversos parámetros del atributo reflejan cuál es el campo interno correspondiente a la propiedad en la clase, el nombre de la columna asociada de la base de datos, su tipo SQL o si es clave primaria, entre otras características.
- La navegación a través de las relaciones entre tablas existentes en la base de datos se implementa de manera natural mediante **propiedades de navegación.** Si tenemos un objeto `p` de la clase `Pais` y deseamos listar los futbolistas nacidos en ese país, basta con recorrer la propiedad `Futbolistas` del objeto (tratándose de una relación `1:N`, aquí sí no nos viene mal un plural, y por eso hemos personalizado el nombre de esta relación mediante el diseñador visual). Estas propiedades "de relación" son gestionadas por el motor de mapeado objeto-relacional de LINQ to SQL. Más adelante hablaremos algo más sobre el funcionamiento de este mecanismo.
- Las propiedades que permiten la navegación de las relaciones se "marcan" mediante el atributo `Association`, cuyos parámetros permiten especificar los detalles de la relación, tanto a nivel de contexto de datos como en la base de datos.

14.3. EJECUCIÓN DE CONSULTAS INTEGRADAS

Una vez que disponemos de las clases necesarias, podemos comenzar a utilizarlas para ejecutar consultas integradas contra la base de datos. Como ya hemos mencionado, el contexto de base de datos incluye propiedades de tipo `Table<T>` que pueden servir como generadores para la ejecución de las consultas integradas. Gracias a una sofisticada implementación de la interfaz `IQueryable<T>`, en línea con lo explicado en el Capítulo 11, las consultas integradas contra estas

propiedades se traducen en **sentencias SELECT de SQL** que serán enviadas al motor de bases de datos para su ejecución; como es tradicional en LINQ, la recuperación de los resultados de la consulta se produce según demanda, de manera que la especificación de una consulta es 100% "gratuita" hasta el momento en que se itera sobre ella.

A continuación, se presenta un ejemplo de ejecución de una expresión de consulta relativamente sencilla contra nuestra base de datos de ejemplo, así como su equivalente en términos de llamadas a métodos del patrón LINQ. Aquí también se muestra la utilización de un sencillo mecanismo de *logging* que ofrece el contexto de datos: a través de su propiedad `Log` podemos indicar un flujo de salida al que se enviará la sentencia SQL generada y otra información de control, de modo que podamos consultarla posteriormente.

```
static void Consulta1()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out; // logging en la consola

      var lista1 =
         from p in ctx.Futbolista
         join q in ctx.Club on p.CodigoClub equals q.Codigo
         where p.FechaNacimiento >= new DateTime(1987, 1, 1)
         orderby q.Nombre, p.Nombre
         select new
         {
            Jugador = p.Nombre + " (" + p.Posicion + ")",
            Equipo = q.Nombre
         };

      // equivalente con métodos extensores
      var lista2 =
         ctx.Futbolista.
            Join(ctx.Club,
               p => p.CodigoClub,
               q => q.Codigo,
               (p, q) => new { p, q }).
         Where(x => x.p.FechaNacimiento >=
                  new DateTime(1987, 1, 1)).
         OrderBy(x => x.q.Nombre).
         ThenBy(x => x.p.Nombre).
         Select(x => new
         {
            Jugador = x.p.Nombre + " (" + x.p.Posicion + ")",
```

Continúa

```
            Equipo = x.q.Nombre
        });

      Console.WriteLine("Consulta 1");
      foreach (var p in lista1)
        Console.WriteLine(p);

      Console.WriteLine("Consulta 2");
      foreach (var p in lista2)
        Console.WriteLine(p);
   }
}
```

La consulta enumera de manera ordenada los futbolistas que han cumplido 20 años en 2007 y los clubes a los que pertenecen. En la huella de registro que se obtiene en la consola podemos ver la sentencia SQL generada, además de los resultados (sólo mostramos una de las dos partes de la salida, que son por supuesto idénticas):

```
Consulta 1
SELECT (([t0].[Nombre] + @p1) + (CONVERT(NVarChar(1),[t0].[Posicion])))
+ @p2 AS [value], [t1].[Nombre]
FROM [dbo].[Futbolista] AS [t0]
INNER JOIN [dbo].[Club] AS [t1] ON [t0].[CodigoClub] =
[t1].[Codigo]
WHERE [t0].[FechaNacimiento] >= @p0
ORDER BY [t1].[Nombre], [t0].[Nombre]
-- @p0: Input DateTime (Size = 0; Prec = 0; Scale = 0) [01/01/1987 0:00:00]
-- @p1: Input String (Size = 2; Prec = 0; Scale = 0) [ (]
-- @p2: Input String (Size = 1; Prec = 0; Scale = 0) [)]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1

{ Jugador = JAVI MARTINEZ (M), Equipo = ATHLETIC BILBAO }
{ Jugador = AGÜERO (L), Equipo = ATLETICO DE MADRID }
{ Jugador = MESSI (L), Equipo = BARCELONA }
{ Jugador = TREJO (L), Equipo = MALLORCA }
{ Jugador = CRISTIAN PORTILLA (M), Equipo = RACING DE SANTANDER }
{ Jugador = HIGUAIN (L), Equipo = REAL MADRID }
{ Jugador = IMANOL (M), Equipo = REAL SOCIEDAD }
{ Jugador = CRESPO (D), Equipo = SEVILLA }
{ Jugador = DIEGO CAPEL (M), Equipo = SEVILLA }
```

Como puede comprobar, las constantes que aparecen en la consulta integrada son pasadas al motor SQL como parámetros; adicionalmente, operaciones como las concatenaciones, conversiones de tipos, llamadas a funciones, etc. son traducidas a la sintaxis específica del motor de bases de datos con el que se está interactuando, en este caso, SQL Server 2005. La documentación de LINQ to SQL indica qué operaciones se pueden utilizar en las consultas integradas; si nos apartamos de esas especificaciones, obtendremos mensajes de error como "*El método Split no tiene traducción a SQL*".

Veamos ahora un segundo ejemplo en el que se listan los países según el orden descendente de la cantidad de futbolistas de cada país:

```
static void Consulta2()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out;

      var grupos =
         from f in ctx.Futbolista
         group f by f.CodigoPaisNacimiento into tmp
         orderby tmp.Count() descending
         join p in ctx.Pais on tmp.Key equals p.Codigo
         select new { p.Nombre, Cantidad = tmp.Count() };

      Console.WriteLine("Futbolistas por países");
      foreach (var p in grupos)
         Console.WriteLine(p);
   }
}
```

En este caso, el registro de actividad mostrará lo siguiente:

```
Futbolistas por países
SELECT [t2].[Nombre], [t1].[value2] AS [Cantidad]
FROM (
   SELECT COUNT(*) AS [value], COUNT(*) AS [value2],
      [t0].[CodigoPaisNacimiento]
   FROM [dbo].[Futbolista] AS [t0]
   GROUP BY [t0].[CodigoPaisNacimiento]
   ) AS [t1]
INNER JOIN [dbo].[Pais] AS [t2]
   ON [t1].[CodigoPaisNacimiento] = [t2].[Codigo]
ORDER BY [t1].[value] DESC
```

Continúa

```
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1

{ Nombre = ESPAÑA, Cantidad = 304 }
{ Nombre = BRASIL, Cantidad = 37 }
{ Nombre = ARGENTINA, Cantidad = 37 }
{ Nombre = FRANCIA, Cantidad = 16 }
{ Nombre = URUGUAY, Cantidad = 10 }
{ Nombre = PORTUGAL, Cantidad = 9 }
{ Nombre = ITALIA, Cantidad = 8 }
{ Nombre = SERBIA, Cantidad = 5 }
{ Nombre = CAMERUN, Cantidad = 4 }
{ Nombre = DINAMARCA, Cantidad = 3 }
{ Nombre = CHILE, Cantidad = 3 }
{ Nombre = RUMANIA, Cantidad = 3 }
{ Nombre = HOLANDA, Cantidad = 3 }
{ Nombre = MEXICO, Cantidad = 2 }
{ Nombre = GRECIA, Cantidad = 2 }
{ Nombre = COSTA DE MARFIL, Cantidad = 2 }
{ Nombre = BULGARIA, Cantidad = 2 }
{ Nombre = COLOMBIA, Cantidad = 2 }
{ Nombre = ECUADOR, Cantidad = 1 }
{ Nombre = REINO UNIDO, Cantidad = 1 }
{ Nombre = GHANA, Cantidad = 1 }
{ Nombre = BOLIVIA, Cantidad = 1 }
{ Nombre = CANADA, Cantidad = 1 }
{ Nombre = ZAIRE, Cantidad = 1 }
{ Nombre = SUIZA, Cantidad = 1 }
{ Nombre = IRLANDA, Cantidad = 1 }
{ Nombre = ISRAEL, Cantidad = 1 }
{ Nombre = IRAN, Cantidad = 1 }
{ Nombre = ISLANDIA, Cantidad = 1 }
{ Nombre = COREA DEL SUR, Cantidad = 1 }
{ Nombre = LETONIA, Cantidad = 1 }
{ Nombre = MARRUECOS, Cantidad = 1 }
{ Nombre = MALI, Cantidad = 1 }
{ Nombre = MOZAMBIQUE, Cantidad = 1 }
{ Nombre = NIGERIA, Cantidad = 1 }
{ Nombre = VENEZUELA, Cantidad = 1 }
{ Nombre = RUSIA, Cantidad = 1 }
{ Nombre = ESLOVENIA, Cantidad = 1 }
{ Nombre = TURQUIA, Cantidad = 1 }
```

En el siguiente ejemplo de consulta se obtiene una lista de todas las posibles parejas de futbolistas de un país determinado que juegan en un mismo equipo. Para complicar la consulta hemos utilizado un encuentro agrupado, que luego se aplana:

```
static void Consulta3()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out;

      string codPais = "BR";   // el valor de parámetro

      var compys =
         from f1 in ctx.Futbolista
         where f1.Pais.Codigo == codPais
         orderby f1.Pais.Nombre
         join f2 in ctx.Futbolista
            on f1.CodigoPaisNacimiento equals
               f2.CodigoPaisNacimiento into tmp
         from t in tmp
         where f1.Id < t.Id && f1.CodigoClub == t.CodigoClub
         select f1.Nombre + " - " + t.Nombre +
            " (" + f1.Pais.Nombre + ")";

      Console.WriteLine("Compatriotas en un mismo equipo");
      foreach (var p in compys)
         Console.WriteLine(p);
   }
}
```

La consulta que se envía a SQL Server 2005 es:

```
Compatriotas en un mismo equipo
SELECT ((([t0].[Nombre] + @p1) + [t2].[Nombre]) + @p2) +
     [t1].[Nombre]) + @p3 AS [value]
FROM [dbo].[Futbolista] AS [t0]
INNER JOIN [dbo].[Pais] AS [t1]
     ON [t1].[Codigo] = [t0].[CodigoPaisNacimiento]
CROSS JOIN [dbo].[Futbolista] AS [t2]
WHERE ([t0].[Id] < [t2].[Id]) AND ([t0].[CodigoClub] =
[t2].[CodigoClub])
   AND ([t1].[Codigo] = @p0)
   AND ([t0].[CodigoPaisNacimiento] = [t2].[CodigoPaisNacimiento])
ORDER BY [t1].[Nombre]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [BR]
-- @p1: Input String (Size = 3; Prec = 0; Scale = 0) [ - ]
-- @p2: Input String (Size = 2; Prec = 0; Scale = 0) [ (]
-- @p3: Input String (Size = 1; Prec = 0; Scale = 0) [)]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1
```

Si hubiésemos elegido la vía conceptualmente más sencilla del producto cartesiano restringido de la tabla de futbolistas sobre sí misma:

```
var compys =from f1 in ctx.Futbolista
   where f1.CodigoPaisNacimiento == codPais
   orderby f1.Pais.Nombre
   from f2 in ctx.Futbolista
   where f1.CodigoPaisNacimiento ==
      f2.CodigoPaisNacimiento &&
      f1.Id < f2.Id &&
      f1.CodigoClub == f2.CodigoClub
   select f1.Nombre + " - " + f2.Nombre +
      " (" + f1.Pais.Nombre + ")";
```

la sentencia SQL generada habría sido exactamente la misma.

Veamos ahora un ejemplo de una consulta en la que se hace uso de uno de los operadores que no tiene reflejo sintáctico directo en C#, el operador `Take()`. La siguiente consulta produce los cinco futbolistas más viejos de la Liga:

```
using (FutbolDataContext ctx = new FutbolDataContext())
{
   ctx.Log = Console.Out;

   var lista1 =
      (from f in ctx.Futbolista
      orderby f.FechaNacimiento
      select new
      {
         Jugador = f.Nombre + " (" + f.Posicion + ")",
         Equipo = f.Club.Nombre
      }).Take(5);

   Console.WriteLine("Los 5 más viejos");
   foreach (var p in lista1)
      Console.WriteLine(p);
}
```

Nuevamente, observe la manera natural en que la llamada a `Take()` se traduce en una cláusula `TOP` de SQL Server:

```
Los 5 más viejos
SELECT TOP 5 [t2].[value] AS [Jugador], [t2].[Nombre] AS [Equipo]
FROM (
   SELECT (([t0].[Nombre] + @p0) +
```

Continúa

```
      (CONVERT(NVarChar(1),[t0].[Posicion])) + @p1 AS [value],
      [t1].[Nombre], [t0].[FechaNacimiento]
   FROM [dbo].[Futbolista] AS [t0]
   INNER JOIN [dbo].[Club] AS [t1] ON [t1].[Codigo] =
[t0].[CodigoClub]
   ) AS [t2]
ORDER BY [t2].[FechaNacimiento]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [ ()]
-- @p1: Input String (Size = 1; Prec = 0; Scale = 0) [)]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1

{ Jugador = CAÑIZARES (P), Equipo = VALENCIA }
{ Jugador = ALFARO (D), Equipo = RACING DE SANTANDER }
{ Jugador = MERINO (D), Equipo = RECREATIVO }
{ Jugador = PINILLA (L), Equipo = NASTIC TARRAGONA }
{ Jugador = ROMERO (D), Equipo = BETIS }
```

Pondremos aún un último ejemplo en el que esté involucrado uno de los agregados tradicionales, `Count()`:

```
using (FutbolDataContext ctx = new FutbolDataContext())
{
   ctx.Log = Console.Out;

   int cant =
      (from f in ctx.Futbolista
      where f.CodigoPaisNacimiento == "PT"
      select f).Count();

   Console.WriteLine("Portugueses: {0}", cant);
}
```

Aquí vemos que LINQ to SQL es lo bastante listo como para no traerse ningún registro a la máquina local, sino solicitar directamente la cantidad al servidor:

```
SELECT COUNT(*) AS [value]
FROM [dbo].[Futbolista] AS [t0]
WHERE [t0].[CodigoPaisNacimiento] = @p0
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [PT]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1

Portugueses: 9
```

Como muestran los ejemplos anteriores, las sentencias SQL que produce la implementación de LINQ to SQL, además de reflejar con total exactitud la intención de las consultas integradas, son generalmente eficientes; se invita al lector a comprobar esto por sí mismo con ejemplos propios. En cualquier caso, una de las ventajas que aporta la utilización de esta manera de programar contra bases de datos sobre la utilización directa de sentencias SQL es el hecho de dejar la creación de estas sentencias a un generador que seguramente con el tiempo irá ganando cada vez más en "inteligencia", hasta llegar a producir un código SQL de muy alta calidad.

La implementación interna de los métodos extensores que conforman el patrón LINQ para LINQ to SQL es bastante compleja, y su análisis va más allá del alcance de este libro; baste con decir que ha sido creada utilizando también a ese nivel una arquitectura abierta, y que si bien con Visual Studio 2008 vienen incluidos únicamente **proveedores para SQL Server 2000 y SQL Server 2005,** cabe esperar que en un futuro próximo otros fabricantes de bases de datos ofrezcan sus propios proveedores para hacer posibles las consultas integradas de LINQ to SQL sobre sus motores relacionales.

14.4. EL MAPEADO OBJETO/RELACIONAL

Como efecto colateral del proceso de ejecución de una consulta integrada de LINQ to SQL, a partir de las filas recuperadas como resultado de la ejecución de la consulta SQL correspondiente se crean **objetos en memoria** que representan a esas filas. En el caso que las consultas soliciten filas enteras de tablas, que se corresponderán con objetos cuyo tipo es una clase de entidad, estos objetos serán "recordados" en un *pool* interno del que dispone el contexto de datos.

14.4.1. Gestión de la identidad

Apoyándose en los metadatos que tiene a su disposición (específicamente, en la información sobre claves primarias), el contexto de base de datos es capaz de hacer un seguimiento de la identidad de los objetos bajo su control. Esto es, si dentro de un mismo contexto se ejecutan dos consultas diferentes que recuperan como parte de sus conjuntos de resultados una fila común, LINQ to SQL reconocerá correctamente que se trata de una misma fila y asignará un único objeto en memoria para esa fila.

Por ejemplo, suponga que primero solicitamos los clubes de fútbol madrileños:

```
var cons1 =
   from c in ctx.Club
   where c.Ciudad == "MADRID"
   select c;
```

y luego pedimos los clubes con más de diez extranjeros en la plantilla:

```
var cons2 =
   from c in ctx.Club
   where (from f in ctx.Futbolista
      where f.Club.Codigo == c.Codigo &&
         f.CodigoPaisNacimiento != "ES"
      select c).Count() > 10
   select c;
```

Al ejecutar la segunda consulta, el motor de LINQ to SQL reconocerá que Real Madrid y Atlético de Madrid ya han sido cargados previamente y no creará para ellos nuevas instancias de la clase `Club` en el *pool*.

Nota:

En el ejemplo anterior hemos hecho uso de una **subconsulta** o consulta dependiente, que como puede ver se expresan de manera natural en la sintaxis de expresiones de consulta. Compruebe usted mismo la sentencia SQL que genera en este caso el motor de LINQ to SQL.

Este mecanismo de seguimiento de la identidad de los objetos, esencial para cualquier gestor ORM, puede desconectarse opcionalmente asignando `false` a la propiedad de tiempo de ejecución `ObjectTrackingEnabled`, cuyo valor por defecto es verdadero. Podría ser útil desactivarlo, por ejemplo, para aumentar el rendimiento en situaciones de acceso de sólo lectura a una base de datos.

14.5. PROPIEDADES DE NAVEGACIÓN

Si ha tomado nota antes de lo que representan las propiedades de navegación, se habrá dado cuenta de que la primera de las consultas de esta sección (la que produce los futbolistas de 20 años) podría escribirse también así:

```
static void Consulta1b()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out;

      var lista1 =
         from p in ctx.Futbolista
         where p.FechaNacimiento >= new DateTime(1987, 1, 1)
         orderby p.Club.Nombre, p.Nombre
```

Continúa

```
        select new
        {
           Jugador = p.Nombre + " (" + p.Posicion + ")",
           Equipo = p.Club.Nombre
        };

      Console.WriteLine("Consulta 1b");
      foreach (var p in lista1)
         Console.WriteLine(p);
   }
}
```

En este caso, utilizando la propiedad de navegación `Club` que ofrece la clase `Futbolista` se hace posible simplificar en gran medida la consulta, evitando la necesidad de especificar explícitamente el encuentro.

Las propiedades de navegación nos permiten modelar las relaciones entre filas de diferentes tablas de la base de datos mediante árboles de objetos. En el caso de una relación `1:N`, (por ejemplo, la relación entre `Club` y `Futbolista`), estas propiedades se expresan en la clase de entidad "de origen" mediante una instancia de la colección genérica `EntitySet<T>` (en nuestro ejemplo, `EntitySet<Futbolista>`), mientras que en la clase de entidad "de destino" se representan mediante una referencia escalar de tipo `EntityRef<T>` (en el mismo ejemplo, la clase `Futbolista` tendrá una propiedad de tipo `EntityRef<Club>`).

14.5.1. Gestión de la carga de las propiedades de navegación

Una propiedad lógica del contexto de datos, `DeferredLoadingEnabled`, con valor `true` de manera predeterminada, permite controlar mediante código si la carga de las filas asociadas a las propiedades que se generan para las relaciones entre tablas se debe producir de manera diferida (según demanda) o de manera inmediata. En el estado predeterminado, los objetos correspondientes a una propiedad de navegación sólo se cargarán en memoria en caso de que la consulta integrada que especifiquemos haga uso de esa propiedad; si se asigna `false` a la propiedad, la carga de estos objetos dependientes se producirá tan pronto como se recupere cada fila de la tabla "principal".

14.6. CONSULTAS DINÁMICAS

La mayor parte de las consultas que se ejecutan en una aplicación tradicional no son consultas estáticas; nos hacen falta los impagos de más de 60 días, los clientes con más de 1.000 euros de pedidos durante el año, etc. Ya en uno de los ejemplos de la sección anterior se mostraba lo fácil que es incorporar a nuestras consultas integradas los valores de las variables que están en ámbito (variable `codPais` en `Consulta3()`); el hecho de que LINQ to SQL utilice parámetros para integrar

estos valores en el código SQL generado nos garantiza, además de la eficiencia, la inmunidad ante posibles ataques de inyección de SQL.

Pero con mucha frecuencia necesitamos más que consultas parametrizadas: son casos en los que no sólo pueden cambiar ciertos valores a incorporar en la consulta, sino en los que debemos generar dinámicamente partes de la consulta o la consulta entera. Para resolver este problema, LINQ to SQL nos ofrece dos enfoques diferentes, que a continuación presentaremos.

14.6.1. El método ExecuteQuery<T>()

La vía más general para ejecutar una consulta arbitraria contra un contexto de datos de LINQ to SQL es utilizando el método genérico `ExecuteQuery<T>()`, que ejecuta la consulta y nos devuelve sus resultados como un objeto de tipo `IEnumerable<T>`. Este método acepta una cantidad variable de argumentos; el primero, que es obligatorio, es una cadena con la sentencia SQL que queremos ejecutar; los siguientes, opcionales, suministran los valores para los posibles parámetros de la consulta, que se representan en forma de "comodines" en el primer argumento, con una sintaxis similar a la que utiliza **`string.`**`Format()`. Un ejemplo nos aclarará esta definición:

```
static void ConsultaDinamica1()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out;

      string ciudad = "MADRID";

      var cm = ctx.ExecuteQuery<Club>(
         "SELECT * FROM Club WHERE Ciudad = {0}", ciudad);
      // la consulta que se envía al servidor es:
      //       SELECT * FROM Club WHERE Ciudad = @p0
      // donde el valor de @p0 es MADRID

      Console.WriteLine("Clubes madrileños");
      foreach (var c in cm)
         Console.WriteLine(c.Nombre);

      var ciudades = ctx.ExecuteQuery<Tmp>(
         @"SELECT Ciudad, COUNT(*) AS CantClubes FROM Club
            GROUP BY Ciudad
            ORDER BY 2 DESC");

      Console.WriteLine("Clubes por ciudad");
      foreach (var c in ciudades)
         Console.WriteLine(c.Ciudad + " - " + c.CantClubes);
   }
}
```

Tratándose de un método tan general, el compilador no puede darnos ayuda de ningún tipo con respecto al tipo del método genérico, que debemos especificar. Si la el resultado que producirá la sentencia no coincide estructuralmente con ninguna de las clases de entidad, debemos definir un tipo adecuado para almacenar los resultados de la consulta. Por ejemplo, para que la segunda llamada a `ExecuteQuery<T>()` funcione correctamente, hemos definido de antemano el siguiente tipo:

```
class Tmp
{
   public string Ciudad { get; set; }
   public int CantClubes { get; set; }
}
```

Aquí es crucial que la clase tenga propiedades cuyos nombres coincidan con las columnas del conjunto de resultados de la consulta.

14.6.2. La clase DynamicQueryable

Conscientes de la frecuencia con la que se presenta la necesidad de generar consultas dinámicamente, los creadores de LINQ to SQL han creado una API especializada para esta tarea. Aunque no sería de extrañar que en un futuro alguna variación de esta API se incorporara oficialmente a .NET Framework, de momento se nos ofrece de forma externa, con código fuente incluido, como parte de uno de los ejemplos incluidos en Visual Studio 2008 llamado precisamente `DynamicQuery`.

El núcleo de esta API lo constituye una clase estática llamada `DynamicQueryable`, que contiene métodos que extienden `IQueryable`, la clase base no genérica de `IQueryable<T>`, con métodos llamados `Where()`, `OrderBy()`, `Select()`, etc. que en lugar de recibir como parámetros árboles de expresiones reciben **cadenas de caracteres,** que internamente esos métodos transforman en árboles de expresiones y pasan al motor de LINQ to SQL. De este modo, podríamos conformar de forma dinámica prácticamente cualquier sentencia SQL.

Entre las desventajas de esta API debemos señalar el hecho de que requieren la utilización de sintaxis de llamadas explícitas a los métodos. Además, como veremos en el ejemplo que sigue, algunos de los métodos extensores (por ejemplo, `Select()`) requieren el uso de una sintaxis no habitual, que no es ni SQL ni C#.

El siguiente código genera dinámicamente por esta vía una sentencia SQL que produce los datos de los clubes de fútbol madrileños:

```
static void ConsultaDinamica2()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out;
```

Continúa

```
      var cm = ctx.Club.
         Where("Ciudad == \"MADRID\"").
         OrderBy("Nombre").
         Select("New(Codigo, Nombre)");

      foreach (var c in cm)
         Console.WriteLine(c);
   }
}
```

El resultado en la consola será:

```
SELECT [t0].[Codigo], [t0].[Nombre]
FROM [dbo].[Club] AS [t0]
WHERE [t0].[Ciudad] = @p0
ORDER BY [t0].[Ciudad]
-- @p0: Input String (Size = 6; Prec = 0; Scale = 0) [MADRID]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1

{Codigo=ATM, Nombre=ATLETICO DE MADRID}
{Codigo=GET, Nombre=GETAFE}
{Codigo=RMA, Nombre=REAL MADRID}
```

Una documentación completa sobre el uso de esta API se ofrece como parte del ejemplo `DynamicQuery` que acompaña a Visual Studio 2008.

14.7. ACTUALIZACIÓN DE DATOS

El contexto de datos no sólo es capaz de crear y gestionar los objetos asociados a las filas recuperadas de la base de datos; también ofrece los mecanismos necesarios para "memorizar" los cambios que se vayan efectuando sobre estos objetos y generar las sentencias SQL necesarias para aplicar esos cambios a la base de datos cuando se llame a su método `SubmitChanges()`. A continuación, se muestra un ejemplo sencillo de inserción, modificación y borrado de datos:

```
private static void Actualizaciones()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out;

      // actualizar un futbolista
```

Continúa

```
      Futbolista ronie = ctx.Futbolista.
         First(x => x.Nombre == "RONALDO");

      if (ronie != null)
         ronie.Nombre = "RONALDO LUIZ NAZARIO DA LIMA";

      // insertar un país
      ctx.Pais.Add(new Pais { Codigo = "CU", Nombre = "CUBA" });

      // borrar un país
      Pais tv = ctx.Pais.Single(x => x.Codigo == "TV");
      ctx.Pais.Remove(tv);

      try
      {
         // aplicar cambios a la base de datos
         ctx.SubmitChanges();
      }
      catch (Exception x)
      {
         Console.WriteLine("ERROR: " + x.Message);
      }
   }
}
```

Como puede observar, las modificaciones se aplican directamente sobre los objetos que representan a las filas, mientras que para las inserciones y borrados se utilizan los métodos `Add()` y `Remove()` de la clase de entidad. A continuación, un extracto de la salida en la consola, que muestra las sentencias SQL que se envían a la base de datos.

```
UPDATE [dbo].[Futbolista]
SET [Nombre] = @p7
WHERE ([Id] = @p0) AND ([Nombre] = @p1) AND ([Sexo] = @p2) AND
   ([FechaNacimiento] = @p3) AND ([CodigoPaisNacimiento] = @p4) AND
   ([CodigoClub] = @p5) AND ([Posicion] = @p6)
-- @p0: Input Int32 (Size = 0; Prec = 0; Scale = 0) [495]
-- @p1: Input String (Size = 7; Prec = 0; Scale = 0) [RONALDO]
-- @p2: Input StringFixedLength (Size = 1; Prec = 0; Scale = 0) [V]
-- @p3: Input DateTime (Size = 0; Prec = 0; Scale = 0)
[22/09/1976 0:00:00]
-- @p4: Input String (Size = 2; Prec = 0; Scale = 0) [BR]
-- @p5: Input String (Size = 3; Prec = 0; Scale = 0) [RMA]
```

Continúa

```
-- @p6: Input StringFixedLength (Size = 1; Prec = 0; Scale = 0) [L]
-- @p7: Input String (Size = 28; Prec = 0; Scale = 0) [RONALDO
LUIZ NAZARIO DA LIMA]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1


INSERT INTO [dbo].[Pais]([Codigo], [Nombre]) VALUES (@p0, @p1)
-- @p0: Input AnsiStringFixedLength (Size = 2; Prec = 0; Scale = 0) [CU]
-- @p1: Input AnsiString (Size = 4; Prec = 0; Scale = 0) [CUBA]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1


DELETE FROM [dbo].[Pais] WHERE ([Codigo] = @p0) AND ([Nombre] = @p1)
-- @p0: Input AnsiStringFixedLength (Size = 2; Prec = 0; Scale = 0) [TV]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [TUVALU]
-- Context: SqlProvider(Sql2005) Build: 3.5.20706.1
```

14.7.1. Utilización de transacciones

Si se da cuenta de que no hay ninguna transacción activa en ese momento, la API de LINQ to SQL engloba las llamadas a `SubmitChanges()` dentro una transacción. De esta manera, los cambios registrados en el contexto de datos se aplicarán bajo el principio "o todos o ninguno". No obstante, el programador puede crear su propia transacción en caso de que ese comportamiento predeterminado no le satisfaga. Por ejemplo, si quisiéramos ejecutar un procedimiento almacenado dentro del mismo ámbito transaccional que las actualizaciones, podríamos hacer lo siguiente:

```
using (TransactionScope ts = new TransactionScope())
{
   ctx.ExecuteCommand("exec sp_Cambios");
   ctx.SubmitChanges();
   ts.Complete();
}
```

Nota:

`ExecuteCommand()` permite ejecutar directamente sentencias (con posibles parámetros) que no devuelvan conjuntos de resultados contra la base de datos subyacente al contexto, de modo similar a como lo hace el método `ExecuteQuery()` presentado antes. `ExecuteCommand()` puede utilizarse para llamar a procedimientos almacenados "de acción", como en el ejemplo, o para ejecutar sentencias `INSERT`, `UPDATE` o `DELETE` (aunque lo más indicado es valernos de los mecanismos de actualización que ofrece LINQ to SQL).

Otra posible vía sería utilizar una transacción de ADO.NET, a través de la propiedad `Transaction` del contexto de datos, que LINQ to SQL ofrece como otro "guiño" a la compatibilidad con ADO.NET. En tales casos, deberemos utilizar un esquema similar al siguiente:

```
ctx.Transaction = ctx.Connection.BeginTransaction();
try
{
   ctx.ExecuteCommand("exec sp_Cambios"); // si hiciera falta
   ctx.SubmitChanges();
   ctx.Transaction.Commit();
}
catch
{
   ctx.Transaction.Rollback();
   throw;
}
finally
{
   ctx.Transaction = null;
}
```

14.7.2. Gestión de la concurrencia

La API de LINQ to SQL viene equipada con los mecanismos necesarios para que el programador pueda configurar del modo que estime más conveniente para su aplicación la gestión de la concurrencia, y en particular para que pueda resolver los conflictos que se producen en caso de que varios usuarios intenten simultáneamente realizar cambios sobre un mismo registro.

Comentaremos aquí brevemente los principales aspectos relacionados con esto:

- Antes de enviar a la base de datos cualquier solicitud de modificación de un registro, se comprueba si la fila a actualizar no ha sido modificada por otro usuario después del momento en que quien intenta actualizarla la leyó de la base de datos. Esta filosofía de trabajo (en contraposición a la que se basa en imponer bloqueos a los registros que se leen para impedir su modificación por otros usuarios) se conoce como **concurrencia optimista.**
- La comprobación de si la fila ha sido modificada después que se leyó o no se lleva a cabo, como es natural, comparando los valores originales de las columnas de esa fila con los actuales. Una columna se utilizará o no en esta comprobación en dependencia del valor que se asigne al parámetro `UpdateCheck` del atributo `Column` asociado a la columna en la clase de entidad. Los posibles valores son `UpdateCheck.Always` (utilizar la columna siempre, el valor por defecto), `UpdateCheck.Never` (no utilizar la columna en las comprobaciones) y `UpdateCheck.WhenChanged` (utilizarla sólo cuando el valor de la columna haya sido modificado).

- Al llamar al método `SubmitChanges()`, se le puede pasar un parámetro que indique si queremos que la aplicación de las actualizaciones se detenga tan pronto se produzca el primer conflicto (`ConflictMode.FailOnFirstConflict`), o si queremos que el lote de actualizaciones se continúe intentando hasta el final y los conflictos se nos devuelvan agrupados (`ConflictMode.ContinueOnConflict`).
- Al producirse uno o más conflictos de concurrencia, nuestra aplicación recibirá una excepción de tipo `ChangeConflictException`, que incluirá todos los detalles relacionados con los problemas detectados. Estos detalles se almacenan también en la propiedad `ChangeConflicts` del contexto.
- Además de poder implementar mecanismos de reconciliación a medida, el programador puede hacer uso de políticas estandarizadas para todos los conflictos detectados (a través del método `ResolveAll()` de la propiedad `ChangeConflicts`) o para cada conflicto individual (mediante el método `Resolve()` de éstos). Estos métodos reciben un parámetro del tipo enumerado `RefreshMode`, en el que debemos indicar la política a aplicar.

El siguiente código ejemplifica los puntos antes comentados:

```
private static void ActualizacionConcurrente()
{
   using (FutbolDataContext ctx = new FutbolDataContext())
   {
      ctx.Log = Console.Out;

      // actualizar un futbolista
      Futbolista ronie = ctx.Futbolista.
         First(x => x.Nombre == "RONALDO");
      if (ronie != null)
         ronie.Nombre = "RONALDO (REAL MADRID)";

      using (FutbolDataContext ctx2 = new FutbolDataContext())
      {
         ctx2.Log = Console.Out;

         // actualizar un futbolista
         Futbolista ronie2 = ctx2.Futbolista.
            First(x => x.Nombre == "RONALDO");
         if (ronie2 != null)
            ronie2.Nombre = "RONALDO (INTER MILAN)";

         try
         {
            // aplicar cambios de 1° contexto
            ctx.SubmitChanges();
```

Continúa

```
            // aplicar cambios de 2° contexto
            ctx2.SubmitChanges(ConflictMode.ContinueOnConflict);
         }
         catch (ChangeConflictException)
         {
            // mantener los valores establecidos localmente
            ctx2.ChangeConflicts.ResolveAll(
               RefreshMode.KeepChanges);
            // reintentar los cambios
            ctx2.SubmitChanges(ConflictMode.FailOnFirstConflict);
         }
      }
}
```

14.7.3. Personalización de las actualizaciones

De manera predeterminada, LINQ to SQL genera automáticamente las sentencias SQL que se utilizan a la hora de efectuar las actualizaciones sobre la base de datos. No obstante, deja la puerta abierta para que un programador pueda personalizar la manera en que estas actualizaciones se aplicarán; es común, por ejemplo, utilizar para esas tareas procedimientos almacenados definidos de antemano.

El mecanismo para personalizar las actualizaciones en LINQ to SQL se basa también en los métodos parciales. Básicamente, al generar el contexto de datos para cada clase de entidad se definen tres métodos parciales, llamados `InsertXXX`, `UpdateXXX` y `DeleteXXX`, donde `XXX` es el nombre de la clase de entidad. Si el desarrollador suministra implementaciones para esos métodos parciales, éstos serán llamados a la hora de aplicar las inserciones, modificaciones y borrados, respectivamente. Estas implementaciones deben programarse en una clase parcial diferente, y nunca en el mismo fichero en el que se genera el código automático, ya que éste puede ser regenerado, con lo que se perderían nuestros aportes. Por ejemplo:

```
// fichero FUTBOL_Club.cs
partial class FutbolDataContext
{
   partial void UpdateClub(Club actual)
   {
      int cant = this.ExecuteCommand("UPDATE Club" +
         " SET Nombre={1}, Ciudad={2} WHERE Codigo = {0}",
         actual.Codigo,
         actual.Nombre,
         actual.Ciudad);
      if (cant < 1)
```

Continúa

```
        throw new Exception("Error al actualizar club");
    }

    partial void InsertClub(Club p)
    {
        // funcionalidad de inserción
    }

    partial void DeleteClub(Club p)
    {
        // funcionalidad de borrado
    }
}
```

En este caso, nos hemos hecho cargo de las actualizaciones sobre la tabla `Club`, aunque probablemente para hacer lo mismo que se haría por defecto. Y tal cual está el código (con los métodos `InsertClub()` y `DeleteClub()` sin código alguno asociado), las inserciones y borrados simplemente no se aplicarán sobre la base de datos.

14.8. EJECUCIÓN DE PROCEDIMIENTOS Y FUNCIONES

Como habrá deducido desde el principio del capítulo, LINQ to SQL ofrece la posibilidad de realizar llamadas a los procedimientos almacenados y funciones definidas por el usuario de la base de datos. Ya antes hemos mencionado una forma de hacerlo, utilizando el método `ExecuteCommand()` (o `ExecuteQuery()`, si devuelven un conjunto de resultados). Pero el diseñador de contextos de datos nos permite simplificar aún mucho más la tarea, incorporando a la clase de contexto métodos especializados que se encargan de la transformación CLR-Transact SQL de los parámetros de entrada, de la ejecución de la llamada y de la transformación inversa de los resultados.

A continuación, se presentan ejemplos de llamadas al procedimiento almacenado y la función definida por el usuario de la base de datos sobre fútbol que hemos mapeado a métodos de nuestro contexto de datos.

```
using (FutbolDataContext ctx = new FutbolDataContext())
{
    // llamada a UDF
    Console.WriteLine("En total hay " +
        ctx.FutbolistasDeUnPais("FR") + " franceses.");

    // llamada a SP
    ctx.InsertarFutbolista(
        "METZELDER", new DateTime(1979, 4, 11), "DE", "RMA", 'D');
}
```

14.9. COMBINANDO TECNOLOGÍAS

Casi que como colofón al libro, presentaremos un pequeño ejemplo de código en el que se combinan dos de las tecnologías que venimos describiendo en estos últimos capítulos, LINQ to XML y LINQ to SQL. No sólo cada una de ellas aporta ventajas por separado; su utilización conjunta también ofrece sustanciosas ganancias en productividad para el programador.

Suponga que deseamos generar a partir de nuestra base de datos un fichero XML con la información de todos los clubes de primera división, incluyendo a su vez dentro de cada club a todos los futbolistas de su plantilla. Pues bien, una sola sentencia bastaría:

```
using (FutbolDataContext ctx = new FutbolDataContext())
{
   XElement liga =
      new XElement("Liga",
         from c in ctx.Club orderby c.Codigo
         select new XElement("Club",
            new XAttribute("Codigo", c.Codigo),
            new XElement("Nombre", c.Nombre),
            new XElement("Ciudad", c.Ciudad),
            new XElement("Jugadores",
               from f in c.Futbolistas
               //from f in ctx.Futbolista
               //where f.CodigoClub == c.Codigo
         select new XElement("Jugador",
            new XAttribute("Codigo", f.Id),
            new XElement("Nombre", f.Nombre),
            new XElement("Posicion", f.Posicion)))));
   liga.Save("Liga2006.XML");
}
```

14.10. LA CLASE LINQDATASOURCE

Antes de terminar, introduciremos un nuevo componente que ha sido añadido al cuadro de herramientas de Visual Studio 2008 y que podrá sernos de gran utilidad a la hora de enlazar a datos los controles visuales de nuestras aplicaciones Web. Se trata de `LinqDataSource`, que se inserta de manera armoniosa en la jerarquía de clases a la que pertenecen `SqlDataSource` y `XmlDataSource`, entre otras, para hacer posible que conectemos nuestros controles al resultado de consultas de LINQ to SQL.

Para hacer uso de este componente, basta con arrastrarlo sobre cualquier página Web de una aplicación a la que hayamos incorporado uno o más contextos de datos (el directorio especial `App _ Code` es el sitio más adecuado para ello). El asistente de configuración de la fuente de datos inicialmente nos permitirá elegir el contexto:

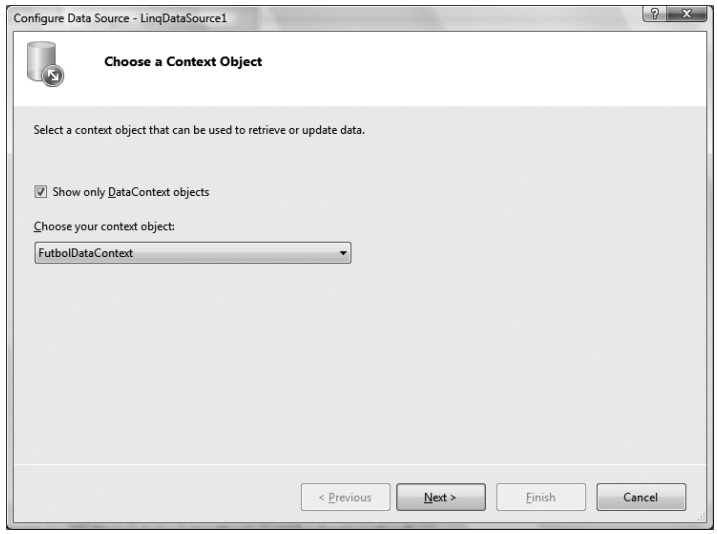


Figura 14.4.

para luego solicitarnos la tabla y las condiciones de filtro, ordenación, etc:

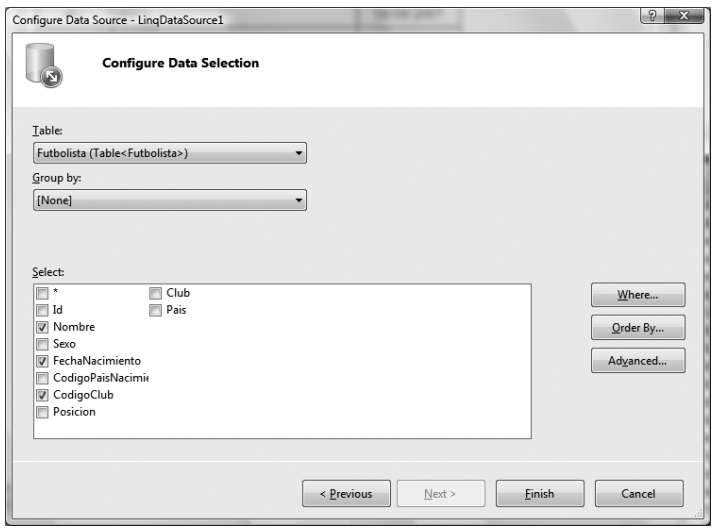
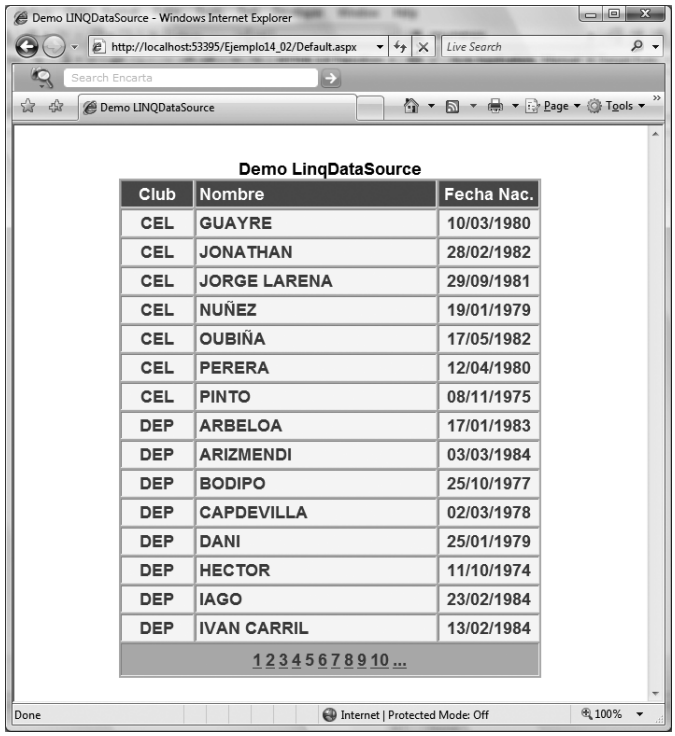


Figura 14.5.

Ya a partir de este punto, todo el trabajo se hace muy similar a si estuviéramos trabajando con una fuente de datos de cualquier otro tipo: podremos, por ejemplo, especificar parámetros en la condición de selección y asociar éstos a valores provenientes de otros controles, la sesión, la cadena de consulta, etc. Una vez definida la fuente de datos LINQ, podremos conectar a ella un `GridView`, y configurar visualmente esta rejilla para que muestre la información en el formato más adecuado,

permita la ordenación y paginación de los datos, así como la inserción, modificación y borrado de filas, entre otras posibilidades. Encontrará mucha más información sobre esos temas en el libro *Programación Web con Visual Studio y ASP.NET 2.0,* de José Manuel Alarcón.

En la siguiente imagen se muestra a la pequeña aplicación de ejemplo en ejecución, mostrando jugadores de los equipos gallegos:



14.11. CONCLUSIÓN

En este capítulo hemos presentado los fundamentos de **LINQ to SQL,** una tecnología que hará posible acceder a los datos almacenados en bases de datos relacionales (tanto para consulta como actualización) sin necesidad de codificar explícitamente sentencias SQL en el código de nuestras aplicaciones. LINQ to SQL debe verse como una parte integral de la tecnología LINQ, pero a la vez como un componente de la siguiente versión de **ADO.NET,** que tiene como objetivo fundamental elevar sustancialmente el nivel de la programación del acceso a datos mediante la definición de modelos conceptuales basados en el concepto de **entidad.**

Índice analítico