Mayur Ramgir, Nick Samoylov

Java: High-Performance Apps with Java 9

Rapid Learning Solution

Boost your application's performance with the new features of Java 9

Packt

Java: High-Performance Apps with Java 9

Optimize the powerful techniques of Java 9 to boost your application's performance

Mayur Ramgir

Nick Samoylov

BIRMINGHAM - MUMBAI

Java: High-Performance Apps with Java 9

First published: March 2018

Production reference: 1080318

Published by Packt Publishing Ltd.
Livery Place, 35 Livery Street
Birmingham B3 2PB, UK.

ISBN: 978-1-78913-051-5

`www.packtpub.com`

Credits

This book is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt product:

- *Java 9 High Performance* by *Mayur Ramgir* and *Nick Samoylov*

Meet Your Experts

We have the best works of the following esteemed authors to ensure that your learning journey is smooth:

Mayur Ramgir has more than 16 years of experience in the software industry, working at various levels. He is a Sun certified Java programmer and Oracle certified SQL database expert. He completed an MS in computational science and engineering at Georgia Tech, USA (rank 7th in the world for computer science), and an M.Sc. in multimedia application and virtual environments at University of Sussex, UK. He has also attended various universities for other degrees and books, such as MIT for applied software security, and University of Oxford for system and software security. He is the CEO of a software company, Zonopact, Inc. headquartered in Boston, USA, which specializes in bringing innovative applications based on AI, robotics, big data, and more. He has single-handedly developed Zonopact's flagship product, Clintra (B2B-integrated AI-assisted business management software). He is also the inventor of two patent pending technologies, ZPOD (an automated cloud-based medical kiosk system) and ZPIC (an AI-enabled robotic in-car camera system).

Nick Samoylov is graduated as an engineer-physicist from Moscow Institute of Physics and Technologies and has even worked as a theoretical physicist. He has learned programming as a tool for testing his mathematical models using FORTRAN and C++. After the demise of the USSR, Nick created and successfully ran a software company, but was forced to close it under the pressure of governmental and criminal rackets. Nick adopted Java in 1997 and used it for his work as a software developer-contractor for a variety of companies, including BEA Systems, Warner Telecom, and Boeing. Nick's current projects are related to machine learning and developing a highly scalable system of microservices using non-blocking reactive technologies, including Vert.x, RxJava, and RESTful web services on Linux deployed in a cloud.

Table of Contents

Preface

This book is about Java 9 which is one of the most popular application development languages. The latest released version Java 9 comes with a host of new features and new APIs with lots of ready to use components to build efficient and scalable applications. Streams, parallel and asynchronous processing, multithreading, JSON support, reactive programming, and microservices comprise the hallmark of modern programming and are now fully integrated into the JDK.

So, if you want to take you Java knowledge to another level and want to improve your application's performance, you are in the right path.

What's in It for Me?

Maps are vital for your journey, especially when you're holidaying in another continent. When it comes to learning, a roadmap helps you in giving a definitive path for progressing towards the goal. So, here you're presented with a roadmap before you begin your journey.

This book is meticulously designed and developed in order to empower you with all the right and relevant information on Java. We've created this Learning Path for you that consists of five lessons:

Lesson 1, Learning Java 9 Underlying Performance Improvements, covers the exciting features of Java 9 that will improve your application's performance. It focuses on modular development and its impact on an application's performance.

Lesson 2, Tools for Higher Productivity and Faster Application, describes two new tools added in Java 9--JShell and Ahead-of-Time (AOT) compiler--that boost your productivity and also improve the overall performance of your applications.

Lesson 3, Multithreading and Reactive Programming, shows how to monitor Java applications programmatically using command-line tools. You will also explore how to improve the application performance via multithreading and how to tune the JVM itself after learning about the bottlenecks through monitoring.

Lesson 4, Microservices, describes the solution many leaders of the industry have adopted while addressing flexible scaling under the load. It talks about adding more workers by splitting the application into several microservices, each deployed independently and each using multiple threads and reactive programming for better performance, response, scalability, and fault-tolerance.

Lesson 5, *Making Use of New APIs to Improve Your Code*, describes improvements in the programming tools, including stream filters, a stack-walking API, the new convenient static factory methods for creating immutable collections, a new powerful CompletableFuture class in support of asynchronous processing, and the JDK 9 stream API improvements.

What Will I Get from This Book?

- Familiarize with modular development and its impact on performance
- Learn various string-related performance improvements, including compact string and indify string concatenation
- Explore various underlying compiler improvements, such as tiered attribution and Ahead-of-Time (AOT) compilation
- Learn security manager improvements
- Understand enhancements in graphics rasterizers
- Use of command-line tools to speed up application development
- Learn how to implement multithreading and reactive programming
- Build microservices in Java 9
- Implement APIs to improve application code

Prerequisites

This book is for Java developers who would like to build reliable and high-performance applications. Some of the prerequisites that is required before you begin this book are:

- Prior Java programming knowledge is assumed

1 Learning Java 9 Underlying Performance Improvements

Just when you think you have a handle on lambdas and all the performance-related features of Java 8, along comes Java 9. What follows are several of the capabilities that made it into Java 9 that you can use to help improve the performance of your applications. These go beyond byte-level changes like for string storage or garbage collection changes, which you have little control over. Also, ignore implementation changes like those for faster object locking, since you don't have to do anything differently and you automatically get these improvements. Instead, there are new library features and completely new command-line tools that will help you create apps quickly.

In this lesson, we will cover the following topics:

- Modular development and its impact on performance
- Various string-related performance improvements, including compact string and indify string concatenation
- Advancement in concurrency
- Various underlying compiler improvements, such as tiered attribution and **Ahead-of-Time** (**AOT**) compilation
- Security manager improvements
- Enhancements in graphics rasterizers

Introducing the New Features of Java 9

In this lesson, we will explore many under the cover improvements to performance that you automatically get by just running your application in the new environment. Internally, string changes also drastically reduce memory footprint requirements for times when you don't need full-scale Unicode support in your character strings. If most of your strings can be encoded either as ISO-8859-1 or Latin-1 (1 byte per character), they'll be stored much more efficiently in Java 9. So, let's dive deep into the core libraries and learn the underlying performance improvements.

Modular Development and Its Impact

In software engineering, modularity is an important concept. From the point of view of performance as well as maintainability, it is important to create autonomous units called **modules**. These modules can be tied together to make a complete system. The modules provides encapsulation where the implementation is hidden from other modules. Each module can expose distinct APIs that can act as connectors so that other modules can communicate with it. This type of design is useful as it promotes loose coupling, helps focus on singular functionality to make it cohesive, and enables testing it in isolation. It also reduces system complexity and optimizes application development process. Improving performance of each module helps improving overall application performance. Hence, modular development is a very important concept.

I know you may be thinking, wait a minute, isn't Java already modular? Isn't the object-oriented nature of Java already providing modular operation? Well, object-oriented certainly imposes uniqueness along with data encapsulation. It only recommends loose coupling but does not strictly enforce it. In addition, it fails to provide identity at the object level and also does not have any versioning provision for the interfaces. Now you may be asking, what about JAR files? Aren't they modular? Well, although JARs provide modularization to some extent, they don't have the uniqueness that is required for modularization. They do have a provision to specify the version number, but it is rarely used and also hidden in the JAR's manifest file.

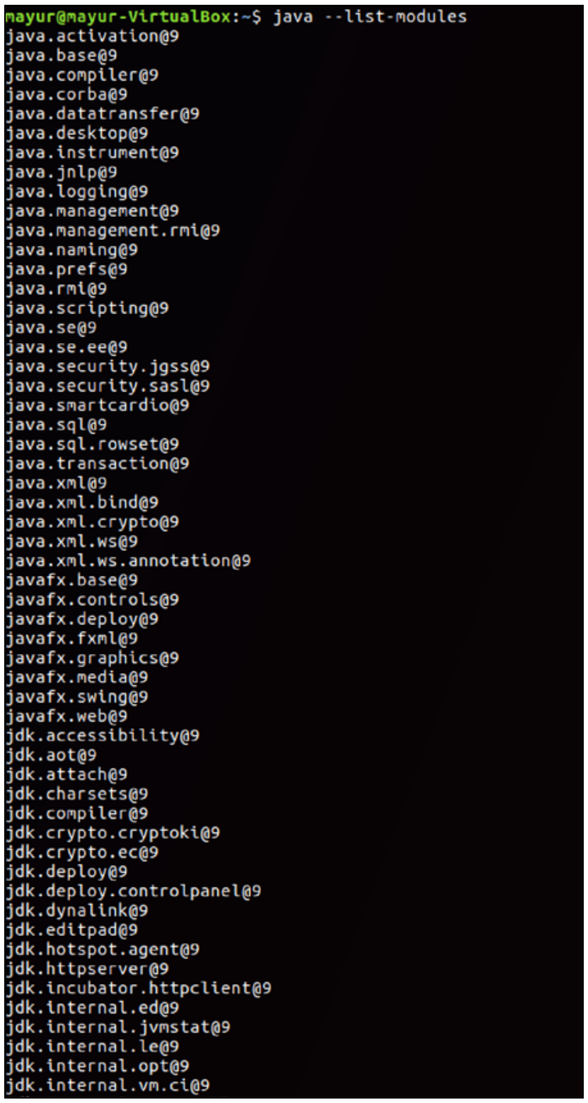
So we need a different design from what we already have. In simple terms, we need a modular system in which each module can contain more than one package and offers robust encapsulation compared to the standard JAR files.

This is what Java 9's modular system offers. In addition to this, it also replaces the fallible classpath mechanism by declaring dependencies explicitly. These enhancements improve the overall application performance as developers can now optimize the individual self-contained unit without affecting the overall system.

This also makes the application more scalable and provides high integrity.

Let's look at some of the basics of the module system and how it is tied together. To start off with, you can run the following commands to see how the module system is structured:

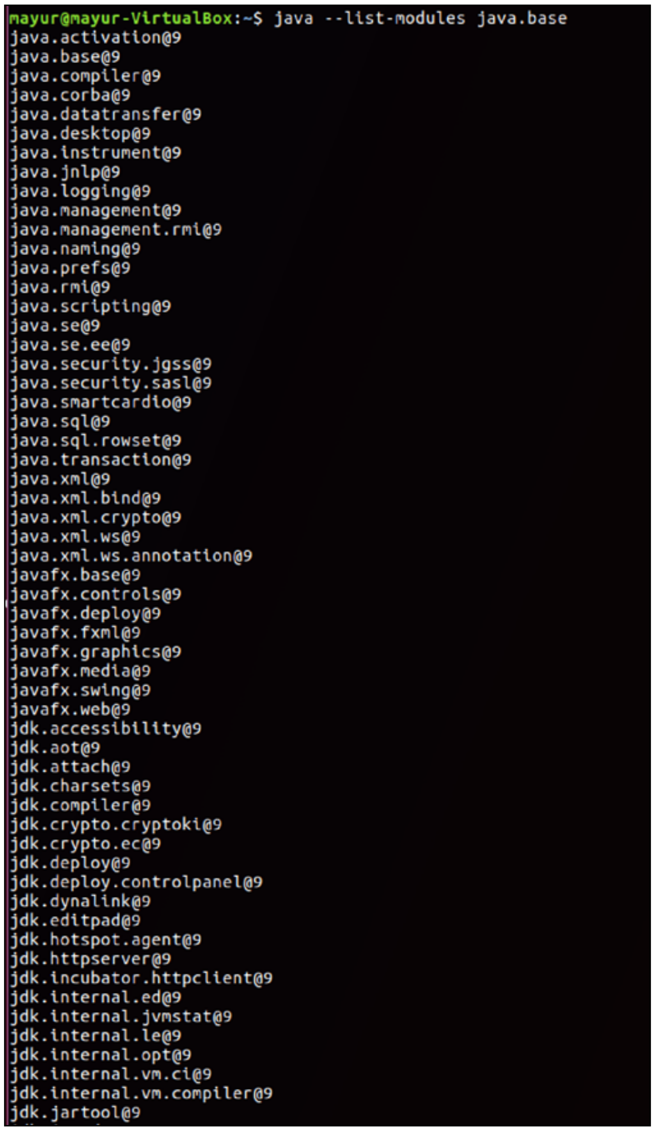
`$java --list-modules`



```
mayur@mayur-VirtualBox:~$ java --list-modules
java.activation@9
java.base@9
java.compiler@9
java.corba@9
java.datatransfer@9
java.desktop@9
java.instrument@9
java.jnlp@9
java.logging@9
java.management@9
java.management.rmi@9
java.naming@9
java.prefs@9
java.rmi@9
java.scripting@9
java.se@9
java.se.ee@9
java.security.jgss@9
java.security.sasl@9
java.smartcardio@9
java.sql@9
java.sql.rowset@9
java.transaction@9
java.xml@9
java.xml.bind@9
java.xml.crypto@9
java.xml.ws@9
java.xml.ws.annotation@9
javafx.base@9
javafx.controls@9
javafx.deploy@9
javafx.fxml@9
javafx.graphics@9
javafx.media@9
javafx.swing@9
javafx.web@9
jdk.accessibility@9
jdk.aot@9
jdk.attach@9
jdk.charsets@9
jdk.compiler@9
jdk.crypto.cryptoki@9
jdk.crypto.ec@9
jdk.deploy@9
jdk.deploy.controlpanel@9
jdk.dynalink@9
jdk.editpad@9
jdk.hotspot.agent@9
jdk.httpserver@9
jdk.incubator.httpclient@9
jdk.internal.ed@9
jdk.internal.jvmstat@9
jdk.internal.le@9
jdk.internal.opt@9
jdk.internal.vm.ci@9
```

If you are interested in a particular module, you can simply add the module name at the end of the command, as shown in the following command:

```
$java --list-modules java.base
```

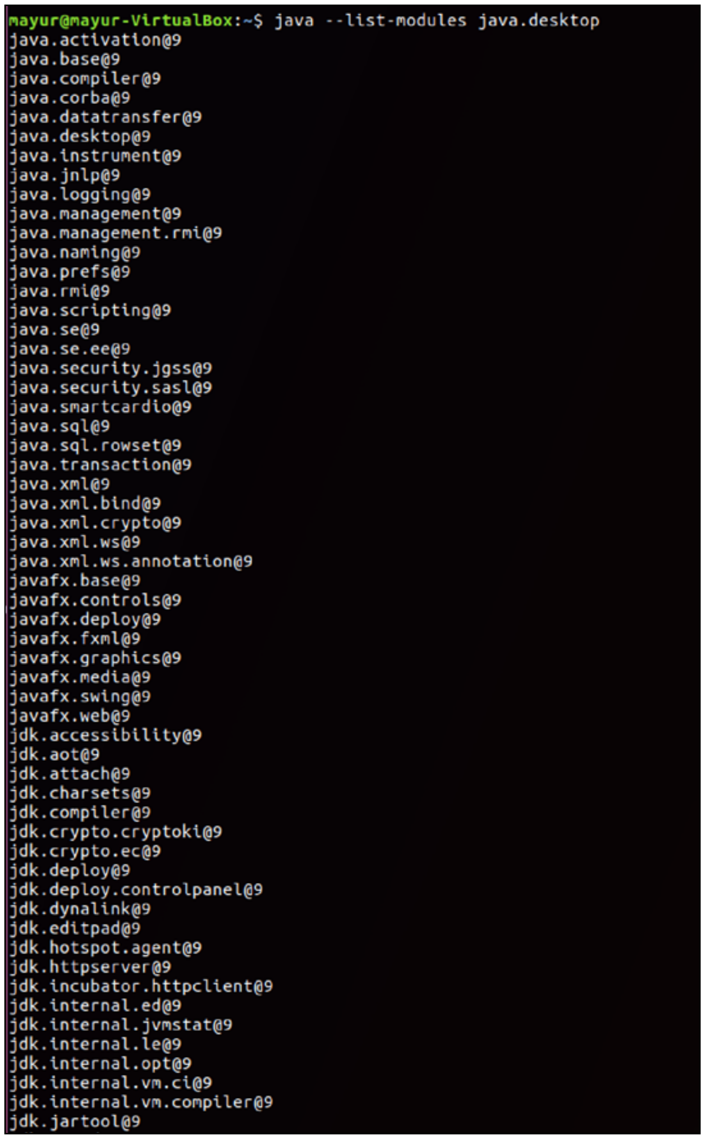


```
mayur@mayur-VirtualBox:~$ java --list-modules java.base
java.activation@9
java.base@9
java.compiler@9
java.corba@9
java.datatransfer@9
java.desktop@9
java.instrument@9
java.jnlp@9
java.logging@9
java.management@9
java.management.rmi@9
java.naming@9
java.prefs@9
java.rmi@9
java.scripting@9
java.se@9
java.se.ee@9
java.security.jgss@9
java.security.sasl@9
java.smartcardio@9
java.sql@9
java.sql.rowset@9
java.transaction@9
java.xml@9
java.xml.bind@9
java.xml.crypto@9
java.xml.ws@9
java.xml.ws.annotation@9
javafx.base@9
javafx.controls@9
javafx.deploy@9
javafx.fxml@9
javafx.graphics@9
javafx.media@9
javafx.swing@9
javafx.web@9
jdk.accessibility@9
jdk.aot@9
jdk.attach@9
jdk.charsets@9
jdk.compiler@9
jdk.crypto.cryptoki@9
jdk.crypto.ec@9
jdk.deploy@9
jdk.deploy.controlpanel@9
jdk.dynalink@9
jdk.editpad@9
jdk.hotspot.agent@9
jdk.httpserver@9
jdk.incubator.httpclient@9
jdk.internal.ed@9
jdk.internal.jvmstat@9
jdk.internal.le@9
jdk.internal.opt@9
jdk.internal.vm.ci@9
jdk.internal.vm.compiler@9
jdk.jartool@9
```

The earlier command will show all the exports in packages from the base module. Java base is the core of the system.

This will show all the graphical user interface packages. This will also show `requires` which are the dependencies:

```
$java --list-modules java.desktop
```



```
mayur@mayur-VirtualBox:~$ java --list-modules java.desktop
java.activation@9
java.base@9
java.compiler@9
java.corba@9
java.datatransfer@9
java.desktop@9
java.instrument@9
java.jnlp@9
java.logging@9
java.management@9
java.management.rmi@9
java.naming@9
java.prefs@9
java.rmi@9
java.scripting@9
java.se@9
java.se.ee@9
java.security.jgss@9
java.security.sasl@9
java.smartcardio@9
java.sql@9
java.sql.rowset@9
java.transaction@9
java.xml@9
java.xml.bind@9
java.xml.crypto@9
java.xml.ws@9
java.xml.ws.annotation@9
javafx.base@9
javafx.controls@9
javafx.deploy@9
javafx.fxml@9
javafx.graphics@9
javafx.media@9
javafx.swing@9
javafx.web@9
jdk.accessibility@9
jdk.aot@9
jdk.attach@9
jdk.charsets@9
jdk.compiler@9
jdk.crypto.cryptoki@9
jdk.crypto.ec@9
jdk.deploy@9
jdk.deploy.controlpanel@9
jdk.dynalink@9
jdk.editpad@9
jdk.hotspot.agent@9
jdk.httpserver@9
jdk.incubator.httpclient@9
jdk.internal.ed@9
jdk.internal.jvmstat@9
jdk.internal.le@9
jdk.internal.opt@9
jdk.internal.vm.ci@9
jdk.internal.vm.compiler@9
jdk.jartool@9
```

So far so good, right? Now you may be wondering, I got my modules developed but how to integrate them together? Let's look into that. Java 9's modular system comes with a tool called **JLink**. I know you can guess what I am going to say now. You are right, it links a set of modules and creates a runtime image. Now imagine the possibilities it can offer. You can create your own executable system with your own custom modules. Life is going to be a lot more fun for you, I hope! Oh, and on the other hand, you will be able to control the execution and remove unnecessary dependencies.

Let's see how to link modules together. Well, it's very simple. Just run the following command:

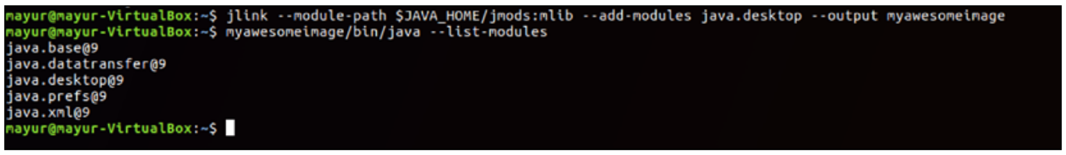
```
$jlink --module-path $JAVA_HOME/jmods:mlib --add-modules java.desktop
--output myawesomeimage
```

This linker command will link all the modules for you and create a runtime image. You need to provide a module path and then add the module that you want to generate a figure and give a name. Isn't it simple?

Now, let's check whether the previous command worked properly or not. Let's verify the modules from the figure:

```
$myawesomeimage/bin/java --list-modules
```

The output looks like this:



```
mayur@mayur-VirtualBox:~$ jlink --module-path $JAVA_HOME/jmods:mlib --add-modules java.desktop --output myawesomeimage
mayur@mayur-VirtualBox:~$ myawesomeimage/bin/java --list-modules
java.base@9
java.datatransfer@9
java.desktop@9
java.prefs@9
java.xml@9
mayur@mayur-VirtualBox:~$
```

With this, you will now be able to distribute a quick runtime with your application. It is awesome, isn't it? Now you can see how we moved from a somewhat monolithic design to a self-contained cohesive one. Each module contains its own exports and dependencies and JLink allows you to create your own runtime. With this, we got our modular platform.

Note that the aim of this section is to just introduce you to the modular system. There is a lot more to explore but that is beyond the scope of this book. In this book, we will focus on the performance enhancement areas.

Quick Introduction to Modules

I am sure that after reading about the modular platform, you must be excited to dive deep into the module architecture and see how to develop one. Hold your excitement please, I will soon take you on a journey to the exciting world of modules.

As you must have guessed, every module has a property `name` and is organized by packages. Each module acts as a self-contained unit and may have native code, configurations, commands, resources, and so on. A module's details are stored in a file named `module-info.java`, which resides in the root directory of the module source code. In that file, a module can be defined as follows:

```
module <name>{
}
```

In order to understand it better, let's go through an example. Let's say, our module name is `PerformanceMonitor`. The purpose of this module is to monitor the application performance. The input connectors will accept method names and the required parameters for that method. This method will be called from our module to monitor the module's performance. The output connectors will provide performance feedback for the given module. Let's create a `module-info.java` file in the root directory of our performance application and insert the following section:

```
module com.java9highperformance.PerformanceMonitor{
}
```

Awesome! You got your first module declaration. But wait a minute, it does not do anything yet. Don't worry, we have just created a skeleton for this. Let's put some flesh on the skeleton. Let's assume that our module needs to communicate with our other (magnificent) modules, which we have already created and named-- `PerformanceBase`, `StringMonitor`, `PrimitiveMonitor`, `GenericsMonitor`, and so on. In other words, our module has an external dependency. You may be wondering, how would we define this relationship in our module declaration? Ok, be patient, this is what we will see now:

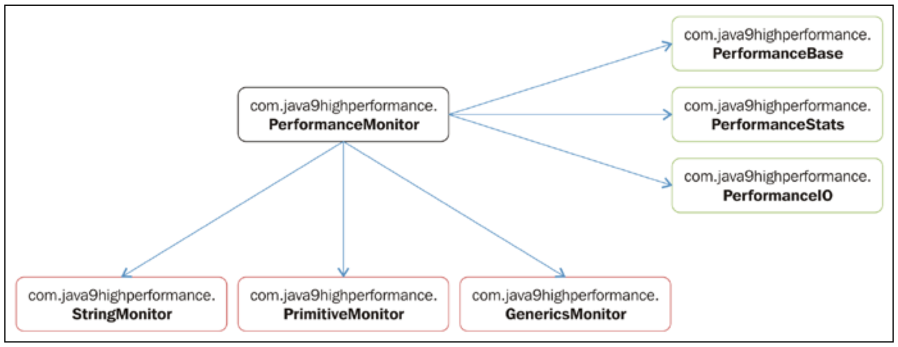
```
module com.java9highperformance.PerformanceMonitor{
    exports com.java9highperformance.StringMonitor;
    exports com.java9highperformance.PrimitiveMonitor;
    exports com.java9highperformance.GenericsMonitor;
    requires com.java9highperformance.PerformanceBase;
    requires com.java9highperformance.PerformanceStat;
    requires com.java9highperformance.PerformanceIO;
}
```

Yes, I know you have spotted two clauses, that is, `exports` and `requires`. And I am sure you are curious to know what they mean and why we have them there. We'll first talk about these clauses and what they mean when used in the module declaration:

- `exports`: This clause is used when your module has a dependency on another module. It denotes that this module exposes only public types to other modules and none of the internal packages are visible. In our case, the module `com.java9highperformance.PerformanceMonitor` has a dependency on `com.java9highperformance.StringMonitor`, `com.java9highperformance.PrimitiveMonitor`, and `com.java9highperformance.GenericsMonitor`. These modules export their API packages `com.java9highperformance.StringMonitor`, `com.java9highperformance.PrimitiveMonitor`, and `com.java9highperformance.GenericsMonitor`, respectively.
- `requires`: This clause denotes that the module depends upon the declared module at both compile and runtime. In our case, `com.java9highperformance.PerformanceBase`, `com.java9highperformance.PerformanceStat`, and `com.java9highperformance.PerformanceIO` modules are required by our `com.java9highperformance.PerformanceMonitor` module. The module system then locates all the observable modules to resolve all the dependencies recursively. This transitive closure gives us a module graph which shows a directed edge between two dependent modules.

Note: Every module is dependent on `java.base` even without explicitly declaring it. As you already know, everything in Java is an object.

Now you know about the modules and their dependencies. So, let's draw a module representation to understand it better. The following figure shows the various packages that are dependent on `com.java9highperformance.PerformanceMonitor`.


com.java9highperformance.
PerformanceMonitor
com.java9highperformance.
PerformanceBase
com.java9highperformance.
PerformanceStats
com.java9highperformance.
PerformanceIO
com.java9highperformance.
StringMonitor
com.java9highperformance.
PrimitiveMonitor
com.java9highperformance.
GenericsMonitor

Modules at the bottom are `exports` modules and modules on the right are `requires` modules.

Now let's explore a concept called **readability relationship**. Readability relationship is a relationship between two modules where one module is dependent on another module. This readability relationship is a basis for reliable configuration. So in our example, we can say `com.java9highperformance.PerformanceMonitor` reads `com.java9highperformance.PerformanceStat`.

Let's look at `com.java9highperformance.PerformanceStat` module's description file `module-info.java`:

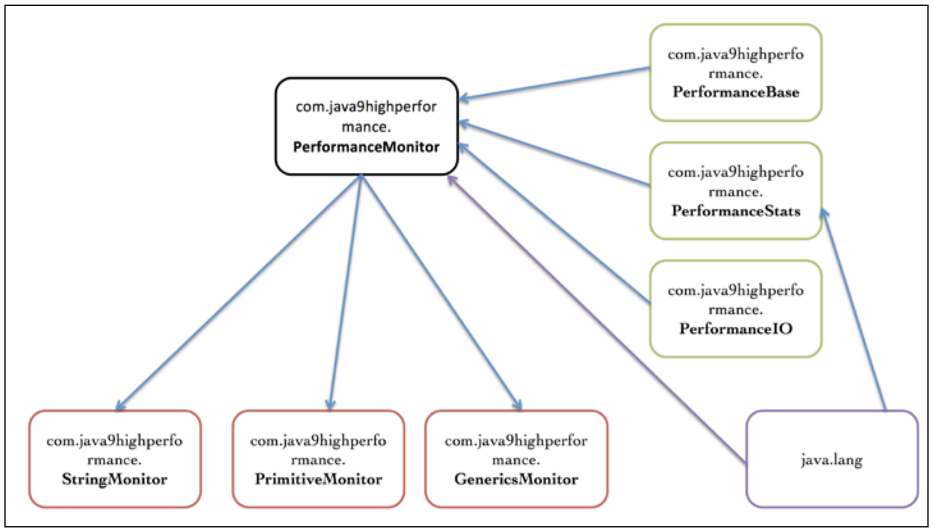
```
module com.java9highperformance.PerformanceStat{
    requires transitive java.lang;
}
```

This module depends on the `java.lang module`. Let's look at the `PerformanceStat` module in detail:

```
package com.java9highperformance.PerformanceStat;
import java.lang.*;

public Class StringProcessor{
    public String processString(){...}
}
```

In this case, `com.java9highperformance.PerformanceMonitor` only depends on `com.java9highperformance.PerformanceStat` but `com.java9highperformance.PerformanceStat` depends on `java.lang`. The `com.java9highperformance.PerformanceMonitor` module is not aware of the `java.lang` dependency from the `com.java9highperformance.PerformanceStat` module. This type of problem is taken care of by the module system. It has added a new modifier called **transitive**. If you look at `com.java9highperformance.PerformanceStat`, you will find it requires transitive `java.lang`. This means that any one depending on `com.java9highperformance.PerformanceStat` reads on `java.lang`.

See the following graph which shows the readability graph:


com.java9highperformance.
PerformanceMonitor
com.java9highperformance.
PerformanceBase
com.java9highperformance.
PerformanceStats
com.java9highperformance.
PerformanceIO
com.java9highperformance.
StringMonitor
com.java9highperformance.
PrimitiveMonitor
com.java9highperformance.
GenericsMonitor
java.lang

Now, in order to compile the `com.java9highperformance.PerformanceMonitor` module, the system must be able to resolve all the dependencies. These dependencies can be found from the module path. That's obvious, isn't that? However, don't misunderstand the classpath with the module path. It is a completely different breed. It doesn't have the issues that the packages have.

String Operations Performance

If you are not new to programming, string must be your best friend so far. In many cases, you may like it more than your spouse or partner. As we all know, you can't live without string, in fact, you can't even complete your application without a single use of string. OK, enough has been expressed about string and I am already feeling dizzy by the string usage just like JVM in the earlier versions. Jokes apart, let's talk about what has changed in Java 9 that will help your application perform better. Although this is an internal change, as an application developer, it is important to understand the concept so you know where to focus for performance improvements.

Java 9 has taken a step toward improving string performance. If you have ever come across JDK 6's failed attempt `UseCompressedStrings`, then you must be looking for ways to improve string performance. Since `UseCompressedStrings` was an experimental feature that was error prone and not designed very well, it was removed in JDK 7. Don't feel bad about it, I know it's terrible but as always the golden days eventually come. The JEP team has gone through immense pain to add a compact string feature that will reduce the footprint of string and its related classes.

Compact strings will improve the footprint of string and help in using memory space efficiently. It also preserves compatibility for all related Java and native interfaces. The second important feature is **Indify String Concatenation**, which will optimize a string at runtime.

In this section, we will take a closure look at these two features and their impact on overall application performance.

Compact String

Before we talk about this feature, it is important to understand why we even care about this. Let's dive deep into the underworld of JVM (or as any star wars fan would put it, the dark side of the Force). Let's first understand how JVM treats our beloved string and that will help us understand this new shiny compact string improvement. Let's enter into the magical world of heap. And as a matter of fact, no performance book is complete without a discussion of this mystical world.

The World of Heap

Each time JVM starts, it gets some memory from the underlining operating system. It is separated into two distinct regions called **heap space** and **Permgen**. These are home to all your application's resources. And as always with all good things in life, this home is limited in size. This size is set during the JVM initialization; however, you can increase or decrease this by specifying the JVM parameters, `-Xmx`, and `-XX:MaxPermSize`.

The heap size is divided into two areas, the nursery or young space and the old space. As the name suggests, the young space is home to new objects. This all sounds great but every house needs a cleanup. Hence, JVM has the most efficient cleaner called **garbage collector** (most efficient? Well... let's not get into that just yet). As any productive cleaner would do, the garbage collector efficiently collects all the unused objects and reclaims memory. When this young space gets filled up with new objects, the garbage collector takes charge and moves any of those who have lived long enough in the young space to the old space. This way, there is always room for more objects in the young space.

And in the same way, if the old space becomes filled up, the garbage collector reclaims the memory used.

Why Bother Compressing Strings?

Now you know a little bit about heap, let's look at the `String` class and how strings are represented on heap. If you dissect the heap of your application, you will notice that there are two objects, one is the Java language `String`object that references the second object `char[]` that actually handles the data. The `char` datatype is UTF-16 and hence takes up to 2 bytes. Let's look at the following example of how two different language strings look:

```
2 byte per char[]
Latin1 String : 1 byte per char[]
```

So you can see that `Latin1 String` only consumes 1 byte, and hence we are losing about 50% of the space here. There is an opportunity to represent it in a more dense form and improve the footprint, which will eventually help in speeding up garbage collection as well.

Now, before making any changes to this, it is important to understand its impact on real-life applications. It is essential to know whether applications use 1 byte per `char[]` strings or 2 bytes per `char[]` strings.

To get an answer to this, the JPM team analyzed a lot of heap dumps of real-world data. The result highlighted that a majority of heap dumps have around 18 percent to 30 percent of the entire heap consumed by `chars[]`, which come from string. Also, it was prominent that most strings were represented by a single byte per `char[]`. So, it is clear that if we try to improve the footprint for strings with a single byte, it will give significant performance boost to many real-life applications.

What Did They Do?

After having gone through a lot of different solutions, the JPM team has finally decided to come up with a strategy to compress string during its construction. First, optimistically try to compress in 1 byte and if it is not successful, copy it as 2 bytes. There are a few shortcuts possible, for example, the use of a special case encoder like ISO-8851-1, which will always spit 1 byte.

This implementation is a lot better than JDK 6's `UseCompressedStrings` implementation, which was only helpful to a handful of applications as it was compressing string by repacking and unpacking on every single instance. Hence the performance gain comes from the fact that it can now work on both the forms.

What is the Escape Route?

Even though it all sounds great, it may affect the performance of your application if it only uses 2 byte per `char[]` string. In that case, it make sense not to use the earlier mentioned, check, and directly store string as 2 bytes per `char[]`. Hence, the JPM team has provided a kill switch `--XX: -CompactStrings` using which you can disable this feature.

What is the Performance Gain?

The previous optimization affects the heap as we saw earlier that the string is represented in the heap. Hence, it is affecting the memory footprint of the application. In order to evaluate the performance, we really need to focus on the garbage collector. We will explore the garbage collection topic later, but for now, let's just focus on the run-time performance.

Indify String Concatenation

I am sure you must be thrilled by the concept of the compact string feature we just learned about. Now let's look at the most common usage of string, which is concatenation. Have you ever wondered what really happens when we try to concatenate two strings? Let's explore. Take the following example:

```
public static String getMyAwesomeString(){
    int javaVersion = 9;
    String myAwesomeString = "I love " + "Java " + javaVersion + "
high       performance book by Mayur Ramgir";
    return myAwesomeString;
}
```

In the preceding example, we are trying to concatenate a few strings with the `int` value. The compiler will then take your awesome strings, initialize a new `StringBuilder` instance, and then append all these individuals strings. Take a look at the following bytecode generation by `javac`. I have used the **ByteCode Outline** plugin for **Eclipse** to visualize the disassembled bytecode of this method. You may download it from `http://andrei.gmxhome.de/bytecode/index.html`:

```
// access flags 0x9
public static getMyAwesomeString()Ljava/lang/String;
  L0
  LINENUMBER 10 L0
  BIPUSH 9
  ISTORE 0
  L1
  LINENUMBER 11 L1
```

```
     NEW java/lang/StringBuilder
     DUP
     LDC "I love Java "
     INVOKESPECIAL java/lang/StringBuilder.<init> (Ljava/lang/String;)V
     ILOAD 0
     INVOKEVIRTUAL java/lang/StringBuilder.append (I)Ljava/lang/
   StringBuilder;
     LDC " high performance book by Mayur Ramgir"
     INVOKEVIRTUAL java/lang/StringBuilder.append (Ljava/lang/String;)
   Ljava/lang/StringBuilder;
     INVOKEVIRTUAL java/lang/StringBuilder.toString ()Ljava/lang/String;
     ASTORE 1
     L2
     LINENUMBER 12 L2
     ALOAD 1
     ARETURN
     L3
     LOCALVARIABLE javaVersion I L1 L3 0
     LOCALVARIABLE myAwesomeString Ljava/lang/String; L2 L3 1
     MAXSTACK = 3
     MAXLOCALS = 2
```

Quick Note: How do we interpret this?

- `INVOKESTATIC`: This is useful for invoking static methods
- `INVOKEVIRTUAL`: This uses of dynamic dispatch for invoking public and protected non-static methods
- `INVOKEINTERFACE`: This is very similar to `INVOKEVIRTUAL` except that the method dispatch is based on an interface type
- `INVOKESPECIAL`: This is useful for invoking constructors, methods of a superclass, and private methods

However, at runtime, due to the inclusion of `-XX:+-OptimizeStringConcat` into the JIT compiler, it can now identify the append of `StringBuilder` and the `toString` chains. In case the match is identified, produce low-level code for optimum processing. Compute all the arguments' length, figure out the final capacity, allocate the storage, copy the strings, and do the in place conversion of primitives. After this, handover this array to the `String` instance without copying. It is a profitable optimization.

But this also has a few drawbacks in terms of concatenation. One example is that in case of a concatenating string with long or double, it will not optimize properly. This is because the compiler has to do `.getChar` first which adds overhead.

Also, if you are appending `int` to `String`, then it works great; however, if you have an incremental operator like `i++`, then it breaks. The reason behind this is that you need to rewind to the beginning of the expression and re-execute, so you are essentially doing `++` twice. And now the most important change in Java 9 compact string. The length spell like `value.length >> coder`; `C2` cannot optimize it as it does not know about the IR.

Hence, to solve the problem of compiler optimization and runtime support, we need to control the bytecode, and we cannot expect `javac` to handle that.

We need to delay the decision of which concatenation can be done at runtime. So can we have just method `String.concat` which will do the magic. Well, don't rush into this yet as how would you design the method `concat`. Let's take a look. One way to go about this is to accept an array of the `String` instance:

```
public String concat(String... n){
    //do the concatenation
}
```

However, this approach will not work with primitives as you now need to convert each primitive to the `String`instance and also, as we saw earlier, the problem is that long and double string concatenation will not allow us to optimize it. I know, I can sense the glow on your face like you got a brilliant idea to solve this painful problem. You are thinking about using the `Object` instance instead of the `String` instance, right? As you know the `Object`instance is catch all. Let's look at your brilliant idea:

```
public String concat(Object... n){
    //do the concatenation
}
```

First, if you are using the `Object` instance, then the compiler needs to do autoboxing. Additionally, you are passing in the `varargs` array, so it will not perform optimally. So, are we stuck here? Does it mean we cannot use the preeminent compact string feature with string concatenation? Let's think a bit more; maybe instead of using the method `runtime`, let `javac` handle the concatenation and just give us the optimized bytecode. That sounds like a good idea. Well, wait a minute, I know you are thinking the same thing. What if JDK 10 optimizes this further? Does that mean, when I upgrade to the new JDK, I have to recompile my code again and deploy it again? In some cases, its not a problem, in other cases, it is a big problem. So, we are back to square one.

We need something that can be handled at runtime. Ok, so that means we need something which will dynamically invoke the methods. Well, that rings a bell. If we go back in our time machine, at the dawn of the era of JDK 7 it gave us `invokedynamic`. I know you can see the solution, I can sense the sparkle in your eyes. Yes, you are right, `invokedynamic` can help us here. If you are not aware of `invokedynamic`, let's spend some time to understand it. For those who have already mastered the topic, you could skip it, but I would recommend you go through this again.

Invokedynamic

The `invokedynamic` feature is the most notable feature in the history of Java. Rather than having a limit to JVM bytecode, we now can define our own way for operations to work. So what is `invokedynamic`? In simple terms, it is the user-definable bytecode. This bytecode (instead of JVM) determines the execution and optimization strategies. It offers various method pointers and adapters which are in the form of method handling APIs. The JVM then work on the pointers given in the bytecode and use reflection-like method pointers to optimize it. This way, you, as a developer, can get full control over the execution and optimization of code.

It is essentially a mix of user-defined bytecode (which is known as **bytecode + bootstrap**) and method handles. I know you are also wondering about the method handles--what are they and how to use them? Ok, I heard you, let's talk about method handles.

Method handles provide various pointers, including field, array, and method, to pass data and get results back. With this, you can do argument manipulation and flow control. From JVM's point of view, these are native instructions that it can optimize as if it were bytecode. However, you have the option to programmatically generate this bytecode.

Let's zoom in to the method handles and see how it all ties up together. The main package's name is `java.lang.invoke`, which has `MethodHandle`, `MethodType`, and `MethodHandles`. `MethodHandle` is the pointer that will be used to invoke the function. `MethodType` is a representation of a set of arguments and return value coming from the method. The utility class `MethodHandles` will act as a pointer to a method which will get an instance of `MethodHandle` and map the arguments.

We won't be going in deep for this section, as the aim was just to make you aware of what the `invokedynamic` feature is and how it works so you will understand the string concatenation solution. So, this is where we get back to our discussion on string concatenation. I know, you were enjoying the `invokedynamic` discussion, but I guess I was able to give you just enough insight to make you understand the core idea of Indify String Concatenation.

Let's get back on the concatenation part where we were looking for a solution to concatenate our awesome compact strings. For concatenating the compact strings, we need to take care of types and the number of types of methods and this is what the `invokedynamic` gives us.

So let's use `invokedynamic` for `concat`. Well, not so quick, my friend. There is a fundamental problem with this approach. We cannot just use `invokedynamic` as it is to solve this problem. Why? Because there is a circular reference. The `concat` function needs `java.lang.invoke`, which uses `concat`. This continues, and eventually you will get `StackOverflowError`.

Take a look at the following code:

```
String concat(int i, long l, String s){
    return s + i + l
}
```

So if we were to use `invokedynamic` here, the `invokedynamic` call would look like this:

```
InvokeDynamic #0: makeConcat(String, int, long)
```

There is a need to break the circular reference. However, in the current JDK implementation, you cannot control what `java.invoke` calls from the complete JDK library. Also, removing the complete JDK library reference from `java.invoke` has severe side effects. We only need the `java.base` module for Indify String Concatenation, and if we can figure out a way to just call the `java.base` module, then it will significantly improve the performance and avoid unpleasant exceptions. I know what you are thinking. We just studied the coolest addition to Java 9, **Project Jigsaw**. It provides modular source code and now we can only accept the `java.base` module. This solves the biggest problem we were facing in terms of concatenating two strings, primitives, and so on.

After going through a couple of different strategies, the Java Performance Management team has settled on the following strategy:

1. Make a call to the `toString()` method on all reference args.
2. Make a call to the `tolength()` method or since all the underlying methods are exposed, just call `T.stringSize(T t)` on every args.
3. Figure out the coders and call `coder()` for all reference args.
4. Allocate `byte[]` storage and then copy all args. And then, convert primitives in-place.
5. Invoke a private constructor `String` by handing over the array for concatenation.

With this, we are able to get an optimized string concat in the same code and not in `C2 IR`. This strategy gives us 2.9x better performance and 6.4x less garbage.

Storing Interned Strings in CDS Archives

The main goal of this feature is to reduce memory footprint caused by creating new instances of string in every JVM process. All the classes that are loaded in any JVM process can be shared with other JVM processes via **Class Data Sharing** (**CDS**) archives.

Oh, I did not tell you about CDS. I think it's important to spend some time to understand what CDS is, so you can understand the underlying performance improvement.

Many times, small applications in particular spend a comparatively long time on startup operations. To reduce this startup time, a concept called CDS was introduced. CDS enables sharing of a set of classes loaded from the system JAR file into a private internal representation during the JRE installation. This helps a lot as then any further JVM invocations can take advantage of these loaded classes' representation from the shared archive instead of loading these classes again. The metadata related to these classes is shared among multiple JVM processes.

CDS stores strings in the form of UTF-8 in the constant pool. When a class from these loaded classes begins the initialization process, these UTF-8 strings are converted into `String` objects on demand. In this structure, every character in every confined string takes 2 bytes in the `String` object and 1 byte to 3 bytes in the UTF-8, which essentially wastes memory. Since these strings are created dynamically, different JVM processes cannot share these strings.

Shared strings need a feature called **pinned regions** in order to make use of the garbage collector. Since the only HotSpot garbage collector that supports pinning is G1; it only works with the G1 garbage collector.

Concurrency Performance

Multithreading is a very popular concept. It allows programs to run multiple tasks at the same time. These multithreaded programs may have more than one unit which can run concurrently. Every unit can handle a different task keeping the use of available resources optimal. This can be managed by multiple threads that can run in parallel.

Java 9 improved contended locking. You may be wondering what is contended locking. Let's explore. Each object has one monitor that can be owned by one thread at a time. Monitors are the basic building blocks of concurrency. In order for a thread to execute a block of code marked as synchronized on an object or a synchronized method declared by an object, it must own this object's monitor. Since there are multiple threads trying to get access to the mentioned monitor, JVM needs to orchestrate the process and only allow one thread at a time. It means the rest of threads go in a wait state. This monitor is then called contended. Because of this provision, the program wastes time in the waiting state.

Also, **Java Virtual Machine** (**JVM**) does some work orchestrating the lock contention. Additionally, it has to manage threads, so once the existing thread finishes its execution, it can allow a new thread to go in. This certainly adds overhead and affects performance adversely. Java 9 has taken a few steps to improve in this area. The provision refines the JVM's orchestration, which will ultimately result in performance improvement in highly contested code.

The following benchmarks and tests can be used to check the performance improvements of contented Java object monitors:

- `CallTimerGrid` (This is more of a stress test than a benchmark)
- `Dacapo-bach` (earlier dacapo2009)
- `_ avrora`
- `_ batik`
- `_ fop`
- `_ h2`
- `_ luindex`
- `_ lusearch`
- `_ pmd`
- `_ sunflow`
- `_ tomcat`
- `_ tradebeans`
- `_ tradesoap`
- `_ xalan`
- `DerbyContentionModelCounted`
- `HighContentionSimulator`
- `LockLoops-JSR166-Doug-Sept2009` (earlier LockLoops)
- `PointBase`
- `SPECjbb2013-critical` (earlier specjbb2005)

- `SPECjbb2013-max`
- `specjvm2008`
- `volano29` (earlier volano2509)

Compiler Improvements

Several efforts have been made to improve the compiler's performance. In this section, we will focus on the improvements to the compiler side.

Tiered Attribution

The first and foremost change providing compiler improvement is related to **Tiered Attribution** (**TA**). This change is more related to lambda expressions. At the moment, the type checking of poly expression is done by type checking the same tree multiple times against different targets. This process is called **Speculative Attribution** (**SA**), which enables the use of different overload resolution targets to check a lambda expression.

This way of type checking, although a robust technique, adversely affects performance significantly. For example, with this approach, *n* number of overload candidates check against the same argument expression up to *n * 3* once per overload phase, strict, loose, and varargs. In addition to this, there is one final check phase. Where lambda returns a poly method call results in combinatorial explosion of attribution calls, this causes a huge performance problem. So we certainly need a different method of type checking for poly expressions.

The core idea is to make sure that a method call creates bottom-up structural types for each poly argument expression with every single details, which will be needed to execute the overload resolution applicability check before performing the overload resolution.

So in summary, the performance improvement was able to achieve an attribute of a given expression by decreasing the total number of tries.

Ahead-of-Time Compilation

The second noticeable change for compiler improvement is Ahead-of-Time compilation. If you are not familiar with the term, let's see what AOT is. As you probably know, every program in any language needs a runtime environment to execute. Java also has its own runtime which is known as **Java Virtual Machine** (**JVM**). The typical runtime that most of us use is a bytecode interpreter, which is JIT compiler as well. This runtime is known as **HotSpot JVM**.

This HotSpot JVM is famous for improving performance by JIT compilation as well as adaptive optimization. So far so good. However, this does not work well in practice for every single application. What if you have a very light program, say, a single method call? In this case, JIT compilation will not help you much. You need something that will load up faster. This is where AOT will help you. With AOT as opposed to JIT, instead of compiling to bytecode, you can compile into native machine code. The runtime then uses this native machine code to manage calls for new objects into mallocs as well as file access into system calls. This can improve performance.

Security Manager Improvements

Ok, let's talk about security. If you are not one of those who cares about application security over pushing more features in a release, then the expression on your face may be like **Uh! What's that?** If you are one those, then let's first understand the importance of security and find a way to consider this in your application development tasks. In today's SaaS-dominated world, everything is exposed to the outside world. A determined individual (a nice way of saying, a **malicious hacker**), can get access to your application and exploit the security holes you may have introduced through your negligence. I would love to talk about application security in depth as this is another area I am very much interested in. However, application security is out of the scope of this book. The reason we are talking about it here is that the JPM team has taken an initiative to improve the existing security manager. Hence, it is important to first understand the importance of security before talking about the security manager.

Hopefully, this one line of description may have generated secure programming interest in you. However, I do understand that sometimes you may not have enough time to implement a complete secure programming model due to tight schedules. So, let's find a way which can fit with your tight schedule. Let's think for a minute; is there any way to automate security? Can we have a way to create a blueprint and ask our program to stay within the boundaries? Well, you are in luck, Java does have a feature called **security manager**. It is nothing but a policy manager that defines a security policy for the application. It sounds exciting, doesn't it? But what does this policy look like? And what does it contain? Both are fair questions to ask. This security policy basically states actions that are dangerous or sensitive in nature. If your application does not comply with this policy, then the security manager throws `SecurityException`. On the other side, you can have your application call this security manager to learn about the permitted actions. Now, let's look at the security manager in detail.

In case of a web applet, a security manager is provided by the browser, or the Java Web Start plugin runs this policy. In many cases, applications other than web applets run without a security manager unless those applications implement one. It's a no brainer to say that if there is no security manager and no security policy attached, the application acts without restrictions.

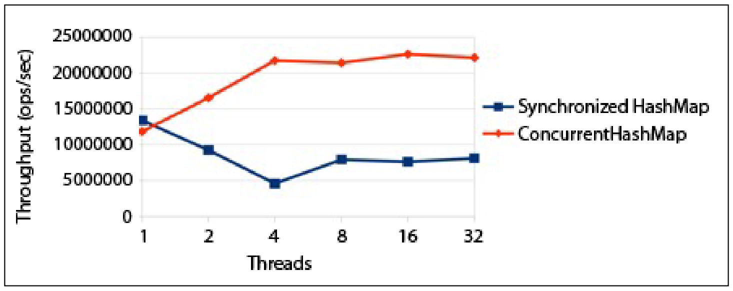
Now we know a little about the security manager, let's look at the performance improvement in this area. As per the Java team, there may be a possibility that an application running with a security manager installed degrades performance by 10 percent to 15 percent. However, it is not possible to remove all the performance bottlenecks but narrowing this gap can assist in improving not only security but also performance.

The Java 9 team looked at some of the optimizations, including the enforcement of security policy and the evaluation of permissions, which will help improve the overall performance of using a security manager. During the performance testing phase, it was highlighted that even though the permission classes are thread safe, they show up as a HotSpot. Numerous improvements have been made to decrease thread contention and improve throughput.

Computing the `hashcode` method of `java.security.CodeSource` has been improved to use a string form of the code source URL to avoid potentially expensive DNS lookups. Also, the `checkPackageAccess` method of `java.lang.SecurityManager`, which contains the package checking algorithm, has been improved.

Some other noticeable changes in security manager improvements are as follows:

- The first noticeable change is that using `ConcurrentHashMap` in place of `Collections.synchronizedMap` helps improving throughput of the `Policy.implie` method. Look at the following graph, taken from the OpenJDK site, which highlights the significant increase in the throughput with `ConcurrentHashMap`:



- In addition to this, `HashMap`, which had been used for maintaining internal collection of `CodeSource` in `java.security.SecureClassLoader`, has been replaced by `ConcurrentHashMap`.
- There are a few other small improvements like an improvement in the throughput by removing the compatibility code from the `getPermissions` method (`CodeSource`), which synchronizes on identities.
- Another significant gain in performance is achieved using `ConcurrentHashMap` instead of `HashMap` surrounded by synchronized blocks in the permission checking code, which yielded in greater thread performance.

Graphics Rasterizers

If you are into Java 2D and using OpenJDK, you will appreciate the efforts taken by the Java 9 team. Java 9 is mainly related to a graphics rasterizer, which is part of the current JDK. OpenJDK uses Pisces, whereas Oracle JDK uses Ductus. Oracle's closed-source Ductus rasterizer performs better than OpenJDK's Pisces.

These graphics rasterizers are useful for anti-aliased rendering except fonts. Hence, for a graphics-intensive application, the performance of this rasterizer is very important. However, Pisces is failing in many fronts and its performance problems are very visible. Hence, the team has decided to replace this with a different rasterizer called Marlin Graphics Renderer.

Marlin is developed in Java and, most importantly, it is the fork of the Pisces rasterizer. Various tests have been done on it and the results are very promising. It consistently performs better than Pisces. It demonstrates multithreaded scalability and even outperforms the closed-source Ductus rasterizer for a single-threaded application.

Summary

In this lesson, we have seen some of the exciting features that will improve your application's performance without making any effort from your end.

In the next lesson, we will learn about JShell and the **Ahead-of-Time** (**AOT**) compiler. We will also learn about **Read-Eval-Print Loop** (**REPL**) tool.

Assessments

1. JLink is a ___________ of Java 9 modular system.
2. What is the relationship between two modules where one module is dependent on another module?
    1. Readability relationship
    2. Operability relationship
    3. Modular relationship
    4. Entity relationship
3. State whether True or False: Each time JVM starts, it gets some memory from the underlining operating system.
4. Which of the following perform some work orchestrating the lock contention?
    1. Pinned regions
    2. Readability relationship
    3. Java Virtual Machine
    4. Class data sharing
5. Which of the following enables the use of different overload resolution targets to check a lambda expression?
    1. Tiered attribution
    2. HotSpot JVM
    3. Speculative attribution
    4. Permgen

2
Tools for Higher Productivity and Faster Application

Since the dawn of programming as a profession, the standing goals of every aspiring coder were to quickly produce applications that perform the assigned tasks with lightning speed. Otherwise, why bother? We could slowly do whatever we were doing for thousands of years. In the book of the last century, we made substantial progress in both aspects, and now, Java 9 makes another step in each of these directions.

Two new tools were introduced in Java 9, JShell and the **Ahead-of-Time** (**AOT**) compiler--both were expected for a long time. JShell is a **Read–Eval–Print Loop** (**REPL**) tool that is well-known for those who program in Scala, Ruby, or Python, for example. It takes a user input, evaluates it, and returns the result immediately. The AOT compiler takes Java bytecode and generates a native (system-dependent) machine code so that the resulting binary file can execute natively.

These tools will be the focus of this lesson.

The JShell Tool Usage

JShell helps a programmer to test fragments (snippets) of code as they are written. It shortens the time for development by avoiding the build-deploy-test part of the development cycle. Programmers can easily copy an expression or even several methods into the JShell session and run-test-modify them multiple times immediately. Such a quick turnaround also helps to understand the library API better before using it and to tune the code to express exactly its purpose, thus facilitating better quality software.

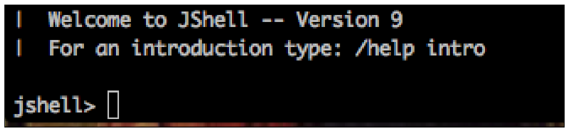
How often have we guessed what the JavaDoc for a particular API meant and wasted build-deploy-test cycles for figuring it out? Or we want to recall, how exactly the string will be split by `substring(3)`? Sometimes, we create a small test application where we run the code we are not sure about, using again the same build-deploy-test cycle. With JShell, we can copy, paste, and run. In this section, we will describe and show how to do it.

JShell is built on the top of JVM, so it processes the code snippets exactly as JVM does. Only a few constructs that do not make sense for REPL are omitted. For example, you cannot use `package` declaration, `static`, or `final` in JShell (these keywords are going to be ignored). Also, the semicolon `;` is allowed but not required at the end of a statement.

JShell comes with API included in the module `jdk.jshell` which can be used for the integration of JShell into other tools (IDE, for example), but it is outside of the scope of this book.

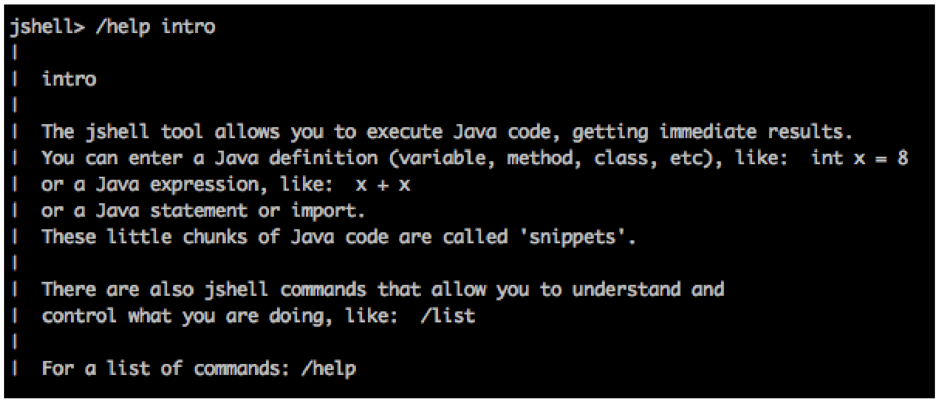
Creating a JShell Session and Setting Context

JShell comes with the JDK installation. You can find it in the `bin` directory as `$JAVA_HOME/bin/jshell`. Execute it to start the JShell session. Before you get familiar with JShell, we recommend starting the session with the option `-v`, which stands for **verbose**. This way, the shell will add more details to each of your actions, explaining what has been accomplished with each of them. After launching `jshell` in a terminal window, you will see the following output:



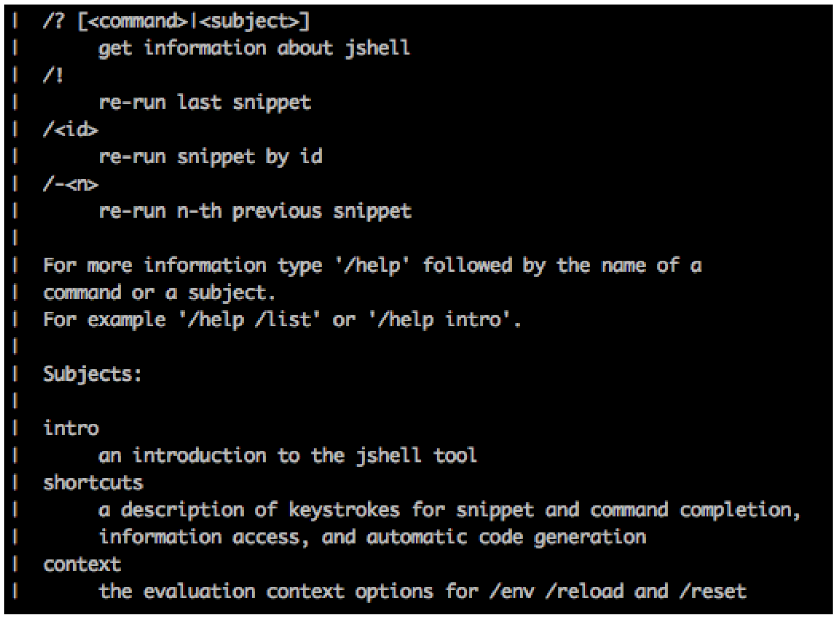
```
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> 
```

This means that a JShell session is created and can be used for Java code running. Enter the recommended command `/help intro` and read the following JShell introduction:



```
jshell> /help intro
|
|  intro
|
|  The jshell tool allows you to execute Java code, getting immediate results.
|  You can enter a Java definition (variable, method, class, etc), like:  int x = 8
|  or a Java expression, like:  x + x
|  or a Java statement or import.
|  These little chunks of Java code are called 'snippets'.
|
|  There are also jshell commands that allow you to understand and
|  control what you are doing, like:  /list
|
|  For a list of commands: /help
```

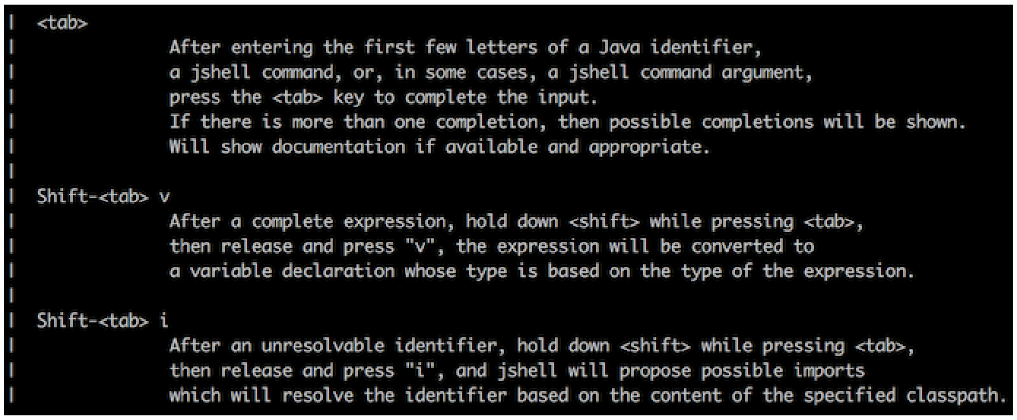
The introduction tells us the very minimum we need to know in order to get going. So, let's follow the guide. If we enter `/help`, we get the list of possible JShell commands with a short description (we will go over every command in more detail later) and the following information:



```
|  /? [<command>|<subject>]
|       get information about jshell
|  /!
|       re-run last snippet
|  /<id>
|       re-run snippet by id
|  /-<n>
|       re-run n-th previous snippet
|
|  For more information type '/help' followed by the name of a
|  command or a subject.
|  For example '/help /list' or '/help intro'.
|
|  Subjects:
|
|  intro
|       an introduction to the jshell tool
|  shortcuts
|       a description of keystrokes for snippet and command completion,
|       information access, and automatic code generation
|  context
|       the evaluation context options for /env /reload and /reset
```

Those are important tips to remember. Notice that the `/?` and `/help` commands produce the same result, so from now on, we will use`/?` only. The commands `/i`, `/<id>` (id is assigned to each snippet automatically and shown to the left of the snippet when listed by the command `/list`), and `/-<n>` allow re-running of the snippets that have been run previously.

Subject `intro` we saw already. Subject `shortcuts` can be viewed by entering the command `/? shortcuts`:

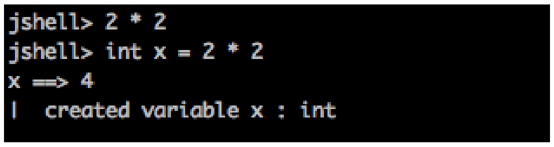


```
|  <tab>
|               After entering the first few letters of a Java identifier,
|               a jshell command, or, in some cases, a jshell command argument,
|               press the <tab> key to complete the input.
|               If there is more than one completion, then possible completions will be shown.
|               Will show documentation if available and appropriate.
|
|  Shift-<tab> v
|               After a complete expression, hold down <shift> while pressing <tab>,
|               then release and press "v", the expression will be converted to
|               a variable declaration whose type is based on the type of the expression.
|
|  Shift-<tab> i
|               After an unresolvable identifier, hold down <shift> while pressing <tab>,
|               then release and press "i", and jshell will propose possible imports
|               which will resolve the identifier based on the content of the specified classpath.
```

As you can see, the *Tab* key can be used to complete the current entry, while double *Tab* brings up possible completion options or JavaDoc, if available. Do not hesitate to press *Tab* several times after each command. It will help you to find more ways to utilize JShell features to your advantage.

Press *Shift* + *Tab* and then press *V* to create a variable based on the just completed expression. Here is an example:

- Type `2*2` on the console and press *Enter*.
- Press *Shift* + *Tab* together.
- Release the keys and press *V*.
- The shell will show `int x = 2*2` and position the cursor just in front of `=`.
- Enter the variable (`x`, for example, and press *Enter*). The resulting screen will show the following output:

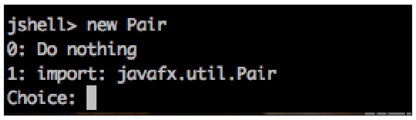


```
jshell> 2 * 2
jshell> int x = 2 * 2
x ==> 4
|  created variable x : int
```

Press *Shift* + *Tab* and then press *I* after an unresolved identifier requests JShell to provide possible imports based on the content of the classpath. Here is an example:

- Type `new Pair` and press *Enter*.
- Press *Shift* + *Tab* together.

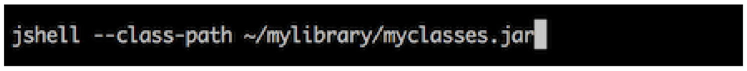
- Release the keys and press *I*. The shell will show the following output:



```
jshell> new Pair
0: Do nothing
1: import: javafx.util.Pair
Choice: 
```

- You will get two options with the values `0` and `1`, respectively.
- In the shell, you will get a statement called `Choice`; type *1* and press *Enter*.
- Now, the `javafx.util.Pair` class is imported.
- You can continue entering the code snippet.

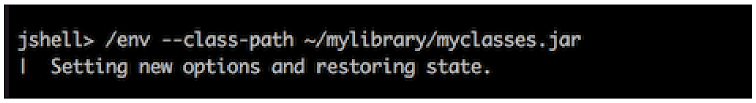
JShell was able to provide the suggestion because the JAR file with the compiled `Pair` class was on the classpath (set there by default as part of JDK libraries). You can also add to the classpath any other JAR file with the compiled classes you need for your coding. You can do it by setting it at JShell startup by the option `--class-path` (can be also used with one dash `-class-path`):



```
jshell --class-path ~/mylibrary/myclasses.jar
```

In the earlier example, the JAR file `myclasses.jar` is loaded from the folder `mylibrary` in the user's home directory. To set several JAR files, you can separate them by a colon `:` (for Linux and MacOS) or by a semicolon `;` (for Windows).

The classpath can also be set by the command `/env` any time during the JShell session:



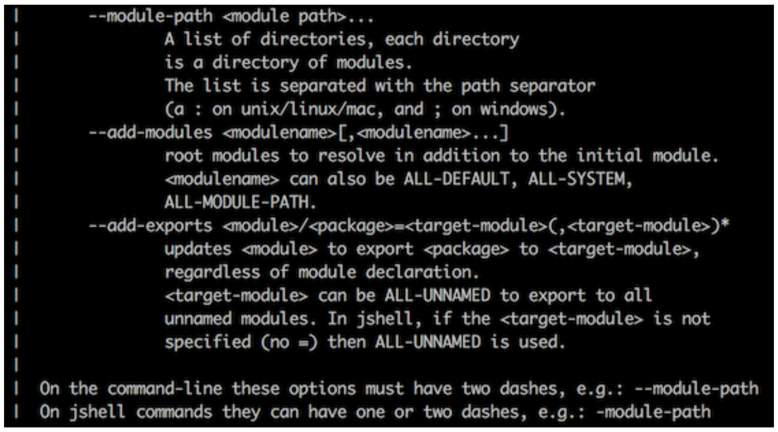
```
jshell> /env --class-path ~/mylibrary/myclasses.jar
|  Setting new options and restoring state.
```

Notice that every time the classpath is set, all the snippets of the current session are reloaded with the new classpath.

The commands `/reset` and `/reload` can be used instead of the `/env` command to set the classpath too. We will describe the difference between these commands in the next section.

If you do not want to collect your compiled classes in a JAR file, the option `--class-path` (or `-class-path`) could point to the directory where the compiled classes are located. Once the classpath is set, the classes associated with it can be imported during a snippet writing using keys *Shift* + *Tab* and then *I* as described earlier.

Other context options are related to the usage of modules and can be seen after entering the command `/? context`:



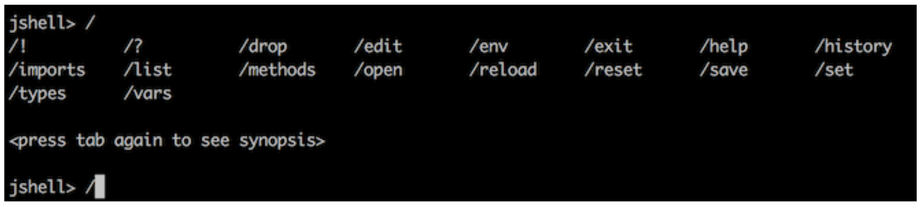
```
|       --module-path <module path>...
|               A list of directories, each directory
|               is a directory of modules.
|               The list is separated with the path separator
|               (a : on unix/linux/mac, and ; on windows).
|       --add-modules <modulename>[,<modulename>...]
|               root modules to resolve in addition to the initial module.
|               <modulename> can also be ALL-DEFAULT, ALL-SYSTEM,
|               ALL-MODULE-PATH.
|       --add-exports <module>/<package>=<target-module>(,<target-module>)*
|               updates <module> to export <package> to <target-module>,
|               regardless of module declaration.
|               <target-module> can be ALL-UNNAMED to export to all
|               unnamed modules. In jshell, if the <target-module> is not
|               specified (no =) then ALL-UNNAMED is used.
|
|   On the command-line these options must have two dashes, e.g.: --module-path
|   On jshell commands they can have one or two dashes, e.g.: -module-path
```

There are several more advanced options of running the `jshell` tool. To learn about them, refer to the Oracle documentation (for example, `https://docs.oracle.com/javase/9/tools/jshell.htm`).

The last important command we would like to mention in this section is `/exit`. It allows exiting the command mode and closing the JShell session.

JShell Commands

As we mentioned in the previous section, the full list of JShell commands can be obtained by typing the `/?` command. Each command comes with a one-line description. There is another way to get the same list but without description, that is by typing `/` followed by *Tab*. The screen would show the following content:



```
jshell> /
/!          /?          /drop       /edit       /env        /exit       /help       /history
/imports    /list       /methods    /open       /reload     /reset      /save       /set
/types      /vars

<press tab again to see synopsis>

jshell> /
```

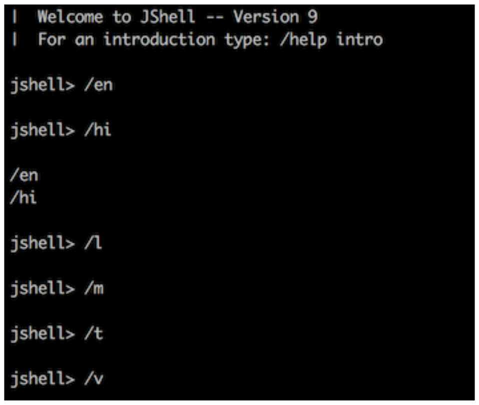
Pressing *Tab* the second time would bring the same list of the commands with a synopsis (one-line description) for each. To make it easier for a user, while typing, a command, subcommand, command argument, or command option can be abbreviated, as long as it remains unique so that the tool can recognize it unambiguously. For example, instead of the previous list of full-name commands, you can use the corresponding list of their abbreviated versions: `/!`, `/?`, `/d`, `/ed`, `/en`, `/ex`, `/he`, `/hi`, `/i`, `/l`, `/m`, `/o`, `/rel`, `/res`, `/sa`, `/se`, `/t`, `/v`. The preceding dash `/` is necessary for distinguishing commands from snippets.

Now, let's review each of these commands. While doing it, we will create a few snippets, variables, and types so that we can demonstrate each command more clearly using specific examples.

You can start a new JShell session by running `jshell` (with option `-v`) and enter the following commands:

- `/en`: To view or change the evaluation context
- `/h`: To view history of what you have typed
- `/l [<name or id>|-all|-start]`: To list the source you have typed
- `/m [<name or id>|-all|-start]L`: To list the declared methods and their signatures
- `/t [<name or id>|-all|-start]`: To list the declared types
- `/v [<name or id>|-all|-start]`: To list the declared variables and their values

The result would be like this:

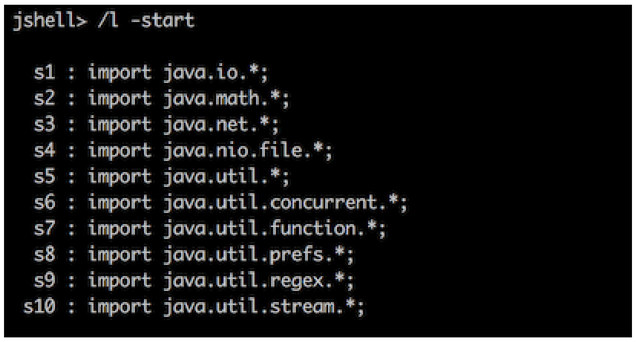


```
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> /en

jshell> /hi

/en
/hi

jshell> /l

jshell> /m

jshell> /t

jshell> /v
```

As you might be expecting, most of these commands yielded no results (except a short history of your typing until that moment) because we have not entered any code snippet yet. The last four commands have the same options:

- `<name or id>`: This is the name or ID of a specific snippet or method or type or variable (we will see examples later)
- `-start`: This shows snippets or methods or types or variables loaded at the JShell start (we will see later how to do it)
- `-all`: This shows snippets or methods or types or variables loaded at the JShell start and entered later during the session

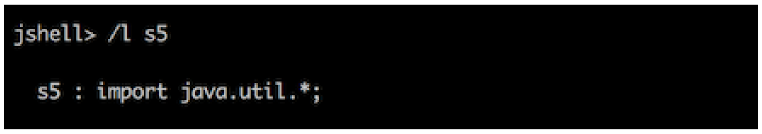
By default, at the startup, several common packages are imported. You can see them by typing the `/l -start` or `/l -all` command:



```
jshell> /l -start

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
```

There is no `java.lang` package in this list, but it is always imported by default and not listed among the imports.

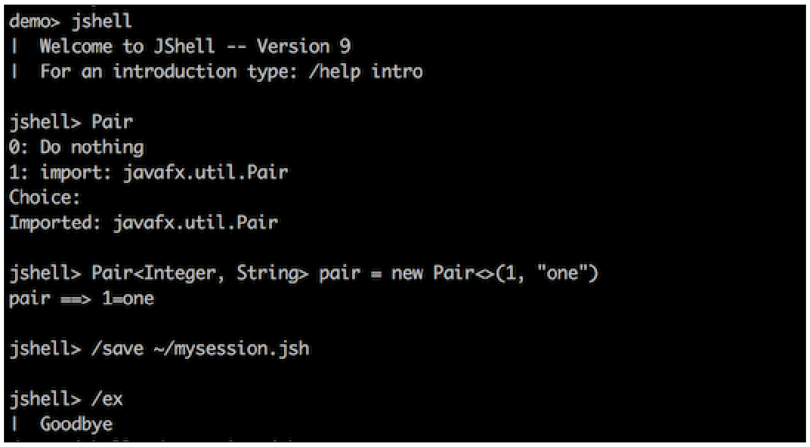
In the left column of the previously mentioned list, you can see the ID of each snippet. If you type the `/l s5`command, for example, it will retrieve the snippet with ID `s5`:



```
jshell> /l s5

  s5 : import java.util.*;
```

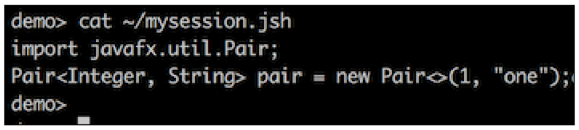
To customize the startup entries, you can use the command `/sa <file>` to save in the specified file all the settings and snippets you have entered in the current session. The next time you would like to continue with the same context, you can start the JShell session with this file `jshell <file>`.

Let's demonstrate this procedure with an example:



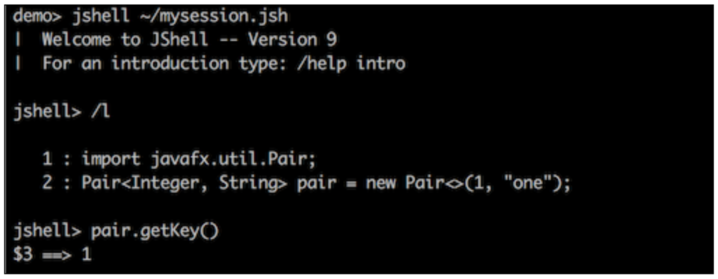
```
demo> jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> Pair
0: Do nothing
1: import: javafx.util.Pair
Choice:
Imported: javafx.util.Pair

jshell> Pair<Integer, String> pair = new Pair<>(1, "one")
pair ==> 1=one

jshell> /save ~/mysession.jsh

jshell> /ex
|  Goodbye
```

In the previous screenshot, you can see that we have started a JShell session and entered the name of the class `Pair` (not imported yet), then pressed *Shift* + *Tab* and *I* and selected option `1` (to import the class `Pair`). After that, we have finished typing the snippet (created a variable `pair`), saved the session entries in the file `mysession.jsh` (in the home directory), and closed the session. Let's look in the file `mysession.jsh` now:



```
demo> cat ~/mysession.jsh
import javafx.util.Pair;
Pair<Integer, String> pair = new Pair<>(1, "one");
demo>
```

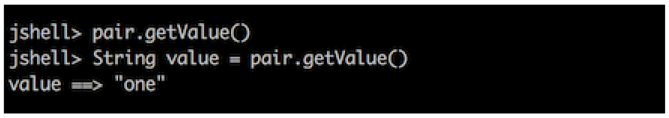
As you can see, the file contains only the new entries from the saved session. If we would like to load them into the next session, we will use the command `jshell ~/mysession.jsh` and continue working in the same context:



```
demo> jshell ~/mysession.jsh
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> /l

   1 : import javafx.util.Pair;
   2 : Pair<Integer, String> pair = new Pair<>(1, "one");

jshell> pair.getKey()
$3 ==> 1
```

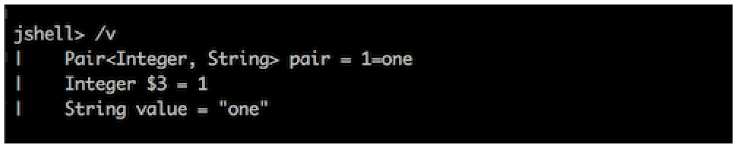
In the previous screenshot, we started a new session, listed all the new entries (reloaded from the previous session), and got a key from the object `pair`. This has created variable `$3` automatically.

We can also create a variable explicitly. Type `pair.getValue()` and press *Shift* + *Tab* and then press *V*, which will prompt you to enter the variable name just in front of the sign `String = pair.getValue()`. Enter `value` and see the result:



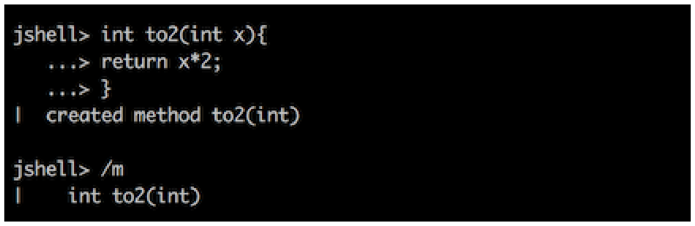
```
jshell> pair.getValue()
jshell> String value = pair.getValue()
value ==> "one"
```

To see all the variables of the current session, type the command `/v`:



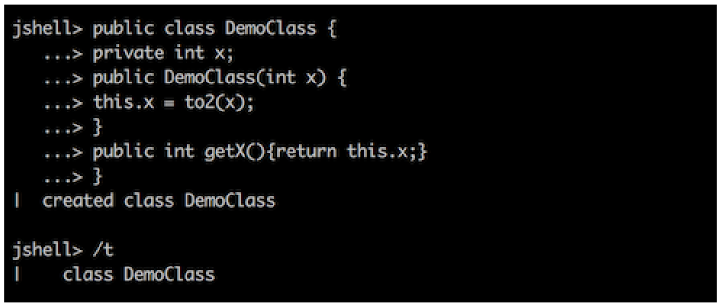
```
jshell> /v
|    Pair<Integer, String> pair = 1=one
|    Integer $3 = 1
|    String value = "one"
```

Let's now create a method `to2()` that multiplies any integer by 2:



```
jshell> int to2(int x){
   ...> return x*2;
   ...> }
|  created method to2(int)

jshell> /m
|    int to2(int)
```

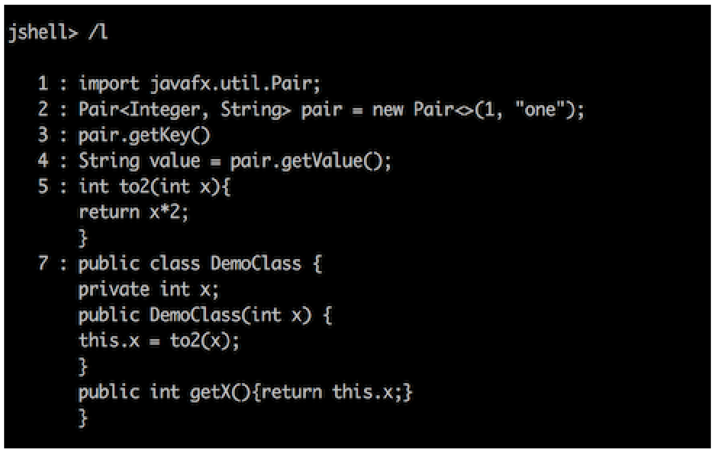
To complete the demonstration of the commands `/l`, `/m`, `/t`, and `/v`, let's create a new type:



```
jshell> public class DemoClass {
   ...> private int x;
   ...> public DemoClass(int x) {
   ...> this.x = to2(x);
   ...> }
   ...> public int getX(){return this.x;}
   ...> }
|  created class DemoClass

jshell> /t
|    class DemoClass
```

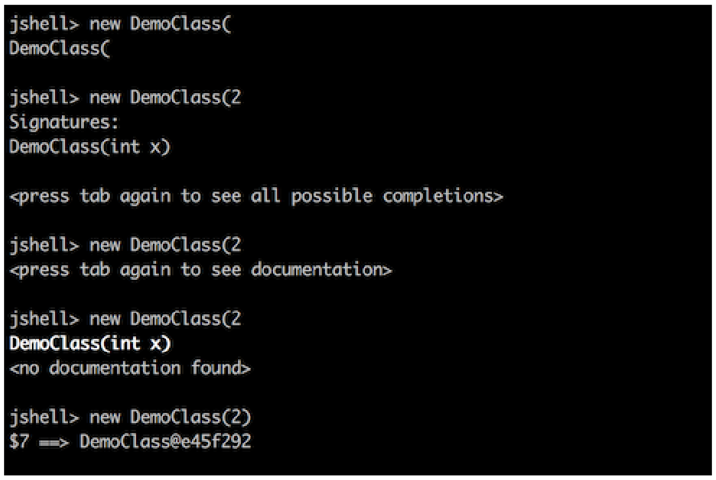
Notice that the method `to2()` is visible inside a new class, which means that all standalone variables, standalone methods, and code inside classes are executed in the same context. This way, testing of a code fragment becomes easier but may introduce subtle errors and even unexpected behavior if the code's author relies on the encapsulation and behavior isolation in different parts of a more complex system than just a flat code fragment.

Now, by using the `/l` command, we can see everything we have typed:



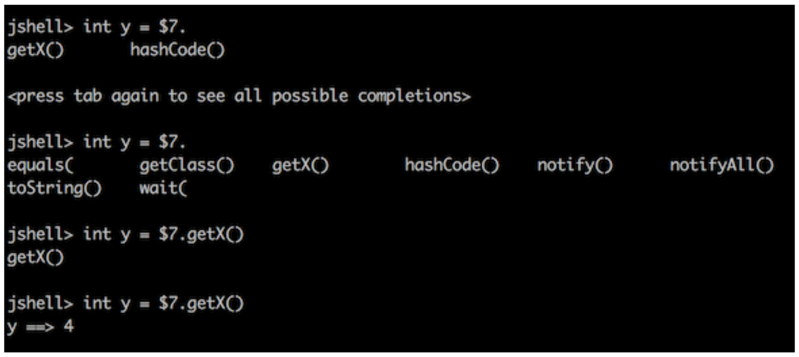
```
jshell> /l

   1 : import javafx.util.Pair;
   2 : Pair<Integer, String> pair = new Pair<>(1, "one");
   3 : pair.getKey()
   4 : String value = pair.getValue();
   5 : int to2(int x){
       return x*2;
       }
   7 : public class DemoClass {
       private int x;
       public DemoClass(int x) {
       this.x = to2(x);
       }
       public int getX(){return this.x;}
       }
```

All these snippets are available for execution. Here is one example of using them:



```
jshell> new DemoClass(
DemoClass(

jshell> new DemoClass(2
Signatures:
DemoClass(int x)

<press tab again to see all possible completions>

jshell> new DemoClass(2
<press tab again to see documentation>

jshell> new DemoClass(2
DemoClass(int x)
<no documentation found>

jshell> new DemoClass(2)
$7 ==> DemoClass@e45f292
```

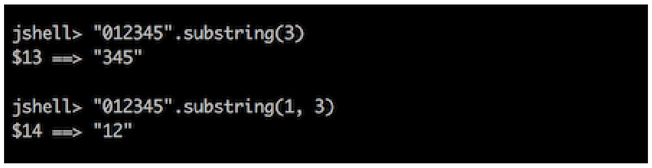
In the previous screenshot, we typed `new Demo` and pressed *Tab*. Then, we entered 2 and pressed *Tab* again. We saw the suggestion about pressing *Tab* to see documentation and did it. Well, there was no documentation found (we did not type any JavaDoc while creating the class `DemoClass`), so we just added `)` and pressed *Enter*. As a result, a new variable `$7` was created that held references to the object of the class `DemoClass`. We can use this variable now like this, for example:



```
jshell> int y = $7.
getX()      hashCode()

<press tab again to see all possible completions>

jshell> int y = $7.
equals(      getClass()    getX()       hashCode()    notify()      notifyAll()
toString()   wait(

jshell> int y = $7.getX()
getX()

jshell> int y = $7.getX()
y ==> 4
```

In the previous screenshot, we entered `int y = $7.` and pressed *Tab*, then pressed *Tab* the second time to see other options. We did it just for demo purposes. Then, we made our selection by typing `getX` after `.` and pressing `Tab`. JShell completed the statement with `()` for us and we pressed *Enter*, thus creating a new variable `y` (with the current evaluated value of `4`).

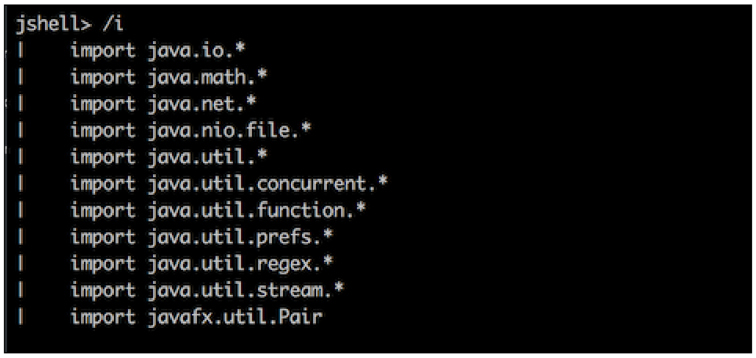
Finally, let's try and test the function `substring()` to make sure it returns us the substring we need:



```
jshell> "012345".substring(3)
$13 ==> "345"

jshell> "012345".substring(1, 3)
$14 ==> "12"
```

We hope you now have a feel of how you can create and execute snippets.

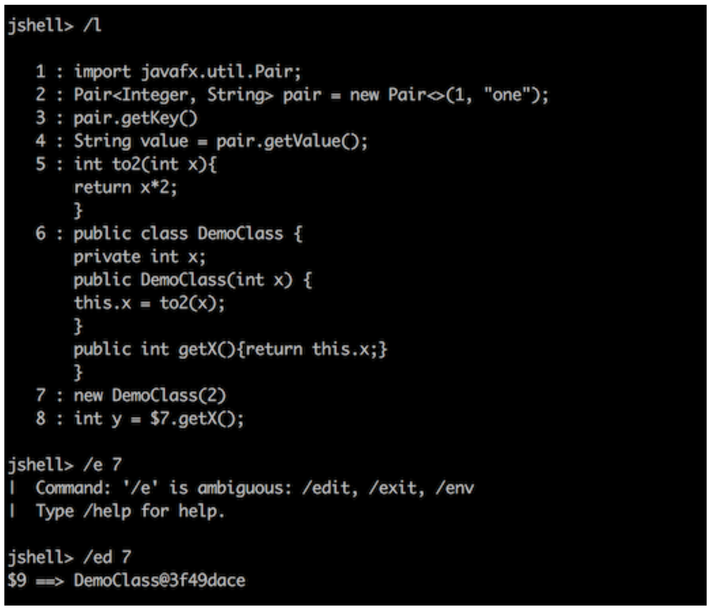
Let's review other JShell commands. The command `/i` lists the imported packages and classes. In our case, if we use this command we will get the following output:



```
jshell> /i
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
|    import javafx.util.Pair
```

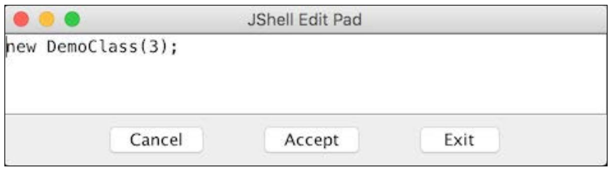
You can see that the class `Pair` is listed as imported, although we have done it in the previous JShell session and brought it in the new session by using the file `~/mysession.jsh`.

The command `/ed <name or id>` allows you to edit any of the entries listed by the command `/l`. Let's do it:



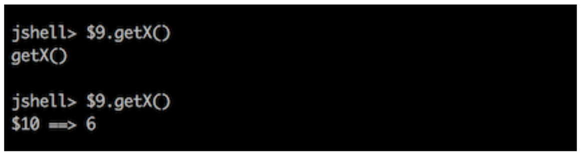
```
jshell> /l

   1 : import javafx.util.Pair;
   2 : Pair<Integer, String> pair = new Pair<>(1, "one");
   3 : pair.getKey()
   4 : String value = pair.getValue();
   5 : int to2(int x){
       return x*2;
       }
   6 : public class DemoClass {
       private int x;
       public DemoClass(int x) {
       this.x = to2(x);
       }
       public int getX(){return this.x;}
       }
   7 : new DemoClass(2)
   8 : int y = $7.getX();

jshell> /e 7
|  Command: '/e' is ambiguous: /edit, /exit, /env
|  Type /help for help.

jshell> /ed 7
$9 ==> DemoClass@3f49dace
```

In the previous screenshot, we listed all the snippets and entered `/e 7` to edit snippets with ID `7`. It turned out that there are several commands starting with `e`, so we added `d` and got the following editor window:



```
new DemoClass(3);
```

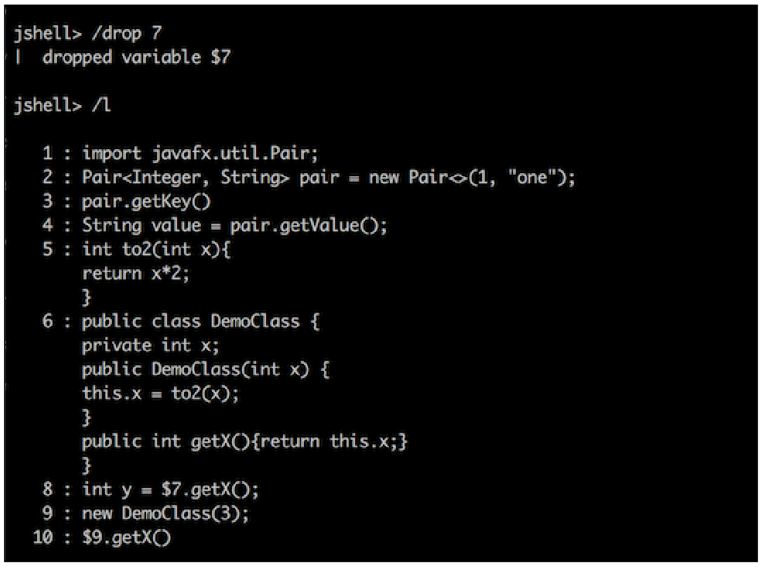
In the previous window, we changed `2` to `3` and clicked the **Accept** button. As a result, a new variable `$9`was created that holds the reference to the new `DemoClass` object. We can now use this new variable too:



```
jshell> $9.getX()
getX()

jshell> $9.getX()
$10 ==> 6
```

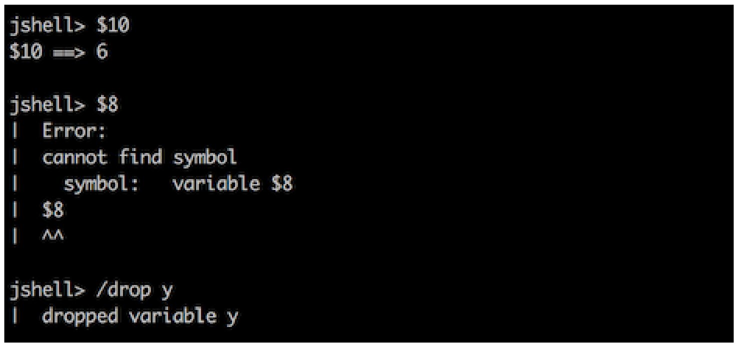
In the previous screenshot, we entered `$9.getX` and pressed *Tab*. The JShell completed the statement by adding `()`. We press *Enter*, and the new variable `$10` (with the current evaluated value `6`) was created.

The command `/d <name or id>` drops a snippet referenced by name or ID. Let's use it to delete a snippet with ID `7`:



```
jshell> /drop 7
|  dropped variable $7

jshell> /l

   1 : import javafx.util.Pair;
   2 : Pair<Integer, String> pair = new Pair<>(1, "one");
   3 : pair.getKey()
   4 : String value = pair.getValue();
   5 : int to2(int x){
       return x*2;
       }
   6 : public class DemoClass {
       private int x;
       public DemoClass(int x) {
       this.x = to2(x);
       }
       public int getX(){return this.x;}
       }
   8 : int y = $7.getX();
   9 : new DemoClass(3);
  10 : $9.getX()
```

As you could guess, the expression that assigns a value to the variable `8` now cannot be evaluated:



```
jshell> $10
$10 ==> 6

jshell> $8
|  Error:
|  cannot find symbol
|    symbol:   variable $8
|  $8
|  ^^

jshell> /drop y
|  dropped variable y
```

In the earlier screenshot, we first requested to evaluate the expression that generates a value for variable `10` (for demonstration purposes), and it was correctly calculated as `6`. Then, we attempted to do the same for variable `8` and received an error because its expression was broken after deleting the variable `7`. So, we have deleted it now, too (this time by name, to demonstrate how a name can be used).

The command `/sa [-all|-history|-start] <file>` saves a snippet to a file. It is complemented by the command `/o <file>` that opens the file as the source input.

The commands `/en`, `/res`, and `/rel` have an overlapping functionality:

- `/en [options]`: This allows to view or change the evaluation context
- `/res [options]`: This discards all entered snippets and restarts the session
- `/rel[options]`: This reloads the session the same way the command `/en` does

See the official Oracle documentation (`http://docs.oracle.com/javase/9/tools/jshell.htm`) for more details and possible options.

The command `[/se [setting]` sets configuration information, including the external editor, startup settings, and feedback mode. This command is also used to create a custom feedback mode with customized prompt, format, and truncation values. If no setting is entered, then the current setting for the editor, startup settings, and feedback mode are displayed. The documentation referred to earlier describes all possible settings in all details.

The JShell is going to be even more helpful when integrated inside of the IDE so that a programmer can evaluate expressions on the fly or, even better, they can be evaluated automatically the same way the compiler today evaluates the syntax.

Ahead-of-Time (AOT)

The big claim of Java was write-once-run-anywhere. It was achieved by creating an implementation of **Java Runtime Environment** (**JRE**) for practically all platforms, so the bytecode generated once from the source by Java compiler (`javac` tool) could be executed everywhere where JRE was installed, provided the version of the compiler `javac` was compatible with the version of JRE.

The first releases of JRE were primarily the interpreters of the bytecode and yielded slower performance than some other languages and their compilers, such as C and C++. However, over time, JRE was improved substantially and now produces quite decent results, on a par with many other popular systems. In big part, it is due to the JIT dynamic compiler that converts the bytecodes of the most frequently used methods to the native code. Once generated, the compiled methods (the platform-specific machine code) is executed as needed without any interpretation, thus decreasing the execution time.

To utilize this approach, JRE needs some time for figuring out which methods of the application are used most often. The people working in this area of programming call them hot methods. This period of discovery, until the peak performance is reached, is often called a JVM's warm-up time. It is bigger for the larger and more complex Java applications and can be just a few seconds for smaller ones. However, even after the peak performance is reached, the application might, because of the particular input, start utilizing an execution path never used before and calling the methods that were not compiled yet, thus suddenly degrading the performance. It can be especially consequential when the code not compiled yet belongs to the complex procedures invoked in some rare critical situations, exactly when the best possible performance is needed.

The natural solution was to allow the programmer to decide which components of the application have to be precompiled into the native machine code--those that are more often used (thus decreasing the application's warm-up time), and those that are used not often but have to be executed as quickly as possible (in support of the critical situations and stable performance overall). That was the motivation of the **Java Enhancement ProposalJEP 295: Ahead-of-Time Compilation**:

JIT compilers are fast, but Java programs can become so large that it takes a long time for the JIT to warm up completely. Infrequently used Java methods might never be compiled at all, potentially incurring a performance penalty due to repeated interpreted invocations.

It is worth noticing though that already in JIT compiler, it is possible to decrease the warm-up time by setting the compilation threshold--how many times a method has to be called before it gets compiled into the native code. By default, the number is 1,500. So, if we set it to less than that, the warm-up time will be shorter. It can be done using the option `-XX:CompileThreshold` with the `java` tool. For example, we can set the threshold to 500 as follows (where `Test` is the compiled Java class with the `main()` method in it):

```
java -XX:CompileThreshold=500 -XX:-TieredCompilation Test
```

The option `-XX:-TieredCompilation` was added to disable the tiered compilation because it is enabled by default and does not honor the compilation threshold. The possible drawback is that the 500 threshold might be too low and too many methods will be compiled, thus slowing down the performance and increasing the warm-up time. The best value for this option will vary from application to an application and may even depend on the particular data input with the same application.

Static versus Dynamic Compilation

Many higher level programming languages such as C or C++ used AOT compilation from the very beginning. They are also called **statically compiled** languages. Since AOT (or static) compilers are not constrained by performance requirements (at least not as much as the interpreters at runtime, also called **dynamic compilers**), they can afford to spend the time producing complex code optimizations. On the other hand, the static compilers do not have the runtime (profiling) data, which is especially limiting in the case of dynamically typed languages, Java being one of them. Since the ability of dynamic typing in Java--downcasting to the subtype, querying an object for its type, and other type operations--is one of the pillars of object-oriented programming (principle of polymorphism), AOT compilation for Java becomes even more limited. Lambda expressions pause another challenge for static compilation and are currently not supported yet.

Another advantage of a dynamic compiler is that it can make assumptions and optimize the code accordingly. If the assumption turned out to be wrong, the compiler can try another assumption until the performance goal is achieved. Such a procedure may slow down the application and/or increase the warm-up time, but it may result in a much better performance in the long run. The profile-guided optimization can help a static compiler to move along this path too, but it will always remain limited in its opportunity to optimize by comparison with a dynamic one.

That said, we should not be surprised that the current AOT implementation in JDK 9 is experimental and limited, so far, to 64-bit Linux-based systems only, with both Parallel or G1 garbage collection and the only supported module being `java.base`. Further, AOT compilation should be executed on the same system or a system with the same configuration on which the resulting machine code will be executed. Yet, despite all that, the JEP 295 states:

Performance testing shows that some applications benefit from AOT-compiled code, while others clearly show regressions.

It is worth noting that AOT compilation has been long supported in **Java Micro Edition** (**ME**), but more use cases for AOT in **Java Standard Edition** (**SE**) are yet to be identified, which was one of the reasons the experimental AOT implementation was released with JDK 9-- in order to facilitate the community to try and tell about the practical needs.

The AOT Commands and Procedures

The underlying AOT compilation in JDK 9 is based on the Oracle project `Graal`--an open source compiler introduced with JDK 8 with a goal of improving the performance of the Java dynamic compiler. The AOT group had to modify it, mostly around constants processing and optimization. They have also added probabilistic profiling and a special inlining policy, thus making Grall more suitable for static compilation.

In addition to the existing compiling tool `javac`, a new `jaotc` tool is included in the JDK 9 installation. The resulting AOT shared libraries `.so` are generated using the `libelf` library--the dependency that is going to be removed in the future releases.

To start AOT compilation, a user has to launch `jaotc` and specify classes, JAR files, or modules that have to be compiled. The name of the output library (that holds the generated machine code) can also be passed as the `jaotc` parameter. If not specified, the default name of the output will be `unnamed.so`. As an example, let's look at how the AOT compiler can work with the class `HelloWorld`:

```
public class HelloWorld {
   public static void main(String... args) {
       System.out.println("Hello, World!");
   }
}
```

First, we will generate the bytecode and produce `HelloWorld.class` using `javac`:

```
javac HelloWorld.java
```

Then, we will use the bytecode from the file `HelloWorld.class` to generate machine code into the library `libHelloWorld.so`:

```
jaotc --output libHelloWorld.so HelloWorld.class
```

Now, we can execute the generated library (on the platform with the same specification as the one where `jaotc`was executed) using the `java` tool with an option `-XX:AOTLibrary`:

```
java -XX:AOTLibrary=./libHelloWorld.so HelloWorld
```

The option `-XX:AOTLibrary` allows us to list several AOT libraries separated by commas.

Notice that the `java` tool requires bytecode of all the applications in addition to the native code of some of its components. This fact diminishes the alleged advantage of static compilation, which some AOT enthusiasts claim, that it protects code better from being decompiled. It might be so in the future when bytecode will not be required at runtime if the same class or method is in the AOT library already. However, as of today, it is not the case.

To see if AOT-compiled methods were used, you can add an option `-XX:+PrintAOT`:

```
java -XX:AOTLibrary=./libHelloWorld.so -XX:+PrintAOT HelloWorld
```

It will allow you to see the line loaded `./libHelloWorld.so` AOT library in the output.

If the source code of a class was changed but not pushed (through the `jaotc` tool) into the AOT library, JVM will notice it at runtime because the fingerprint of each compiled class is stored with its native code in the AOT library. JIT will then ignore the code in the AOT library and use the bytecode instead.

The `java` tool in JDK 9 supports a few other flags and options related to AOT:

- `-XX:+/-UseAOT` tells the JVM to use or to ignore AOT-compiled files (by default, it is set to use AOT)
- `-XX:+/-UseAOTStrictLoading` turns on/off the AOT strict loading; if on, it directs JVM to exit if any of the AOT libraries were generated on a platform with a configuration different from the current runtime configuration

The JEP 295 describes the `jaotc` tool's command format as follows:

```
jaotc <options> <name or list>
```

The `name` is a class name or JAR file. The `list` is a colon `:` separated list of class names, modules, JAR files, or directories that contain class files. The `options` is one or many flags from the following list:

- `--output <file>`: This is the output file name (by default, `unnamed.so`)
- `--class-name <class names>`: This is the list of Java classes to compile
- `--jar <jar files>`: This is the list of JAR files to compile
- `--module <modules>`: This is the list of Java modules to compile
- `--directory <dirs>`: This is the list of directories where you can search for files to compile
- `--search-path <dirs>`: This is the list of directories where to search for specified files
- `--compile-commands <file>`: This is the name of the file with compile commands; here is an example:

  ```
  exclude sun.util.resources..*.TimeZoneNames_.*.getContents\(\)\[\
  [Ljava/lang/Object;
  exclude sun.security.ssl.*
  compileOnly java.lang.String.*
  ```

AOT recognizes two compile commands currently:

- `exclude`: This excludes the compilation of specified methods
- `compileOnly`: This compiles only specified methods

Regular expressions are used to specify classes and methods, which are mentioned here:

- `--compile-for-tiered`: This generates profiling code for tiered compilation (by default, profiling code is not generated)
- `--compile-with-assertions`: This generates code with Java assertions (by default, assertions code is not generated)
- `--compile-threads <number>`: This is the number of compilation threads to be used (by default, the smaller value of 16 and number of available CPUs)
- `--ignore-errors`: This ignores all exceptions thrown during class loading (by default, exits on compilation if class loading throws an exception)
- `--exit-on-error`: This exits on compilation errors (by default, failed compilation is skipped, while the compilation of other methods continues)
- `--info`: This prints information about compilation phases
- `--verbose`: This prints more details about compilation phases

- `--debug`: This prints even more details
- `--help`: This prints help information
- `--version`: This prints version information
- `-J<flag>`: This passes a flag directly to the JVM runtime system

As we mentioned already, some applications can improve performance using AOT, while others may become slower. Only testing will provide a definite answer to the question about the usefulness of AOT for each application. In any case, one of the ways to improve performance is to compile and use the AOT library of the `java.base` module:

```
jaotc --output libjava.base.so --module java.base
```

At runtime, the AOT initialization code looks for shared libraries in the `$JAVA_HOME/lib` directory or among the libraries listed by the `-XX:AOTLibrary` option. If shared libraries are found, they are picked up and used. If no shared libraries can be found, AOT will be turned off.

Summary

In this lesson, we described two new tools that can help a developer be more productive (JShell tool) and help improve Java application performance (`jaotc` tool). The examples and steps to use them will help you understand the benefits of their usage and get you started in case you decide to try them.

In the next lesson, we will discuss how to monitor Java applications programmatically using command-line tools. We will also explore how to improve the application performance via multithreading and how to tune the JVM itself after learning about the bottlenecks through monitoring.

Assessments

1. The ________ compiler takes Java bytecode and generates a native machine code so that the resulting binary file can execute natively.
2. Which of the following commands drops a snippet referenced by a name or on ID?
    1. `/d <name or id>`
    2. `/drop <name or id>`
    3. `/dr <name or id>`
    4. `/dp <name or id>`

3. State whether True or False: Shell is Ahead-of-Time tool that is well-known for those who program in Scala, Ruby. It takes a user input, evaluates it, and returns the result after sometime.
4. Which of the following commands is used to list the source you have typed in JShell?
    1. `/l [<name or id>|-all|-start]`
    2. `/m [<name or id>|-all|-start]L`
    3. `/t [<name or id>|-all|-start]`
    4. `/v [<name or id>|-all|-start]`
5. Which of the following regular expressions ignores all exceptions thrown during class loading?
    1. `--exit-on-error`
    2. `–ignores-errors`
    3. `--ignore-errors`
    4. `--exits-on-error`

3
Multithreading and Reactive Programming

In this lesson, we will look at an approach to support a high performance of an application by programmatically splitting the task between several workers. That was how the pyramids were built 4,500 years ago, and this method has not failed to deliver since then. But there is a limitation on how many laborers can be brought to work on the same project. The shared resources provide a ceiling to how much the workforce can be increased, whether the resources are counted in square feet and gallons (as the living quarters and water in the time of the pyramids) or in gigabytes and gigahertz (as the memory and processing power of a computer).

Allocation, usage, and limitations of a living space and computer memory are very similar. However, we perceive the processing power of the human workforce and CPU quite differently. Historians tell us that thousands of ancient Egyptians worked on cutting and moving massive stone blocks at the same time. We do not have any problem understanding what they mean even if we know that these workers rotated all the time, some of them resting or attending to other matters temporarily and then coming back to replace the ones who have finished their annual assignment, others died or got injured and were replaced by the new recruits.

But in case of computer data processing, when we hear about working threads executing at the same time, we automatically assume that they literally do what they are programmed to do in parallel. Only after we look under the hood of such a system we realize that such parallel processing is possible only when the threads are executed each by a different CPU. Otherwise, they time share the same processing power, and we perceive them working at the same time only because the time slots they use are very short--a fraction of the time units we have used in our everyday life. When the threads share the same resource, in computer science we say they do it concurrently.

In this lesson, we will discuss the ways to increase Java application performance by using the workers (threads) that process data concurrently. We will show how to use threads effectively by pooling them, how to synchronize the concurrently accessed data, how to monitor and tune worker threads at runtime, and how to take advantage of the reactive programming concept.

But before doing that, let's revisit the basics of creating and running multiple threads in the same Java process.

Prerequisites

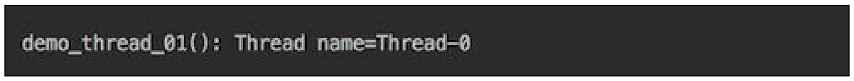
There are principally two ways to create worker threads--by extending the `java.lang.Thread` class and by implementing the `java.lang.Runnable` interface. While extending the `java.lang.Thread` class, we are not required to implement anything:

```
class MyThread extends Thread {
}
```

Our `MyThread` class inherits the `name` property with an automatically generated value and the `start()` method. We can run this method and check the `name`:

```
System.out.print("demo_thread_01(): ");
MyThread t1 = new MyThread();
t1.start();
System.out.println("Thread name=" + t1.getName());
```

If we run this code, the result will be as follows:



```
demo_thread_01(): Thread name=Thread-0
```

As you can see, the generated `name` is `Thread-0`. If we created another thread in the same Java process, the `name` would be `Thread-1` and so on. The `start()` method does nothing. The source code shows that it calls the `run()` method if such a method is implemented.

We can add any other method to the `MyThread` class as follows:

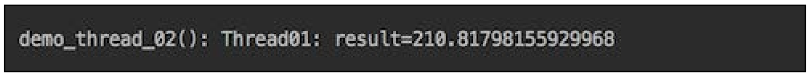
```
class MyThread extends Thread {
    private double result;
    public MyThread(String name){ super(name); }
    public void calculateAverageSqrt(){
        result =  IntStream.rangeClosed(1, 99999)
                           .asDoubleStream()
                           .map(Math::sqrt)
```

```
                             .average()
                             .getAsDouble();
        }
        public double getResult(){ return this.result; }
    }
```

The `calculateAverageSqrt()` method calculates the average square root of the first 99,999 integers and assigns the result to a property that can be accessed anytime. The following code demonstrates how we can use it:

```
    System.out.print("demo_thread_02(): ");
    MyThread t1 = new MyThread("Thread01");
    t1.calculateAverageSqrt();
    System.out.println(t1.getName() + ": result=" + t1.getResult());
```

Running this brings up the following result:



```
demo_thread_02(): Thread01: result=210.81798155929968
```

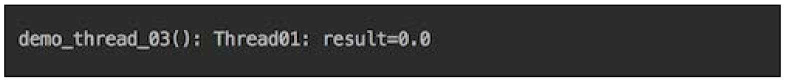
As you would expect, the `calculateAverageSqrt()` method blocks until the calculations are completed. It was executed in the main thread without it taking advantage of multithreading. To do this, we move the functionality in the `run()` method:

```
    class MyThread01 extends Thread {
        private double result;
        public MyThread01(String name){ super(name); }
        public void run(){
            result =  IntStream.rangeClosed(1, 99999)
                             .asDoubleStream()
                             .map(Math::sqrt)
                             .average()
                             .getAsDouble();
        }
        public double getResult(){ return this.result; }
    }
```

Now we call the `start()` method again, as in the first example and expect the result to be calculated:

```
    System.out.print("demo_thread_03(): ");
    MyThread01 t1 = new MyThread01("Thread01");
    t1.start();
    System.out.println(t1.getName() + ": result=" + t1.getResult());
```

However, the output of this code may surprise you:

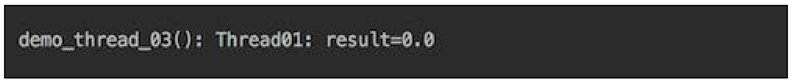


```
demo_thread_03(): Thread01: result=0.0
```

This means that the main thread accessed (and printed) the `t1.getResult()` function before the new `t1` thread finished its calculations. We can experiment and change the implementation of the `run()` method to see if the `t1.getResult()` function can get a partial result:

```
public void run() {
    for (int i = 1; i < 100000; i++) {
        double s = Math.sqrt(1. * i);
        result = result + s;
    }
    result = result / 99999;
}
```

However, if we run the `demo_thread_03()` method again, the result remains the same:



```
demo_thread_03(): Thread01: result=0.0
```

It takes time to create a new thread and get it going. Meanwhile, the `main` thread calls the `t1.getResult()` function immediately, thus getting no results yet.

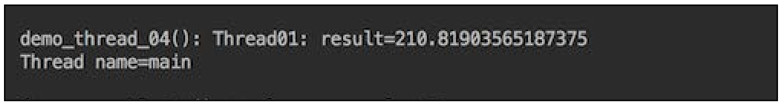
To give the new (child) thread time to complete the calculations, we add the following code:

```
try {
     t1.join();
 } catch (InterruptedException e) {
     e.printStackTrace();
 }
```

The `join()` method tells the current thread to wait until the `t1` thread is finished executing. Let's run the following snippet of code:

```
System.out.print("demo_thread_04(): ");
MyThread01 t1 = new MyThread01("Thread01");
t1.start();
try {
    t1.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println(t1.getName()
                + ": result=" + t1.getResult());
System.out.println("Thread name="
            + Thread.currentThread().getName());
```

You have noticed that we have paused the main thread by 100 ms and added printing of the current thread name, to illustrate what we mean by `main` thread, the name that is assigned automatically to the thread that executes the `main()` method. The output of the previous code is as follows:



```
demo_thread_04(): Thread01: result=210.81903565187375
Thread name=main
```

The delay of 100 ms was enough for the `t1` thread to finish the calculations. That was the first of two ways of creating threads for multithreaded calculation. The second way is to implement the `Runnable` interface. It may be the only way possible if the class that does calculations already extends some other class and you cannot or don't want to use composition for some reasons. The `Runnable` interface is a functional interface (has only one abstract method) with the `run()` method that has to be implemented:

```
    @FunctionalInterface
    public interface Runnable {
        /**
         * When an object implementing interface <code>Runnable</code> is
    used
         * to create a thread, starting the thread causes the object's
         * <code>run</code> method to be called in that separately
    executing
         * thread.
         */
        public abstract void run();
```

We implement this interface in the `MyRunnable` class:

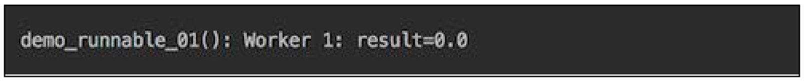
```
    class MyRunnable01 implements Runnable {
        private String id;
        private double result;
        public MyRunnable01(int id) {
            this.id = String.valueOf(id);
        }
        public String getId() { return this.id; }
        public double getResult() { return this.result; }
        public void run() {
            result = IntStream.rangeClosed(1, 99999)
                              .asDoubleStream()
                              .map(Math::sqrt)
                              .average()
                              .getAsDouble();
        }
    }
```

It has the same functionality as the `Thread01` class earlier plus we have added id that allows identifying the thread if necessary since the `Runnable` interface does not have the built-in `getName()` method like the `Thread` class has.

Similarly, if we execute this class without pausing the `main` thread, like this:

```
System.out.print("demo_runnable_01(): ");
MyRunnable01 myRunnable = new MyRunnable01(1);
Thread t1 = new Thread(myRunnable);
t1.start();
System.out.println("Worker " + myRunnable.getId()
           + ": result=" + myRunnable.getResult());
```

The output will be as follows:

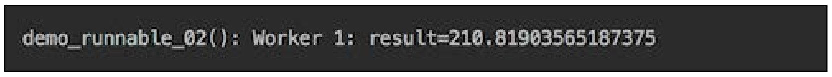


```
demo_runnable_01(): Worker 1: result=0.0
```

We will now add the pause as follows:

```
System.out.print("demo_runnable_02(): ");
MyRunnable01 myRunnable = new MyRunnable01(1);
Thread t1 = new Thread(myRunnable);
t1.start();
try {
    t1.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Worker " + myRunnable.getId()
           + ": result=" + myRunnable.getResult());
```

The result is exactly the same as the one produced by the `Thread01` class:

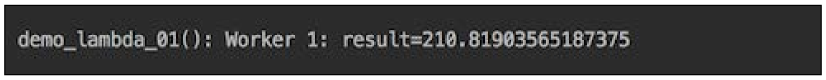


```
demo_runnable_02(): Worker 1: result=210.81903565187375
```

All the previous examples stored the generated result in the class property. But it is not always the case. Typically, the worker thread either passes its value to another thread or stores it in a database or somewhere else externally. In such a case, one can take advantage of the `Runnable` interface being a functional interface and pass the necessary processing function into a new thread as a lambda expression:

```
System.out.print("demo_lambda_01(): ");
String id = "1";
Thread t1 =
    new Thread(() -> IntStream.rangeClosed(1, 99999)
         .asDoubleStream().map(Math::sqrt).average()
         .ifPresent(d -> System.out.println("Worker "
                           + id + ": result=" + d)));
t1.start();
try {
    t1.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

The result is going to be exactly the same, as shown here:



```
demo_lambda_01(): Worker 1: result=210.81903565187375
```

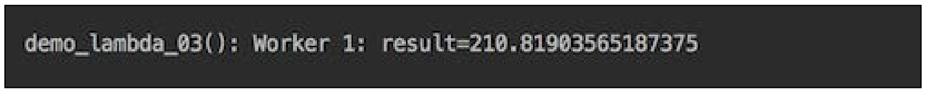
Depending on the preferred style, you can re-arrange the code and isolate the lambda expression in a variable, as follows:

```
Runnable r = () -> IntStream.rangeClosed(1, 99999)
        .asDoubleStream().map(Math::sqrt).average()
      .ifPresent(d -> System.out.println("Worker "
                           + id + ": result=" + d));
Thread t1 = new Thread(r);
```

Alternatively, you can put the lambda expression in a separate method:

```
void calculateAverage(String id) {
    IntStream.rangeClosed(1, 99999)
        .asDoubleStream().map(Math::sqrt).average()
        .ifPresent(d -> System.out.println("Worker "
                              + id + ": result=" + d));
}
void demo_lambda_03() {
    System.out.print("demo_lambda_03(): ");
    Thread t1 = new Thread(() -> calculateAverage("1"));
    ...
}
```

The result is going to be the same, as shown here:



```
demo_lambda_03(): Worker 1: result=210.81903565187375
```

With the basic understanding of threads creation in place, we can now return to the discussion about using the multithreading for building a high-performance application. In other words, after we understand the abilities and resources needed for each worker, we can now talk about logistics of bringing in many of them for such a big-scale project as the Great Pyramid of Giza.

To write code that manages the life cycle of worker threads and their access to the shared resources is possible, but it is quite the same from one application to another. That's why, after several releases of Java, the thread management plumbing became part of the standard JDK library as the `java.util.concurrent` package. This package has a wealth of interfaces and classes that support multithreading and concurrency. We will discuss how to use most of this functionality in the subsequent sections, while talking about thread pools, threads monitoring, thread synchronization, and the related subjects.

Thread Pools

In this section, we will look into the `Executor` interfaces and their implementations provided in the `java.util.concurrent` package. They encapsulate thread management and minimize the time an application developer spends on the writing code related to threads' life cycles.

There are three `Executor` interfaces defined in the `java.util.concurrent` package. The first is the base `Executor` interface has only one `void execute(Runnable r)` method in it. It basically replaces the following:

```
Runnable r = ...;
(new Thread(r)).start()
```

However, we can also avoid a new thread creation by getting it from a pool.

The second is the `ExecutorService` interface extends `Executor` and adds the following groups of methods that manage the life cycle of the worker threads and of the executor itself:

- `submit()`: Place in the queue for the execution of an object of the interface `Runnable` or interface `Callable` (allows the worker thread to return a value); return object of `Future` interface, which can be used to access the value returned by the `Callable` and to manage the status of the worker thread
- `invokeAll()`: Place in the queue for the execution of a collection of interface `Callable` objects return, list of `Future` objects when all the worker threads are complete (there is also an overloaded `invokeAll()` method with timeout)
- `invokeAny()`: Place in the queue for the execution of a collection of interface `Callable` objects; return one `Future` object of any of the worker threads, which has completed (there is also an overloaded `invokeAny()` method with timeout)

Methods that manage the worker threads status and the service itself:

- `shutdown()`: This prevents new worker threads from being submitted to the service
- `isShutdown()`: This checks whether the shutdown of the executor was initiated
- `awaitTermination(long timeout, TimeUnit timeUnit)`: This waits until all worker threads have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first
- `isTerminated()`: This checks whether all the worker threads have completed after the shutdown was initiated; it never returns `true` unless either `shutdown()` or `shutdownNow()` was called first
- `shutdownNow()`: This interrupts each worker thread that is not completed; a worker thread should be written so that it checks its own status (using `Thread.currentThread().isInterrupted()`, for example) periodically and gracefully shuts down on its own; otherwise, it will continue running even after `shutdownNow()` was called

The third interface is `ScheduledExecutorService` that extends `ExecutorService` and adds methods that allow scheduling of the execution (one-time and periodic one) of the worker threads.

A pool-based implementation of ExecutorService can be created using the java.util.concurrent.ThreadPoolExecutor or java.util.concurrent.ScheduledThreadPoolExecutor class. There is also a java.util.concurrent.Executors factory class that covers most of the practical cases. So, before writing a custom code for worker threads pool creation, we highly recommend looking into using the following factory methods of the java.util.concurrent.Executors class:

- newSingleThreadExecutor(): This creates an ExecutorService (pool) instance that executes worker threads sequentially
- newFixedThreadPool(): This creates a thread pool that reuses a fixed number of worker threads; if a new task is submitted when all the worker threads are still executing, it will be set into the queue until a worker thread is available
- newCachedThreadPool(): This creates a thread pool that adds a new thread as needed, unless there is an idle thread created before; threads that have been idle for sixty seconds are removed from the cache
- newScheduledThreadPool(): This creates a thread pool of a fixed size that can schedule commands to run after a given delay, or to execute periodically
- newSingleThreadScheduledExecutor(): This creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically
- newWorkStealingThreadPool(): This creates a thread pool that uses the same work-stealing mechanism used by ForkJoinPool, which is particularly useful in case the worker threads generate other threads, such as in recursive algorithms

Each of these methods has an overloaded version that allows passing in a ThreadFactory that is used to create a new thread when needed. Let's see how it all works in a code sample.

First, we create a MyRunnable02 class that implements Runnable—our future worker threads:

```
class MyRunnable02 implements Runnable {
    private String id;
    public MyRunnable02(int id) {
        this.id = String.valueOf(id);
    }
    public String getId(){ return this.id; }
    public void run() {
        double result = IntStream.rangeClosed(1, 100)
           .flatMap(i -> IntStream.rangeClosed(1, 99999))
           .takeWhile(i ->
                 !Thread.currentThread().isInterrupted())
           .asDoubleStream()
           .map(Math::sqrt)
           .average()
           .getAsDouble();
        if(Thread.currentThread().isInterrupted()){
            System.out.println(" Worker " + getId()
                       + ": result=ignored: " + result);
        } else {
            System.out.println(" Worker " + getId()
                                + ": result=" + result);
        }
}
```

Notice the important difference of this implementation from the previous examples--the `takeWhile(i -> !Thread.currentThread().isInterrupted())` operation allows the stream flowing as long as the thread worker status is not set to interrupted, which happens when the `shutdownNow()` method is called. As soon as the predicate of the `takeWhile()` returns `false` (the worker thread is interrupted), the thread stops producing the result (just ignores the current `result` value). In a real system, it would equate to skipping storing `result` value in the database, for example.

It is worth noting here that using the `interrupted()` status method for checking the thread status in the preceding code may lead to inconsistent results. Since the `interrupted()` method returns the correct state value and then clears the thread state, the second call to this method (or the call to the method `isInterrupted()` after the call to the method `interrupted()`) always returns `false`.

Although it is not the case in this code, we would like to mention here a mistake some developers make while implementing `try/catch` block in a worker thread. For example, if the worker needs to pause and wait for an interrupt signal, the code often looks like this:

```
try {
    Thread.currentThread().wait();
} catch (InterruptedException e) {}
// Do what has to be done
```

The problem with the preceding snippet is that the thread status never becomes interrupted, while the higher level code might be monitoring the worker thread and changes behavior depending on whether the worker has been interrupted or not.

The better implementation is as follows:

```
try {
    Thread.currentThread().wait();
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
// Do what has to be done
```

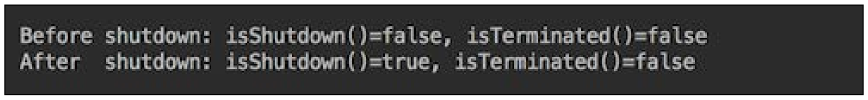
This way the status `interrupted` is set on the thread and can be checked later by the `isInterrupted()` method. To be fair, in many applications, once the thread is interrupted, its code is not checked again. But setting the correct state is a good practice, especially in the cases when you are not the author of the client code.

In the snippet of code with the `join()` method, we did not need to do that because that was the main code (the highest level code) that had to be paused.

Now we can show how to execute the earlier `MyRunnable02` class with a cached pool implementation of the `ExecutiveService` pool (other types of thread pool are used similarly). First, we create the pool, submit three instances of the `MyRunnable02` class for execution and shut down the pool:

```
ExecutorService pool = Executors.newCachedThreadPool();
IntStream.rangeClosed(1, 3).
       forEach(i -> pool.execute(new MyRunnable02(i)));
System.out.println("Before shutdown: isShutdown()="
          + pool.isShutdown() + ", isTerminated()="
                               + pool.isTerminated());
pool.shutdown(); // New threads cannot be submitted
System.out.println("After  shutdown: isShutdown()="
          + pool.isShutdown() + ", isTerminated()="
                               + pool.isTerminated());
```

If we run these lines, we will see the following output:



```
Before shutdown: isShutdown()=false, isTerminated()=false
After  shutdown: isShutdown()=true, isTerminated()=false
```

No surprises here! The `isShutdown()` method returns a `false` value before the `shutdown()` method is called and a `true` value afterward. The `isTerminated()` method returns a `false` value, because none of the worker threads has completed yet.

Let's test the `shutdown()` method by adding the following code after it:

```
try {
    pool.execute(new MyRunnable02(100));
} catch(RejectedExecutionException ex){
    System.err.println("Cannot add another worker-thread to the
service queue:\n" + ex.getMessage());
}
```

The output will now have the following message (the screenshot would be either too big for this page or not readable when fitting):

```
Cannot add another worker-thread to the service queue:
Task com.packt.java9hp.ch09_threads.MyRunnable02@6f7fd0e6
    rejected from java.util.concurrent.ThreadPoolExecutor
    [Shutting down, pool size = 3, active threads = 3,
    queued tasks = 0, completed tasks = 0]
```

As expected, after the `shutdown()` method is called, no more worker threads can be added to the pool.

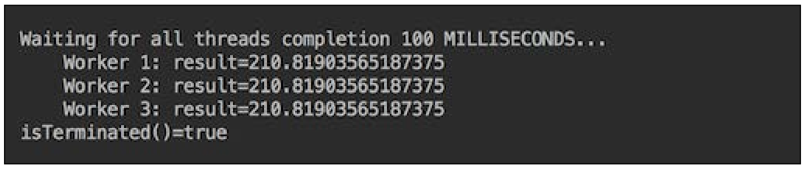
Now, let's see what we can do after the shutdown was initiated:

```
long timeout = 100;
TimeUnit timeUnit = TimeUnit.MILLISECONDS;
System.out.println("Waiting for all threads completion "
                     + timeout + " " + timeUnit + "...");
// Blocks until timeout or all threads complete execution
boolean isTerminated =
                pool.awaitTermination(timeout, timeUnit);
System.out.println("isTerminated()=" + isTerminated);
if (!isTerminated) {
    System.out.println("Calling shutdownNow()...");
    List<Runnable> list = pool.shutdownNow();
    printRunningThreadIds(list);
    System.out.println("Waiting for threads completion "
                     + timeout + " " + timeUnit + "...");
    isTerminated =
                pool.awaitTermination(timeout, timeUnit);
    if (!isTerminated){
        System.out.println("Some threads are running...");
    }
    System.out.println("Exiting.");
}
```

The `printRunningThreadIds()` method looks like this:

```
void printRunningThreadIds(List<Runnable> l){
    String list = l.stream()
            .map(r -> (MyRunnable02)r)
            .map(mr -> mr.getId())
            .collect(Collectors.joining(","));
    System.out.println(l.size() + " thread"
       + (l.size() == 1 ? " is" : "s are") + " running"
            + (l.size() > 0 ? ": " + list : "") + ".");
}
```

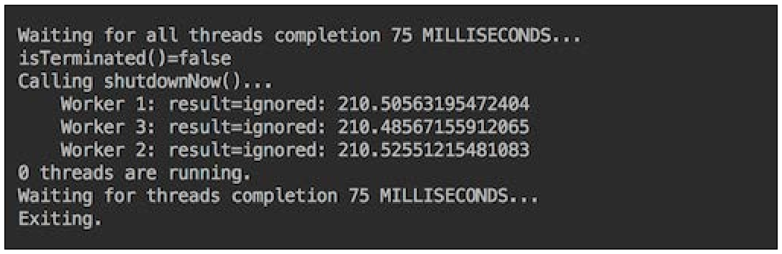
The output of the preceding code will be as follows:



```
Waiting for all threads completion 100 MILLISECONDS...
    Worker 1: result=210.81903565187375
    Worker 2: result=210.81903565187375
    Worker 3: result=210.81903565187375
isTerminated()=true
```

This means that 100 ms was enough for each worker thread to complete the calculations. (Notice, if you try to reproduce this data on your computer, the results might be slightly different because of the difference in performance, so you would need to adjust the timeout.)

When we have decreased the wait time to 75 ms, the output became as follows:



```
Waiting for all threads completion 75 MILLISECONDS...
isTerminated()=false
Calling shutdownNow()...
    Worker 1: result=ignored: 210.50563195472404
    Worker 3: result=ignored: 210.48567155912065
    Worker 2: result=ignored: 210.52551215481083
0 threads are running.
Waiting for threads completion 75 MILLISECONDS...
Exiting.
```

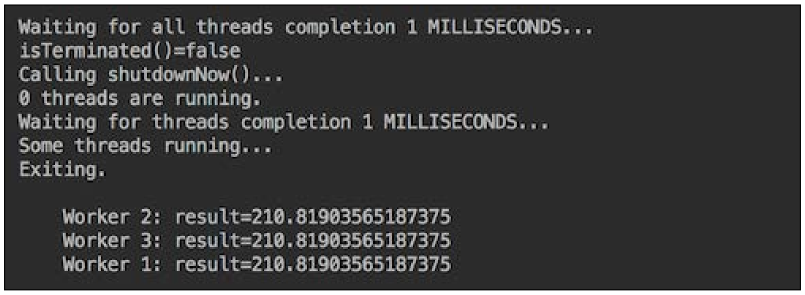
The 75 ms on our computer was not enough to let all the threads complete, so they were interrupted by `shutdownNow()` and their partial results were ignored.

Let's now remove the check of the interrupted status in the `MyRunnable01` class:

```
class MyRunnable02 implements Runnable {
    private String id;
    public MyRunnable02(int id) {
        this.id = String.valueOf(id);
    }
```

```
        public String getId(){ return this.id; }
        public void run() {
            double result = IntStream.rangeClosed(1, 100)
               .flatMap(i -> IntStream.rangeClosed(1, 99999))
               .asDoubleStream()
               .map(Math::sqrt)
               .average()
               .getAsDouble();
            System.out.println(" Worker " + getId()
                                  + ": result=" + result);
    }
```

Without the check, even if we decrease the timeout to 1 ms, the result will be as follows:



```
Waiting for all threads completion 1 MILLISECONDS...
isTerminated()=false
Calling shutdownNow()...
0 threads are running.
Waiting for threads completion 1 MILLISECONDS...
Some threads running...
Exiting.

    Worker 2: result=210.81903565187375
    Worker 3: result=210.81903565187375
    Worker 1: result=210.81903565187375
```

That is because the worker threads have never noticed that somebody tried to interrupt them and completed their assigned calculations. This last test demonstrates the importance of watching for the interrupted state in a work thread in order to avoid many possible problems, namely, data corruption and memory leak.

The demonstrated cached pool works fine and poses no problem if the worker threads perform short tasks and their number cannot grow excessively large. If you need to have more control over the max number of worker threads running at any time, use the fixed size thread pool. We will discuss how to choose the pool size in one of the following sections of this lesson.

The single-thread pool is a good fit for executing tasks in a certain order or in the case when each of them requires so many resources that cannot be executed in parallel with another. Yet another case for using a single-thread execution would be for workers that modify the same data, but the data cannot be protected from the parallel access another way. The thread synchronization will be discussed in more detail in one of the following sections of this lesson, too.

In our sample code, so far we have only included the `execute()` method of the `Executor` interface. We will demonstrate the other methods of the `ExecutorService` pool in the following section while discussing threads monitoring.

And the last remark in this section. The worker threads are not required to be objects of the same class. They may represent completely different functionality and still be managed by one pool.

Monitoring Threads

There are two ways to monitor threads, programmatically and using the external tools. We have already seen how the result of a worker calculation could be checked. Let's revisit that code. We will also slightly modify our worker implementation:

```
class MyRunnable03 implements Runnable {
  private String name;
  private double result;
  public String getName(){ return this.name; }
  public double getResult() { return this.result; }
  public void run() {
    this.name = Thread.currentThread().getName();
    double result = IntStream.rangeClosed(1, 100)
      .flatMap(i -> IntStream.rangeClosed(1, 99999))
      .takeWhile(i -> !Thread.currentThread().isInterrupted())
      .asDoubleStream().map(Math::sqrt).average().getAsDouble();
    if(!Thread.currentThread().isInterrupted()){
      this.result = result;
    }
  }
}
```

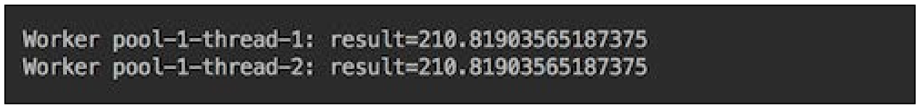
For the worker thread identification, instead of custom ID, we now use the thread name assigned automatically at the time of the execution (that is why we assign the `name` property in the `run()` method that is called in the context of the execution when the thread acquires its name). The new class `MyRunnable03` can be used like this:

```
void demo_CheckResults() {
    ExecutorService pool = Executors.newCachedThreadPool();
    MyRunnable03 r1 = new MyRunnable03();
    MyRunnable03 r2 = new MyRunnable03();
    pool.execute(r1);
    pool.execute(r2);
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Worker " + r1.getName() + ": result=" +
r1.getResult());
    System.out.println("Worker " + r2.getName() + ": result=" +
r2.getResult());
    shutdown(pool);
}
```

The `shutdown()` method contains the following code:

```
    void shutdown(ExecutorService pool) {
        pool.shutdown();
        try {
            if(!pool.awaitTermination(1, TimeUnit.SECONDS)){
                pool.shutdownNow();
            }
        } catch (InterruptedException ie) {}
    }
```

If we run the preceding code, the output will be as follows:



```
Worker pool-1-thread-1: result=210.81903565187375
Worker pool-1-thread-2: result=210.81903565187375
```

If the result on your computer is different, try to increase the input value to the `sleepMs()` method.

Another way to get information about the application worker threads is by using the `Future` interface. We can access this interface using the `submit()` method of the `ExecutorService` pool, instead of the `execute()`, `invokeAll()`, or `invokeAny()` methods. This code shows how to use the `submit()` method:

```
    ExecutorService pool = Executors.newCachedThreadPool();
    Future f1 = pool.submit(new MyRunnable03());
    Future f2 = pool.submit(new MyRunnable03());
    printFuture(f1, 1);
    printFuture(f2, 2);
    shutdown(pool);
```

The `printFuture()` method has the following implementation:

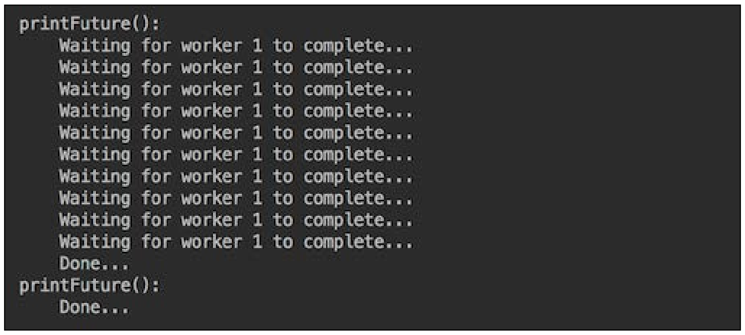
```
    void printFuture(Future future, int id) {
        System.out.println("printFuture():");
        while (!future.isCancelled() && !future.isDone()){
            System.out.println("    Waiting for worker "
                                    + id + " to complete...");
            sleepMs(10);
        }
        System.out.println("    Done...");
    }
```

The `sleepMs()` method contains the following code:

```
    void sleepMs(int sleepMs) {
        try {
            TimeUnit.MILLISECONDS.sleep(sleepMs);
        } catch (InterruptedException e) {}
    }
```

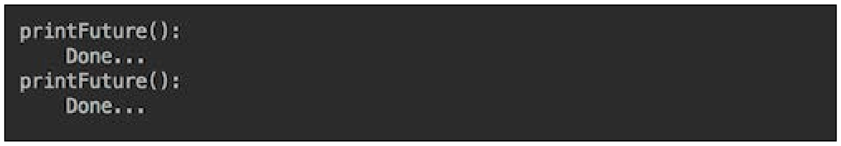
We prefer this implementation instead of the traditional `Thread.sleep()` because it is explicit about the time units used.

If we execute the previous code, the result will be similar to the following:



```
printFuture():
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Done...
printFuture():
    Done...
```

The `printFuture()` method has blocked the main thread execution until the first thread has completed. Meanwhile, the second thread has completed too. If we call the `printFuture()` method after the `shutdown()` method, both the threads would complete by that time already because we have set a wait time of 1 second (see the `pool.awaitTermination()` method), which is enough for them to finish their job:



```
printFuture():
    Done...
printFuture():
    Done...
```

If you think it is not much information from a threads monitoring point of view, the `java.util.concurrent` package provides more capabilities via the `Callable` interface. It is a functional interface that allows returning any object (containing results of the worker thread calculations) via the `Future` object using `ExecutiveService` methods--`submit()`, `invokeAll()`, and `invokeAny()`. For example, we can create a class that contains the result of a worker thread:

```
class Result {
    private double result;
    private String workerName;
    public Result(String workerName, double result) {
        this.result = result;
        this.workerName = workerName;
    }
    public String getWorkerName() { return workerName; }
    public double getResult() { return result;}
}
```

We have included the name of the worker thread too for monitoring which thread generated the result that is presented. The class that implements the `Callable` interface may look like this:

```
class MyCallable01<T> implements Callable {
  public Result call() {
    double result = IntStream.rangeClosed(1, 100)
       .flatMap(i -> IntStream.rangeClosed(1, 99999))
       .takeWhile(i -> !Thread.currentThread().isInterrupted())
       .asDoubleStream().map(Math::sqrt).average().getAsDouble();

    String workerName = Thread.currentThread().getName();
    if(Thread.currentThread().isInterrupted()){
        return new Result(workerName, 0);
    } else {
        return new Result(workerName, result);
    }
  }
}
```

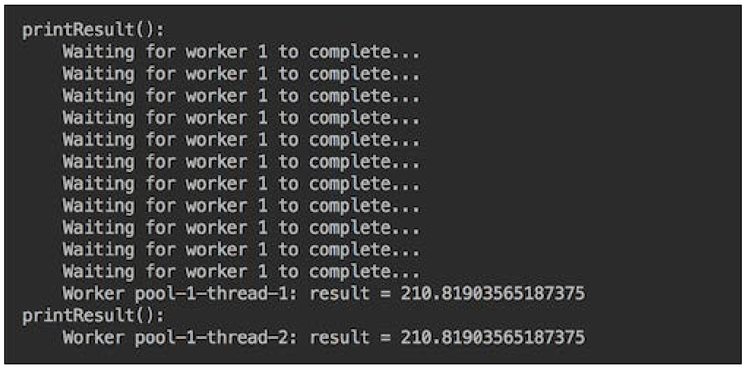
And here is the code that uses the `MyCallable01` class:

```
ExecutorService pool = Executors.newCachedThreadPool();
Future f1 = pool.submit(new MyCallable01<Result>());
Future f2 = pool.submit(new MyCallable01<Result>());
printResult(f1, 1);
printResult(f2, 2);
shutdown(pool);
```

The `printResult()` method contains the following code:

```
void printResult(Future<Result> future, int id) {
    System.out.println("printResult():");
    while (!future.isCancelled() && !future.isDone()){
        System.out.println("    Waiting for worker "
                              + id + " to complete...");
        sleepMs(10);
    }
    try {
        Result result = future.get(1, TimeUnit.SECONDS);
        System.out.println("    Worker "
                + result.getWorkerName() + ": result = "
                                   + result.getResult());
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

The output of this code may look like this:



```
printResult():
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Waiting for worker 1 to complete...
    Worker pool-1-thread-1: result = 210.81903565187375
printResult():
    Worker pool-1-thread-2: result = 210.81903565187375
```

The earlier output shows, as in the previous examples, that the `printResult()` method waits until the first of the worker threads finishes, so the second thread manages to finish its job at the same time. The advantage of using `Callable`, as you can see, is that we can retrieve the actual result from a `Future` object, if we need it.

The usage of the `invokeAll()` and `invokeAny()` methods looks similar:

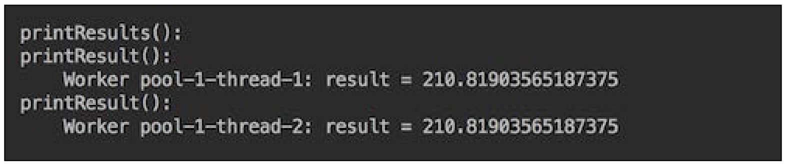
```
ExecutorService pool = Executors.newCachedThreadPool();
try {
    List<Callable<Result>> callables =
              List.of(new MyCallable01<Result>(),
                           new MyCallable01<Result>());
```

```
        List<Future<Result>> futures =
                                 pool.invokeAll(callables);
        printResults(futures);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    shutdown(pool);
```

The `printResults()` method is using the `printResult()` method, which you already know:

```
    void printResults(List<Future<Result>> futures) {
        System.out.println("printResults():");
        int i = 1;
        for (Future<Result> future : futures) {
            printResult(future, i++);
        }
    }
```

If we run the preceding code, the output will be as follows:



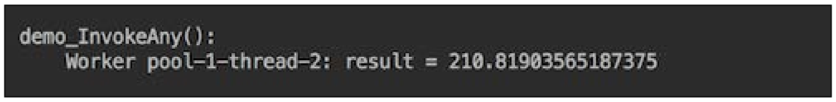
```
printResults():
printResult():
    Worker pool-1-thread-1: result = 210.81903565187375
printResult():
    Worker pool-1-thread-2: result = 210.81903565187375
```

As you can see, there is no more waiting for the worker thread completing the job. That is so because the `invokeAll()` method returns the collection of the `Future` object after all the jobs have completed.

The `invokeAny()` method behaves similarly. If we run the following code:

```
    System.out.println("demo_InvokeAny():");
    ExecutorService pool = Executors.newCachedThreadPool();
    try {
        List<Callable<Result>> callables =
                       List.of(new MyCallable01<Result>(),
                                new MyCallable01<Result>());
        Result result = pool.invokeAny(callables);
        System.out.println("    Worker "
                         + result.getWorkerName()
                      + ": result = " + result.getResult());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
    shutdown(pool);
```

The following will be the output:



```
demo_InvokeAny():
    Worker pool-1-thread-2: result = 210.81903565187375
```

These are the basic techniques for monitoring the threads programmatically, but one can easily extend our examples to cover more complicated cases tailored to the needs of a specific application. In *Lesson 5*, *Making Use of New APIs to Improve Your Code*, we will also discuss another way to programmatically monitor worker threads using the `java.util.concurrent.CompletableFuture` class introduced in JDK 8 and extended in JDK 9.

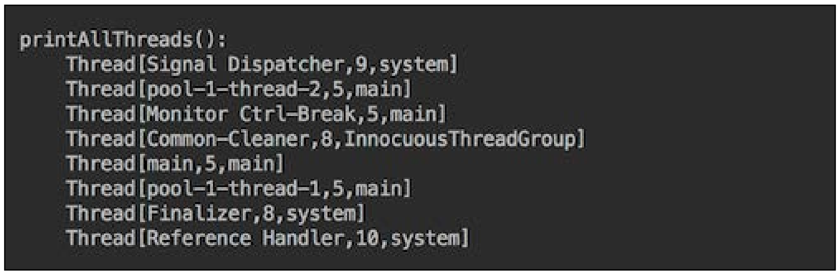
If necessary, it is possible to get information not only about the application worker threads, but also about all other threads in the JVM process using the `java.lang.Thread` class:

```
void printAllThreads() {
    System.out.println("printAllThreads():");
    Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
    for(Thread t: map.keySet()){
        System.out.println("    " + t);
    }
```

Now, let's call this method as follows:

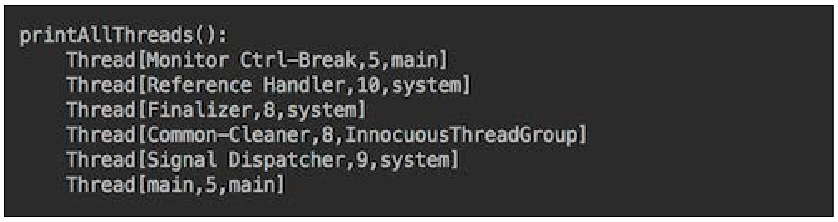
```
void demo_CheckResults() {
    ExecutorService pool = Executors.newCachedThreadPool();
    MyRunnable03 r1 = new MyRunnable03();
    MyRunnable03 r2 = new MyRunnable03();
    pool.execute(r1);
    pool.execute(r2);
    sleepMs(1000);
    printAllThreads();
    shutdown(pool);
}
```

The result looks like this:



```
printAllThreads():
    Thread[Signal Dispatcher,9,system]
    Thread[pool-1-thread-2,5,main]
    Thread[Monitor Ctrl-Break,5,main]
    Thread[Common-Cleaner,8,InnocuousThreadGroup]
    Thread[main,5,main]
    Thread[pool-1-thread-1,5,main]
    Thread[Finalizer,8,system]
    Thread[Reference Handler,10,system]
```

We took advantage of the `toString()` method of the `Thread` class that prints only the thread name, priority, and the thread group it belongs to. And we see the two application threads we have created explicitly (in addition to the `main` thread) in the list under the names `pool-1-thread-1` and `pool-1-thread-2`. But if we call the `printAllThreads()` method after calling the `shutdown()` method, the output will be as follows:



```
printAllThreads():
    Thread[Monitor Ctrl-Break,5,main]
    Thread[Reference Handler,10,system]
    Thread[Finalizer,8,system]
    Thread[Common-Cleaner,8,InnocuousThreadGroup]
    Thread[Signal Dispatcher,9,system]
    Thread[main,5,main]
```

We do not see the `pool-1-thread-1` and `pool-1-thread-2` threads in the list anymore because the `ExecutorService` pool has been shut down.

We could easily add the stack trace information pulled from the same map:

```
void printAllThreads() {
    System.out.println("printAllThreads():");
    Map<Thread, StackTraceElement[]> map
                               = Thread.getAllStackTraces();
    for(Thread t: map.keySet()){
        System.out.println("   " + t);
        for(StackTraceElement ste: map.get(t)){
            System.out.println("        " + ste);
        }
    }
}
```

However, that would take too much space on the book page. In *Lesson 5, Making Use of New APIs to Improve Your Code*, while presenting new Java capabilities that came with JDK 9, we will also discuss a better way to access a stack trace via the `java.lang.StackWalker` class.

The `Thread` class object has several other methods that provide information about the thread, which are as follows:

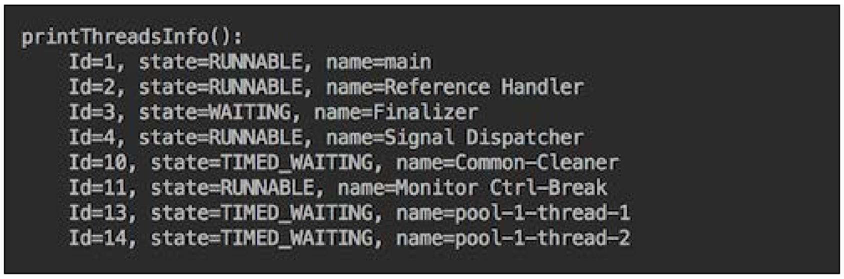
- `dumpStack()`: This prints a stack trace to the standard error stream
- `enumerate(Thread[] arr)`: This copies active threads in the current thread's thread group and its subgroups into the specified array `arr`
- `getId()`: This provides the thread's ID
- `getState()`: This reads the state of the thread; the possible values from `enum Thread.State` can be one of the following:
    - `NEW`: This is the thread that has not yet started
    - `RUNNABLE`: This is the thread that is currently being executed
    - `BLOCKED`: This is the thread that is blocked waiting for a monitor lock to be released
    - `WAITING`: This is the thread that is waiting for an interrupt signal
    - `TIMED_WAITING`: This is the thread that is waiting for an interrupt signal up to a specified waiting time
    - `TERMINATED`: This is the thread that has exited
- `holdsLock(Object obj)`: This indicates whether the thread holds the monitor lock on the specified object
- `interrupted()` or `isInterrupted()`: This indicates whether the thread has been interrupted (received an interrupt signal, meaning that the flag interrupted was set to `true`)
- `isAlive()`: This indicates whether the thread is alive
- `isDaemon()`: This indicates whether the thread is a daemon thread.

The `java.lang.management` package provides similar capabilities for monitoring threads. Let's run this code snippet, for example:

```
void printThreadsInfo() {
    System.out.println("printThreadsInfo():");
    ThreadMXBean threadBean =
                      ManagementFactory.getThreadMXBean();
    long ids[] = threadBean.getAllThreadIds();
    Arrays.sort(ids);
```

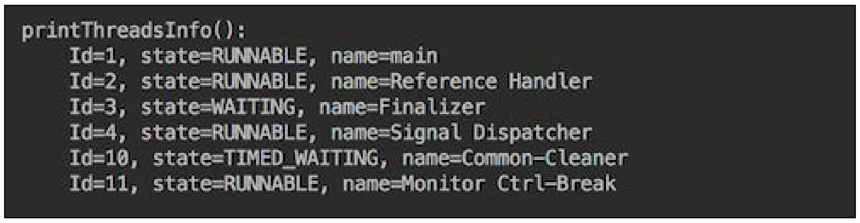
```
        ThreadInfo[] tis = threadBean.getThreadInfo(ids, 0);
        for (ThreadInfo ti : tis) {
            if (ti == null) continue;
            System.out.println("    Id=" + ti.getThreadId()
                               + ", state=" + ti.getThreadState()
                                  + ", name=" + ti.getThreadName());
        }
    }
```

For better presentation, we took advantage of having thread IDs listed and, as you could see previously, have sorted the output by ID. If we call the `printThreadsInfo()` method before the `shutdown()` method the output will be as follows:



```
printThreadsInfo():
    Id=1, state=RUNNABLE, name=main
    Id=2, state=RUNNABLE, name=Reference Handler
    Id=3, state=WAITING, name=Finalizer
    Id=4, state=RUNNABLE, name=Signal Dispatcher
    Id=10, state=TIMED_WAITING, name=Common-Cleaner
    Id=11, state=RUNNABLE, name=Monitor Ctrl-Break
    Id=13, state=TIMED_WAITING, name=pool-1-thread-1
    Id=14, state=TIMED_WAITING, name=pool-1-thread-2
```

However, if we call the `printThreadsInfo()` method after the `shutdown()` method, the output will not include our worker threads anymore, exactly as in the case of using the `Thread` class API:



```
printThreadsInfo():
    Id=1, state=RUNNABLE, name=main
    Id=2, state=RUNNABLE, name=Reference Handler
    Id=3, state=WAITING, name=Finalizer
    Id=4, state=RUNNABLE, name=Signal Dispatcher
    Id=10, state=TIMED_WAITING, name=Common-Cleaner
    Id=11, state=RUNNABLE, name=Monitor Ctrl-Break
```

The `java.lang.management.ThreadMXBean` interface provides a lot of other useful data about threads. You can refer to the official API on the Oracle website about this interface for more information check this link: `https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/management/ThreadMXBean.html`.

In the list of threads mentioned earlier, you may have noticed the `Monitor Ctrl-Break` thread. This thread provides another way to monitor the threads in the JVM process. Pressing the *Ctrl* and *Break* keys on Windows causes the JVM to print a thread dump to the application's standard output. On Oracle Solaris or Linux operating systems, the same effect has the combination of the *Ctrl* key and the backslash `\`. This brings us to the external tools for thread monitoring.

In case you don't have access to the source code or prefer to use the external tools for the threads monitoring, there are several diagnostic utilities available with the JDK installation. In the following list, we mention only the tools that allow for thread monitoring and describe only this capability of the listed tools (although they have other extensive functionality too):

- The `jcmd` utility sends diagnostic command requests to the JVM on the same machine using the JVM process ID or the name of the main class: `jcmd <process id/main class> <command> [options]`, where the `Thread.print` option prints the stack traces of all the threads in the process.
- The JConsole monitoring tool uses the built-in JMX instrumentation in the JVM to provide information about the performance and resource consumption of running applications. It has a thread tab pane that shows thread usage over time, the current number of live threads, the highest number of live threads since the JVM started. It is possible to select the thread and its name, state, and stack trace, as well as, for a blocked thread, the synchronizer that the thread is waiting to acquire, and the thread owning the lock. Use the **Deadlock Detection** button to identify the deadlock. The command to run the tool is `jconsole <process id>` or (for remote application) `jconsole <hostname>:<port>`, where `port` is the port number specified with the JVM start command that enabled the JMX agent.
- The `jdb` utility is an example command line debugger. It can be attached to the JVM process and allows you to examine threads.
- The `jstack` command line utility can be attached to the JVM process and print the stack traces of all threads, including JVM internal threads, and optionally native stack frames. It allows you to detect deadlocks too.
- **Java Flight Recorder** (**JFR**) provides information about the Java process, including threads waiting for locks, garbage collections, and so on. It also allows getting thread dumps, which are similar to the one generated by the `Thread.print` diagnostic command or by using the jstack tool. It is possible to set up **Java Mission Control** (**JMC**) to dump a flight recording if a condition is met. JMC UI contains information about threads, lock contention, and other latencies. Although JFR is a commercial feature, it is free for developer desktops/laptops, and for evaluation purposes in test, development, and production environments.

[You can find more details about these and other diagnostic tools in the official Oracle documentation at `https://docs.oracle.com/javase/9/troubleshoot/diagnostic-tools.htm`.]

Sizing Thread Pool Executors

In our examples, we have used a cached thread pool that creates a new thread as needed or, if available, reuses the thread already used, but which completed its job and returned to the pool for a new assignment. We did not worry about too many threads created because our demo application had two worker threads at the most and they were quite short lived.

But in the case where an application does not have a fixed limit of the worker threads it might need or there is no good way to predict how much memory a thread may take or how long it can execute, setting a ceiling on the worker thread count prevents an unexpected degradation of the application performance, running out of memory or depletion of any other resources the worker threads use. If the thread behavior is extremely unpredictable, a single thread pool might be the only solution, with an option of using a custom thread pool executor (more about this last option is explained later). But in most of the cases, a fixed-size thread pool executor is a good practical compromise between the application needs and the code complexity. Depending on the specific requirements, such an executor might be one of these three flavors:

- A straightforward, fixed-sized `ExecutorService.newFixedThreadPool(int nThreads)` pool that does not grow beyond the specified size, but does not adopt either
- Several `ExecutorService.newScheduledThreadPool(int nThreads)` pools that allow scheduling different groups of threads with a different delay or cycle of execution
- `ExecutorService.newWorkStealingPool(int parallelism)` that adapts to the specified number of CPUs, which you may set higher or smaller than the actual CPUs count on your computer

Setting the fixed size in any of the preceding pools too low may deprive the application of the chance to utilize the available resources effectively. So, before selecting the pool size, it is advisable to spend some time on monitoring it and tuning JVM (see how to do it in one of the sections of this lesson) with the goal of the identification of the idiosyncrasy of the application behavior. In fact, the cycle deploy-monitor-tune-adjust has to be repeated throughout the application life cycle in order to accommodate and take advantage of the changes that happened in the code or the executing environment.

The first parameter you take into account is the number of CPUs in your system, so the thread pool size can be at least as big as the CPU's count. Then, you can monitor the application and see how much time each thread engages the CPU and how much of the time it uses other resources (such as I/O operations). If the time spent not using the CPU is comparable with the total executing time of the thread, then you can increase the pool size by **time not using CPU/total executing time**. But that is in the case that another resource (disk or database) is not a subject of contention between the threads. If the latter is the case, then you can use that resource instead of the CPU as the delineating factor.

Assuming the worker threads of your application are not too big or too long executing and belong to the mainstream population of the typical working threads that complete their job in a reasonably short period of time, you can increase the pool size by adding the (rounded up) ratio of the desired response time and the time a thread uses CPU or another most contentious resource. This means that, with the same desired response time, the less a thread uses CPU or another concurrently accessed resource, the bigger the pool size should be. If the contentious resource has its own ability to improve concurrent access (like a connection pool in the database), consider utilizing that feature first.

If the required number of threads running at the same time changes at runtime under the different circumstances, you can make the pool size dynamic and create a new pool with a new size (shutting down the old pool after all its threads have completed). The recalculation of the size of a new pool might be necessary also after you add to remove the available resources. You can use `Runtime.getRuntime().availableProcessors()` to programmatically adjust the pool size based on the current count of the available CPUs, for example.

If none of the ready-to-use thread pool executor implementations that come with the JDK suit the needs of a particular application, before writing the thread managing code from scratch, try to use the `java.util.concurrent.ThreadPoolExecutor` class first. It has several overloaded constructors.

To give you an idea of its capabilities, here is the constructor with the biggest number of options:

```
ThreadPoolExecutor (int corePoolSize, int maximumPoolSize, long
keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
ThreadFactory threadFactory, RejectedExecutionHandler handler)
```

The earlier mentioned parameters are (quoting from the JavaDoc):

- `corePoolSize`: This is the number of threads to keep in the pool, even if they are idle unless `allowCoreThreadTimeOut` is set
- `maximumPoolSize`: This is the maximum number of threads to allow in the pool
- `keepAliveTime`: When the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating
- `unit`: This is the time unit for the `keepAliveTime` argument
- `workQueue`: This is the queue to use for holding tasks before they are executed, this queue will hold only the `Runnable` tasks submitted by the execute method
- `threadFactory`: This is the factory to use when the executor creates a new thread
- `handler`: This is the handler to use when the execution is blocked because the thread bounds and queue capacities are reached

Each of the previous constructor parameters except the `workQueue` parameter can also be set via the corresponding setter after the object of the `ThreadPoolExecutor` method has been created, thus allowing more flexibility in dynamic adjustment of the existing pool characteristics.

Thread Synchronization

We have collected enough people and resources such as food, water, and tools for the pyramid building. We have divided people into teams and assigned each team a task. A number (a pool) of people are living in the nearby village on a standby mode, ready to replace the ones that got sick or injured on their assignment. We adjusted the workforce count so that there are only a few people who will remain idle in the village. We rotate the teams through the work-rest cycle to keep the project going at maximum speed. We monitored the process and have adjusted the number of teams and the flow of supplies they need so that there are no visible delays and there is steady measurable progress in the project as a whole. Yet, there are many moving parts overall and various small and big unexpected incidents and problems happen all the time.

To make sure that the workers and teams do not step on each other and that there is some kind of traffic regulation so that the next technological step does not start until the previous one is finished, the main architect sends his representatives to all the critical points of the construction site. These representatives make sure that the tasks are executed with the expected quality and in the prescribed order. They have the power to stop the next team from starting their job until the previous team has not finished yet. They act like traffic cops or the locks that can shut down the access to the workplace or allow it, if/when necessary.

The job these representatives are doing can be defined in the modern language as a coordination or synchronization of actions of the executing units. Without it, the results of the efforts of the thousands of workers would be unpredictable. The big picture from ten thousand feet would look smooth and harmonious, as the farmers' fields from the windows of an airplane. But without closer inspection and attention to the critical details, this perfect looking picture may bring a poor harvest, if any.

Similarly, in the quiet electronic space of the multithreaded execution environment, the working threads have to be synchronized if they share access to the same working place. For example, let's create the following class-worker for a thread:

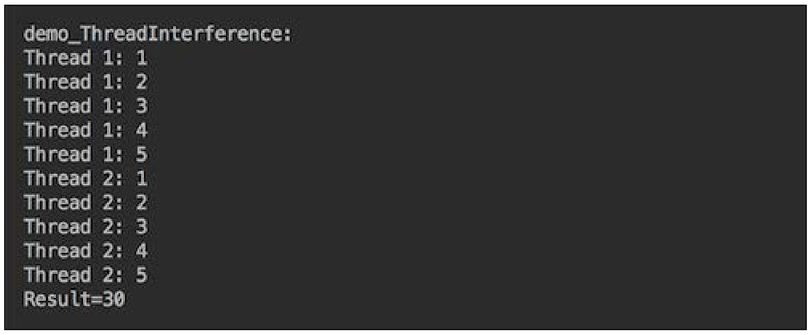
```
class MyRunnable04 implements Runnable {
  private int id;
  public MyRunnable04(int id) { this.id = id; }
  public void run() {
    IntStream.rangeClosed(1, 5)
      .peek(i -> System.out.println("Thread "+id+": "+ i))
      .forEach(i -> Demo04Synchronization.result += i);
    }
}
```

As you can see, it sequentially adds 1, 2, 3, 4, 5 (so, that the resulting total is expected to be 15) to the static property of the `Demo04Synchronization` class:

```
public class Demo04Synchronization {
    public static int result;
    public static void main(String... args) {
        System.out.println();
        demo_ThreadInterference();
    }
    private static void demo_ThreadInterference(){
        System.out.println("demo_ThreadInterference: ");
        MyRunnable04 r1 = new MyRunnable04(1);
        Thread t1 = new Thread(r1);
        MyRunnable04 r2 = new MyRunnable04(2);
        Thread t2 = new Thread(r2);
```

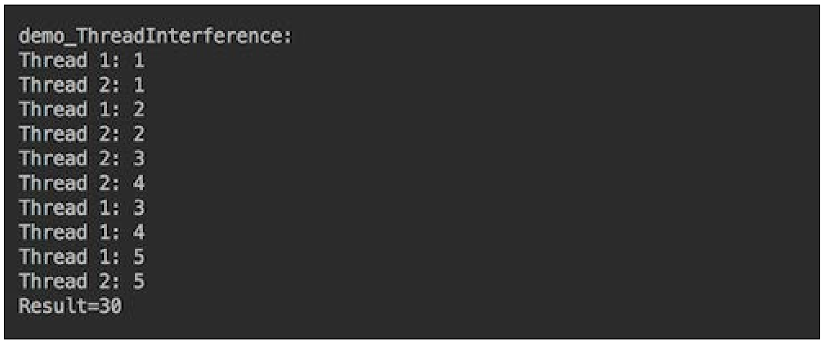
```
            t1.start();
            sleepMs(100);
            t2.start();
            sleepMs(100);
            System.out.println("Result=" + result);
        }
        private static void sleepMs(int sleepMs) {
            try {
                TimeUnit.MILLISECONDS.sleep(sleepMs);
            } catch (InterruptedException e) {}
        }
    }
```

In the earlier code, while the main thread pauses for 100 ms the first time, the thread `t1` brings the value of the variable result to 15, then the thread `t2` adds another 15 to get the total of 30. Here is the output:



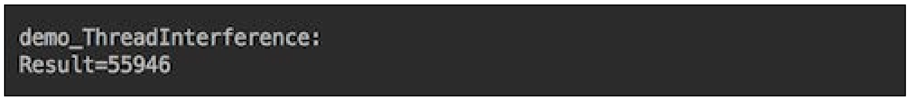
```
demo_ThreadInterference:
Thread 1: 1
Thread 1: 2
Thread 1: 3
Thread 1: 4
Thread 1: 5
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
Thread 2: 5
Result=30
```

If we remove the first pause of 100 ms, the threads will work concurrently:



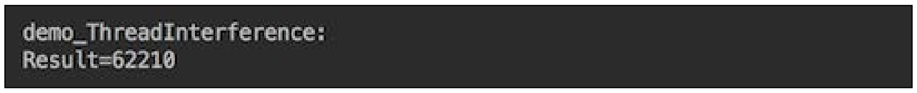
```
demo_ThreadInterference:
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 2: 3
Thread 2: 4
Thread 1: 3
Thread 1: 4
Thread 1: 5
Thread 2: 5
Result=30
```

The final result is still 30. We feel good about this code and deploy it to production as a well-tested code. However, if we increase the number of additions from 5 to 250, for example, the result becomes unstable and changes from run to run. Here is the first run (we commented out the printout in each thread in order to save space):



```
demo_ThreadInterference:
Result=55946
```

And here is the output of another run:



```
demo_ThreadInterference:
Result=62210
```

It demonstrates the fact that the `Demo04Synchronization.result += i` operation is not atomic. This means it consists of several steps, reading the value from the `result` property, adding a value to it, assigning the resulting sum back to the `result` property. This allows the following scenario, for example:

- Both the threads have read the current value of `result` (so each of the threads has a copy of the same original `result` value)
- Each thread adds another integer to the same original one
- The first thread assigns the sum to the `result` property
- The second thread assigns its sum to the `result` property

As you can see, the second thread did not know about the addition the first thread made and has overwritten the value assigned to the `result` property by the first thread. But such thread interleaving does not happen every time. It is just a game of chance. That's why we did not see such an effect with five numbers only. But the probability of this happening increases with the growth of the number of concurrent actions.

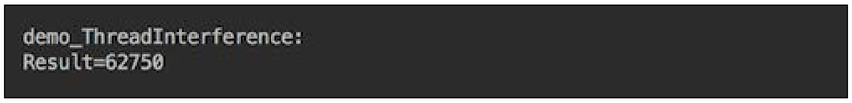
A similar thing could happen during the pyramid building too. The second team could start doing something before the first team has finished their task. We definitely need a **synchronizer** and it comes with a `synchronized` keyword. Using it, we can create a method (an architect representative) in the `Demo04Synchronization` class that will control access to the `result` property and add to it this keyword:

```
private static int result;
public static synchronized void incrementResult(int i){
    result += i;
}
```

Now we have to modify the `run()` method in the worker thread too:

```
public void run() {
    IntStream.rangeClosed(1, 250)
       .forEach(Demo04Synchronization::incrementResult);
}
```

The output now shows the same final number for every run:



```
demo_ThreadInterference:
Result=62750
```

The `synchronized` keyword tells JVM that only one thread at a time is allowed to enter this method. All the other threads will wait until the current visitor of the method exits from it.

The same effect could be achieved by adding the `synchronized` keyword to a block of code:

```
public static void incrementResult(int i){
    synchronized (Demo04Synchronization.class){
        result += i;
    }
}
```

The difference is that the block synchronization requires an object--a class object in the case of static property synchronization (as in our case) or any other object in the case of an instance property synchronization. Each object has an intrinsic lock or monitor lock, often referred to simply as a monitor. Once a thread acquires a lock on an object, no other thread can acquire it on the same object until the first thread releases the lock after normal exit from the locked code or if the code throws an exception.

In fact, in the case of a synchronized method, an object (the one to which the method belongs) is used for locking, too. It just happens behind the scene automatically and does not require the programmer to use an object's lock explicitly.

In case you do not have access to the `main` class code (as in the example earlier) you can keep the `result` property public and add a synchronized method to the worker thread (instead of the class as we have done):

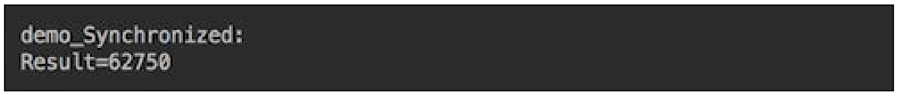
```
class MyRunnable05 implements Runnable {
    public synchronized void incrementResult(int i){
        Demo04Synchronization.result += i;
    }
```

```
        public void run() {
            IntStream.rangeClosed(1, 250)
                    .forEach(this::incrementResult);
        }
    }
```

In this case, the object of the `MyRunnable05` worker class provides its intrinsic lock by default. This means, you need to use the same object of the `MyRunnable05` class for all the threads:

```
    void demo_Synchronized(){
        System.out.println("demo_Synchronized: ");
        MyRunnable05 r1 = new MyRunnable05();
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r1);
        t1.start();
        t2.start();
        sleepMs(100);
        System.out.println("Result=" + result);
    }
```

The output of the preceding code is the same as before:



```
demo_Synchronized:
Result=62750
```

One can argue that this last implementation is preferable because it allocates the responsibility of the synchronization with the thread (and the author of its code) and not with the shared resource. This way the need for synchronization changes along with the thread implementation evolution, provided that the client code (that uses the same or different objects for the threads) can be changed as needed as well.

There is another possible concurrency issue that may happen in some operating systems. Depending on how the thread caching is implemented, a thread might preserve a local copy of the property `result` and not update it after another thread has changed its value. By adding the `volatile` keyword to the shared (between threads) property guarantees that its current value will be always read from the main memory, so each thread will see the updates done by the other threads. In our previous examples, we just set the `Demo04Synchronization` class property as `private static volatile int result`, add a synchronized `incrementResult()` method to the same class or to the thread and do not worry anymore about threads stepping on each other.

The described thread synchronization is usually sufficient for the mainstream application. But the higher performance and highly concurrent processing often require looking closer into the thread dump, which typically shows that method synchronization is more efficient than block synchronization. Naturally, it also depends on the size of the method and the block. Since all the other threads that try to access the synchronized method or block are going to stop execution until the current visitor of the method or block exits it, it is possible that despite the overhead a small synchronized block yields better performance than the big synchronized method.

For some applications, the behavior of the default intrinsic lock, which just blocks until the lock is released, maybe not well suited. If that is the case, consider using locks from the `java.util.concurrent.locks` package. The access control based on locks from that package has several differences if compared with using the default intrinsic lock. These differences may be advantageous for your application or provide the unnecessary complication, but it's important to know them, so you can make an informed decision:

- The synchronized fragment of code does not need to belong to one method; it can span several methods, delineated by the calls to the `lock()` and `unlock()` methods (invoked on the object that implements the `Lock` interface)
- While creating an object of the `Lock` interface called `ReentrantLock`, it is possible to pass into the constructor a `fair` flag that makes the lock able to grant an access to the longest-waiting thread first, which helps to avoid starvation (when the low priority thread never can get access to the lock)
- Allows a thread to test whether the lock is accessible before committing to be blocked
- Allows interrupting a thread waiting for the lock, so it does not remain blocked indefinitely
- You can implement the `Lock` interface yourself with whatever features you need for your application

A typical pattern of usage of the `Lock` interface looks like this:

```
Lock lock = ...;
...
    lock.lock();
    try {
        // the fragment that is synchronized
    } finally {
        lock.unlock();
    }
...
}
```

Notice the `finally` block. It is the way to guarantee that the `lock` is released eventually. Otherwise, the code inside the `try-catch` block can throw an exception and the lock is never released.

In addition to the `lock()` and `unlock()` methods, the `Lock` interface has the following methods:

- `lockInterruptibly()`: This acquires the lock unless the current thread is interrupted. Similar to the `lock()` method, this method blocks while waiting until the lock is acquired, in difference to the `lock()` method, if another thread interrupts the waiting thread, this method throws the `InterruptedException` exception
- `tryLock()`: This acquires the lock immediately if it is free at the time of invocation
- `tryLock(long time, TimeUnit unit)`: This acquires the lock if it is free within the given waiting time and the current thread has not been interrupted
- `newCondition()`: This returns a new `Condition` instance that is bound to this `Lock` instance, after acquiring the lock, the thread can release it (calling the `await()` method on the `Condition` object) until some other thread calls `signal()` or `signalAll()` on the same `Condition` object, it is also possible to specify the timeout period (by using an overloaded `await()` method), so the thread will resume after the timeout if there was no signal received, see the `Condition` API for more details

The scope of this book does not allow us to show all the possibilities for thread synchronization provided in the `java.util.concurrent.locks` package. It would take several lessons to describe all of them. But even from this short description, you can see that one would be hard pressed to find a synchronization problem that cannot be solved using the `java.util.concurrent.locks` package.

The synchronization of a method or block of code makes sense when several lines of code have to be isolated as an atomic (all or nothing) operation. But in the case of a simple assignment to a variable or increment/decrement of a number (as in our earlier examples), there is a much better way to synchronize this operation by using classes from the `java.util.concurrent.atomic` package that support lock-free thread-safe programming on a single variable. The variety of classes covers all the numbers and even arrays and reference types such as `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicReference`, and `AtomicReferenceArray`.

There are 16 classes in total. Depending on the value type, each of them allows a full imaginable range of operations, that is, `set()`, `get()`, `addAndGet()`, `compareAndSet()`, `incrementAndGet()`, `decrementAndGet()`, and many others. Each operation is implemented much more efficiently than the same operations implemented with the `synchronized` keyword. And there is no need for the `volatile` keyword because it uses it under the hood.

If the concurrently accessed resource is a collection, the `java.util.concurrent` package offers a variety of thread-safe implementations that perform better than synchronized `HashMap`, `Hashtable`, `HashSet`, `Vector`, and `ArrayList` (if we compare the corresponding `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `CopyOnWriteHashSet`). The traditional synchronized collections lock the whole collection while concurrent collections use such advanced techniques such as lock stripping to achieve thread safety. The concurrent collections especially shine with more reading and fewer updates and they are much more scalable than synchronized collections. But if the size of your shared collection is small and writes dominate, the advantage of concurrent collections is not as obvious.

Tuning JVM

Each pyramid building, as any big project, goes through the same life cycle of design, planning, execution, and delivery. And throughout each of these phases, a continuous tuning is going on, a complex project is called so for a reason. A software system is not different in this respect. We design, plan and build it, then change and tune continuously. If we are lucky, then the new changes do not go too far back to the initial stages and do not require changing the design. To hedge against such drastic steps, we use prototypes (if the waterfall model is used) or iterative delivery (if the agile process is adopted) for early detection of possible problems. Like young parents, we are always on alert, monitoring the progress of our child, day and night.

As we mentioned already in one of the previous sections, there are several diagnostic tools that come with each JDK 9 installation or can be used in addition to them for monitoring your Java application. The full list of these tools (and the recommendations how to create a custom tool, if needed) can be found in official Java SE documentation on the Oracle site: `https://docs.oracle.com/javase/9/troubleshoot/diagnostic-tools.htm`.

Using these tools one identifies the bottleneck of the application and addresses it either programmatically or by tuning the JVM itself or both. The biggest gain usually comes with the good design decisions and from using certain programming techniques and frameworks, some of which we have described in other sections. In this section, we are going to look at the options available after all possible code changes are applied or when changing code is not an option, so all we can do is to tune JVM itself.

The goal of the effort depends on the results of the application profiling and the nonfunctional requirements for:

- Latency, or how responsive the application is to the input
- Throughput, or how much work the application is doing in a given unit of time
- Memory footprint, or how much memory the application requires

The improvements in one of them often are possible only at the expense of the one or both of the others. The decrease in the memory consumption may bring down the throughput and latency, while the decrease in latency typically can be achieved only via the increase in memory footprint unless you can bring in faster CPUs thus improving all three characteristics.

Application profiling may show that one particular operation keeps allocating a lot of memory in the loop. If you have an access to the code, you can try to optimize this section of the code and thus ease the pressure on JVM. Alternatively, it may show that there is an I/O or another interaction with a low device is involved, and there is nothing you can do in the code to improve it.

Defining the goal of the application and JVM tuning requires establishing metrics. For example, it is well known already that the traditional measure of latency as the average response time hides more than it reveals about the performance. The better latency metrics would be the maximum response time in conjunction with 99% best response time. For throughput, a good metrics would be the number of transactions per a unit of time. Often the inverse of this metrics (time per transaction) closely reflects latency. For the memory footprint, the maximum allocated memory (under the load) allows for the hardware planning and setting guards against the dreaded `OutOfMemoryError` exception. Avoiding full (stop-the-world) garbage collection cycle would be ideal. In practice, though, it would be good enough if **Full GC** happens not often, does not visibly affect the performance and ends up with approximately the same heap size after several cycles.

Unfortunately, such simplicity of the requirements does happen in practice. Real life brings more questions all the time as follows:

- Can the target latency (response time) be ever exceeded?
- If yes, how often and by how much?
- How long can the period of the poor response time last?
- Who/what measures the latency in production?
- Is the target performance the peak performance?

- What is the expected peak load?
- How long is the expected peak load going to last?

Only after all these and similar questions are answered and the metrics (that reflect the nonfunctional requirements) are established, we can start tweaking the code, running it and profiling again and again, then tweaking the code and repeating the cycle. This activity has to consume most of the efforts because tuning of the JVM itself can bring only the fraction of the performance improvements by comparison with the performance gained by the code changes.

Nevertheless, several passes of the JVM tuning must happen early in order to avoid wasting of the efforts and trying to force the code in the not well-configured environment. The JVM configuration has to be as generous as possible for the code to take advantage of all the available resources.

First of all, select garbage collector from the four that JVM 9 supports, which are as follows:

- **Serial collector**: This uses a single thread to perform all the garbage collection work
- **Parallel collector**: This uses multiple threads to speed up garbage collection
- **Concurrent Mark Sweep (CMS) collector**: This uses shorter garbage collection pauses at the expense of taking more of the processor time
- **Garbage-First (G1) collector**: This is intended for multiprocessor machines with a large memory, but meets garbage collection pause-time goals with high probability, while achieving high throughput.

The official Oracle documentation (`https://docs.oracle.com/javase/9/gctuning/available-collectors.htm`) provides the following initial guidelines for the garbage collection selection:

- If the application has a small dataset (up to approximately 100 MB), then select the serial collector with the `-XX:+UseSerialGC` option
- If the application will be run on a single processor and there are no pause-time requirements, then select the serial collector with the `-XX:+UseSerialGC` option
- If (a) peak application performance is the first priority and (b) there are no pause-time requirements or pauses of one second or longer are acceptable, then let the VM select the collector or select the parallel collector with `-XX:+UseParallelGC`

- If the response time is more important than the overall throughput and garbage collection pauses must be kept shorter than approximately one second, then select a concurrent collector with `-XX:+UseG1GC or -XX:+UseConcMarkSweepGC`

But if you do not have particular preferences yet, let the JVM select garbage collector until you learn more about your application's needs. In JDK 9, the G1 is selected by default on certain platforms, and it is a good start if the hardware you use has enough resources.

Oracle also recommends using G1 with its default settings, then later playing with a different pause-time goal using the `-XX:MaxGCPauseMillis` option and maximum Java heap size using the `-Xmx` option. Increasing either the pause-time goal or the heap size typically leads to a higher throughput. The latency is affected by the change of the pause-time goal too.

While tuning the GC, it is beneficial to keep the `-Xlog:gc*=debug` logging option. It provides many useful details about garbage collection activity. The first goal of JVM tuning is to decrease the number of full heap GC cycles (Full GC) because they are very resource consuming and thus may slow down the application. It is caused by too high occupancy of the old generation area. In the log, it is identified by the words `Pause Full (Allocation Failure)`. The following are the possible steps to reduce chances of Full GC:

- Bring up the size of the heap using `-Xmx`. But make sure it does not exceed the physical size of RAM. Better yet, leave some RAM space for other applications.
- Increase the number of concurrent marking threads explicitly using `-XX:ConcGCThreads`.
- If the humongous objects take too much of the heap (watch for **gc+heap=info** logging that shows the number next to humongous regions) try to increase the region size using `-XX: G1HeapRegionSize`.
- Watch the GC log and modify the code so that almost all the objects created by your application are not moved beyond the young generation (dying young).
- Add or change one option at a time, so you can understand the causes of the change in the JVM's behavior clearly.

These few steps will help you go and create a trial-and-error cycle that will bring you a better understanding of the platform you are using, the needs of your application, and the sensitivity of the JVM and the selected GC to different options. Equipped with this knowledge, you will then be able to meet the nonfunctional performance requirements whether by changing the code, tuning the JVM, or reconfiguring the hardware.

Reactive Programming

After several false starts and a few disastrous disruptions, followed by heroic recoveries, the process of pyramid building took shape and ancient builders were able to complete a few projects. The final shape sometimes did not look exactly as envisioned (the first pyramids have ended up bent), but, nevertheless, the pyramids still decorate the desert today. The experience was passed from generation to generation, and the design and the process were tuned well enough to produce something magnificent and pleasant to look at more than 4,000 years later.

The software practices also change over time, albeit we have had only some 70 years since Mr. Turing wrote the first modern program. In the beginning, when there were only a handful of programmers in the world, a computer program used to be a continuous list of instructions. Functional programming (pushing a function around like a first-class citizen) was introduced very early too but has not become a mainstream. Instead, the **GOTO** instruction allowed you to roll code in a spaghetti bowl. Structural programming followed, then object-oriented programming, with functional programming moving along and even thriving in certain areas. Asynchronous processing of the events generated by the pressed keys became routine for many programmers. JavaScript tried to use all of the best practices and gained a lot of power, even if at the expense of programmers' frustration during the debugging (fun) phase. Finally, with thread pools and lambda expressions being part of JDK SE, adding reactive streams API to JDK 9 made Java part of the family that allows reactive programming with asynchronous data streams.

To be fair, we were able to process data asynchronously even without this new API--by spinning worker threads and using thread pools and callables (as we described in the previous sections) or by passing the callbacks (even if lost once in a while in the maze of the one who-calls-whom). But, after writing such a code a few times, one notices that most of such code is just a plumbing that can be wrapped inside a framework that can significantly simplify asynchronous processing. That's how the Reactive Streams initiative (`http://www.reactive-streams.org`) came to be created and the scope of the effort is defined as follows:

The scope of Reactive Streams is to find a minimal set of interfaces, methods and protocols that will describe the necessary operations and entities to achieve the goal--asynchronous streams of data with non-blocking back pressure.

The term **non-blocking back pressure** is an important one because it identifies one of the problems of the existed asynchronous processing--coordination of the speed rate of the incoming data with the ability of the system to process them without the need of stopping (blocking) the data input. The solution would still include some back pressure by informing the source that the consumer has difficulty in keeping up with the input, but the new framework should react to the change of the rate of the incoming data in a more flexible manner than just blocking the flow, thus the name **reactive**.

The Reactive Streams API consists of the five interfaces included in the class which are `java.util.concurrent.Flow`, `Publisher`, `Subscriber`, `Subscription`, and `Processor`:

```
@FunctionalInterface
public static interface Flow.Publisher<T> {
  public void subscribe(Flow.Subscriber<? super T> subscriber);
}

public static interface Flow.Subscriber<T> {
  public void onSubscribe(Flow.Subscription subscription);
  public void onNext(T item);
  public void onError(Throwable throwable);
  public void onComplete();
}

public static interface Flow.Subscription {
  public void request(long numberOfItems);
  public void cancel();
}

public static interface Flow.Processor<T,R>
               extends Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

A `Flow.Subscriber` object becomes a subscriber of the data produced by the object of `Flow.Publisher` after the object of `Flow.Subscriber` is passed as a parameter into the `subscribe()` method. The publisher (object of `Flow.Publisher`) calls the subscriber's `onSubscribe()` method and passes as a parameter a `Flow.Subsctiption` object. Now, the subscriber can request `numberOffItems` of data from the publisher by calling the subscription's `request()` method. That is the way to implement the pull model when a subscriber decides when to request another item for processing. The subscriber can unsubscribe from the publisher services by calling the `cancel()` subscription method.

In return (or without any request, if the implementer has decided to do so, that would be a push model), the publisher can pass to the subscriber a new item by calling the subscriber's `onNext()` method. The publisher can also tell the subscriber that the item production has encountered a problem (by calling the subscriber's `onError()` method) or that no more data will be coming (by calling the subscriber's `onComplete()` method).

The `Flow.Processor` interface describes an entity that can act as both a subscriber and a publisher. It allows creating chains (pipelines) of such processors, so a subscriber can receive an item from a publisher, tweak it, and then pass the result to the next subscriber.

This is the minimal set of interfaces the Reactive Streams initiative has defined (and it is a part of JDK 9 now) in support of the asynchronous data streams with non-blocking back pressure. As you can see, it allows the subscriber and publisher to talk to each other and coordinate, if need be, the rate of incoming data, thus making possible a variety of solutions for the back pressure problem we discussed in the beginning.

There are many ways to implement these interfaces. Currently, in JDK 9, there is only one example of implementation of one of the interfaces--the `SubmissionPublisher` class implements `Flow.Publisher`. But several other libraries already exist that implemented Reactive Streams API: RxJava, Reactor, Akka Streams, and Vert.x are among the most known. We will use RxJava 2.1.3 in our examples. You can find the RxJava 2.x API on `http://reactivex.io` under the name ReactiveX, which stands for Reactive Extension.

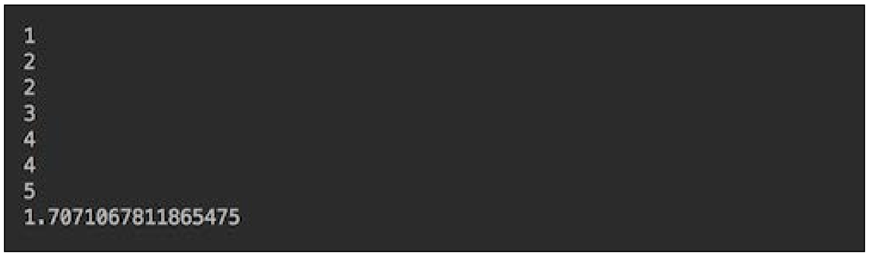
While doing that, we would also like to address the difference between the streams of the `java.util.stream` package and Reactive Streams (as implemented in RxJava, for example). It is possible to write very similar code using any of the streams. Let's look at an example. Here is a program that iterates over five integers, selects even numbers only (2 and 4), transforms each of them (takes a square root of each of the selected numbers) and then calculates an average of the two square roots. It is based on the traditional `for` loop.

Let's start with the similarity. It is possible to implement the same functionality using any of the streams. For example, here is a method that iterates over five integers, selects even numbers only (2 and 4, in this case), transforms each of them (takes a square root of each of the even numbers) and then calculates an average of the two square roots. It is based on the traditional `for` loop:

```
void demo_ForLoop(){
    List<Double> r = new ArrayList<>();
    for(int i = 1; i < 6; i++){
        System.out.println(i);
```

```
            if(i%2 == 0){
                System.out.println(i);
                r.add(doSomething(i));
            }
        }
        double sum = 0d;
        for(double d: r){ sum += d; }
        System.out.println(sum / r.size());
    }
    static double doSomething(int i){
        return Math.sqrt(1.*i);
    }
```

If we run this program, the result will be as follows:



```
1
2
2
3
4
4
5
1.7071067811865475
```

The same functionality (with the same output) can be implemented using the package `java.util.stream` as follows:

```
    void demo_Stream(){
        double a = IntStream.rangeClosed(1, 5)
            .peek(System.out::println)
            .filter(i -> i%2 == 0)
            .peek(System.out::println)
            .mapToDouble(i -> doSomething(i))
            .average().getAsDouble();
        System.out.println(a);
    }
```

The same functionality can be implemented with RxJava:

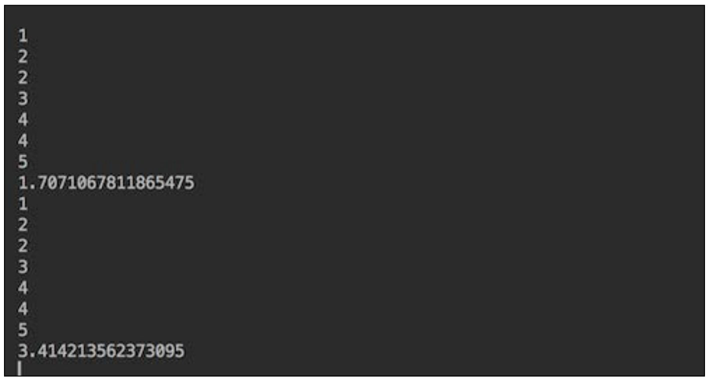
```
    void demo_Observable1(){
        Observable.just(1,2,3,4,5)
            .doOnNext(System.out::println)
            .filter(i -> i%2 == 0)
            .doOnNext(System.out::println)
            .map(i -> doSomething(i))
            .reduce((r, d) -> r + d)
            .map(r -> r / 2)
            .subscribe(System.out::println);
}
```

RxJava is based on the `Observable` object (which plays the role of `Publisher`) and `Observer` that subscribes to the `Observable` and waits for data to be emitted. Each item of the emitted data (on the way from the `Observable` to the `Observer`) can be processed by the operations chained in a fluent style (see the previous code). Each operation takes a lambda expression. The operation functionality is obvious from its name.

Despite being able to behave similarly to the streams, an `Observable` has significantly different capabilities. For example, a stream, once closed, cannot be reopened, while an `Observable` can be reused. Here is an example:

```
void demo_Observable2(){
    Observable<Double> observable = Observable
            .just(1,2,3,4,5)
            .doOnNext(System.out::println)
            .filter(i -> i%2 == 0)
            .doOnNext(System.out::println)
            .map(Demo05Reactive::doSomething);

    observable
            .reduce((r, d) -> r + d)
            .map(r -> r / 2)
            .subscribe(System.out::println);

    observable
            .reduce((r, d) -> r + d)
            .subscribe(System.out::println);
}
```

In the previous code, we use `Observable` twice--for average value calculation and for the summing all the square roots of the even numbers. The output is as shown in the following screenshot:

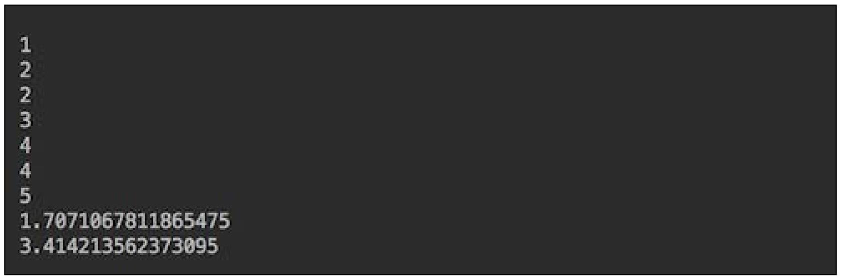


```
1
2
2
3
4
4
5
1.7071067811865475
1
2
2
3
4
4
5
3.414213562373095
```

If we do not want `Observable` to run twice, we can cache its data, by adding the `.cache()` operation:

```
void demo_Observable2(){
    Observable<Double> observable = Observable
            .just(1,2,3,4,5)
            .doOnNext(System.out::println)
            .filter(i -> i%2 == 0)
            .doOnNext(System.out::println)
            .map(Demo05Reactive::doSomething)
            .cache();

    observable
            .reduce((r, d) -> r + d)
            .map(r -> r / 2)
            .subscribe(System.out::println);

    observable
            .reduce((r, d) -> r + d)
            .subscribe(System.out::println);
}
```

The result of the previous code is as follows:



```
1
2
2
3
4
4
5
1.7071067811865475
3.414213562373095
```

You can see that the second usage of the same `Observable` took advantage of the cached data, thus allowing for better performance.

Another `Observable` advantage is that the exception can be caught by `Observer`:

```
subscribe(v -> System.out.println("Result=" + v),
        e -> {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        },
        () -> System.out.println("All the data processed"));
```

The `subscribe()` method is overloaded and allows to pass in one, two, or three functions:

- The first is to be used in case of success
- The second is to be used in case of an exception
- The third is to be called after all the data is processed

The `Observable` model also allows more control over multithreaded processing. Using `.parallel()` in the streams does not allow you to specify the thread pool to be used. But, in RxJava, you can set the type of pool you prefer using the method `subscribeOn()` in `Observable`:

```
observable.subscribeOn(Schedulers.io())
        .subscribe(System.out::println);
```

The `subscribeOn()` method tells `Observable` on which thread to put the data. The `Schedulers` class has methods that generate thread pools dealing mostly with I/O operations (as in our example), or heavy on computation (method `computation()`), or creating a new thread for each unit of work (method `newThread()`), and several others, including passing in a custom thread pool (method `from(Executor executor)`).

The format of this book does not allow us to describe all the richness of RxJava API and other Reactive Streams implementations. Their main thrust is reflected in Reactive Manifesto (`http://www.reactivemanifesto.org/`) that describes Reactive Systems as a new generation of high performing software solutions. Built on asynchronous message-driven processes and Reactive Streams, such systems are able to demonstrate the qualities declared in the Reactive Manifesto:

- **Elasticity**: This has the ability to expand and contract as needed based on the load
- **Better responsiveness**: Here, the processing can be parallelized using asynchronous calls
- **Resilience**: Here, the system is broken into multiple (loosely coupled via messages) components, thus facilitating flexible replication, containment, and isolation

Writing code for Reactive Systems using Reactive Streams for implementing the previously mentioned qualities constitutes reactive programming. The typical application of such systems today is microservices, which is described in the next lesson.

Summary

In this lesson, we have discussed the ways to improve Java application performance by using multithreading. We described how to decrease an overhead of creating the threads using thread pools and various types of such pools suited for different processing requirements. We also brought up the considerations used for selecting the pool size and how to synchronize threads so that they do not interfere with each other and yield the best performance results. We pointed out that every decision on the performance improvements has to be made and tested through direct monitoring of the application, and we discussed the possible options for such monitoring programmatically and using various external tools. The final step, the JVM tuning, can be done via Java tool flags that we listed and commented in the corresponding section. Yet more gains in Java application performance might be achieved by adopting the concept of reactive programming, which we presented as the strong contender among most effective moves toward highly scalable and highly performing Java applications.

In the next lesson, we will talk about adding more workers by splitting the application into several microservices, each deployed independently and each using multiple threads and reactive programming for better performance, response, scalability, and fault-tolerance.

Assessments

1. Name the method that calculates the average square root of the first 99,999 integers and assigns the result to a property that can be accessed anytime.
2. Which of the following methods creates a thread pool of a fixed size that can schedule commands to run after a given delay, or to execute periodically:
    1. `newscheduledThreadPool()`
    2. `newWorkStealingThreadPool()`
    3. `newSingleThreadScheduledExecutor()`
    4. `newFixedThreadPool()`
3. State whether True or False: One can take advantage of the `Runnable` interface being a functional interface and pass the necessary processing function into a new thread as a lambda expression.
4. After the ___________ method is called, no more worker threads can be added to the pool.
    1. `shutdownNow()`
    2. `shutdown()`
    3. `isShutdown()`
    4. `isShutdownComplete()`
5. ________ is based on the `Observable` object, which plays the role of a Publisher.

4 Microservices

As long as we kept talking about the designing, implementation, and tuning of one process, we were able to keep illustrating it with vivid images (albeit in our imagination only) of pyramid building. Multiple thread management, based on the democratic principle of equality between thread pool members, had also a sense of centralized planning and supervision. Different priorities were assigned to threads programmatically, hardcoded (for most cases) after thoughtful consideration by the programmer in accordance with the expected load, and adjusted after monitoring. The upper limits of the available resources were fixed, although they could be increased after, again, a relatively big centralized decision.

Such systems had great success and still constitute the majority of the web applications currently deployed to production. Many of them are monoliths, sealed inside a single `.ear` or `.war` file. This works fine for relatively small applications and a corresponding team size that supports them. They are easy (if the code is well structured) to maintain, build, and if the production load is not very high, they can be easily deployed. If the business does not grow or has little impact on the company's internet presence, they continue to do the job and will do so probably for the foreseeable future. Many service providers are eager to host such websites by charging a small fee and relieving the website owner of the technical worries of production maintenance not directly related to the business. But that is not the case for everybody.

The higher the load, the more difficult and expensive the scaling becomes unless the code and the overall architecture is restructured in order to become more flexible and resilient to the growing load. This lesson describes the solution many leaders of the industry have adopted while addressing the issue and the motivation behind it.

The particular aspects of the microservices we are going to discuss in this lesson include the following:

- The motivation for the microservices rising
- The frameworks that were developed recently in support of microservices
- The process of microservices development with practical examples, including the considerations and decision-making process during microservices building
- Pros and cons of the three main deployment methods such as container-less, self-contained, and in-container

Why Microservices?

Some businesses have a higher demand for the deployment plan because of the need to keep up with the bigger volume of traffic. The natural answer to this challenge would be and was to add servers with the same `.ear` or `.war` file deployed and join all the servers into a cluster. So, one failed server could be automatically replaced with another one from the cluster, and the site user would never experience disconnect of the service. The database that backed all the clustered servers could be clustered too. A connection to each of the clusters went through a load balancer, making sure that none of the cluster members worked more than the others.

The web server and database clustering help but only to a degree, because as the code base grows, its structure can create one or several bottlenecks unless such and similar issues are addressed with a scalable design. One of the ways to do it is to split the code into tiers: front end (or web tier), middle tier (or app tier) and back end (or backend tier). Then, again, each tier can be deployed independently (if the protocol between tiers has not changed) and in its own cluster of servers, as each tier can grow horizontally as needed independently of other tiers. Such a solution provides more flexibility for scaling up, but makes the deployment plan more complex, especially if the new code introduces breaking changes. One of the approaches is to create a second cluster that will host a new code, then take the servers one by one from the old cluster, deploy the new code, and put them in the new cluster. The new cluster would be turned on as soon as at least one server in each tier has the new code. This approach worked fine for the web and app tiers but was more complex for the backend, which once in a while required data migration and similar joyful exercises. Add to it unexpected outages in the middle of the deployment caused by human errors, defects in the code, pure accidents, or some combination of all the earlier mentioned (one time, for example, an electric power cable was cut by an excavator in the nearby construction site), and it is easy to understand why very few people love a deployment of a major release to production.

Programmers, being by nature problem solvers, tried to prevent the earlier scenario as best as they could by writing defensive code, deprecating instead of changing, testing, and so on. One of the approaches was to break the application into more independently deployable parts with the hope of avoiding deploying everything at the same time. They called these independent units **services**, and **Service-Oriented Architecture** (**SOA**) was born.

Unfortunately, in many companies, the natural growth of the code base was not adjusted to the new challenges in a timely manner. Like the frog that was eventually boiled in a slowly heated pot of water, they never had time to jump out of the hot spot by changing the design. It was always cheaper to add another feature to the blob of the existing functionality than redesign the whole app. Business metrics of the time-to-market and keeping the bottom line in the black always were and will remain the main criterion for the decision making, until the poorly structured source code eventually stops working, pulling down all the business transactions with it or, if the company is lucky, allows them to weather the storm and shows the importance of the investment in the redesign.

As a result of all that, some lucky companies remained in the business with their monolithic application still running as expected (maybe not for long, but who knows), some went out of business, some learned from their mistakes and progressed into the brave world of the new challenges, and others learned from their mistakes and designed their systems to be SOA upfront.

It is interesting to observe similar tendencies in the social sphere. Society moved from the strong centralized governments to more loosely coupled confederations of semi-independent states tied together by the mutually beneficial economic and cultural exchange.

Unfortunately, maintaining such a loose structure comes with a price. Each participant has to be more responsible in maintaining the contract (social, in the case of a society, and API, in the case of the software) not only formally but also in spirit. Otherwise, for example, the data flowing from a new version of one component, although correct by type, might be unacceptable to another component by value (too big or too small). Maintaining a cross-team understanding and overlapping of responsibility requires constant vigilance in keeping the culture alive and enlightening. Encouraging innovation and risk taking, which can lead to a business breakthrough, contradict the protecting tendencies for stability and risk aversion coming from the same business people.

Moving from monolithic single-team development to multiple teams and an independent components-based system requires an effort on all levels of the enterprise. What do you mean by **No more Quality Assurance Department**? Who then will care about the professional growth of the testers? And what about the IT group? What do you mean by **The developers are going to support production**? Such changes affect human lives and are not easy to implement. That's why SOA architecture is not just a software principle. It affects everybody in the company.

Meanwhile, the industry leaders, who have managed to grow beyond anything we could imagine just a decade ago, were forced to solve even more daunting problems and came back to the software community with their solutions. And that is where our analogy with the pyramid building does not work anymore. Because the new challenge is not just to build something so big that was never built before but also to do it quickly not in a matter of years, but in a few weeks and even days. And the result has to last not for a thousand years but has to be able to evolve constantly and be flexible enough to adapt to new, unexpected requirements in real time. If only one aspect of the functionality has changed, we should be able to redeploy only this one service. If the demand for any service grows, we should be able to scale only along this one service and release resources when the demand drops.

To avoid big deployments with all hands on deck and to come closer to the continuous deployment (which decreases time-to-market and is thus supported by business), the functionality continued to split into smaller chunks of services. In response to the demand, more sophisticated and robust cloud environments, deployment tools (including containers and container orchestration), and monitoring systems supported this move. The reactive streams, described in the previous lesson, started to develop even before the Reactive Manifesto came out and plugged a snag into the stack of modern frameworks.

Splitting an application into independent deployment units brought several not quite expected benefits that have increased the motivation for plowing ahead. The physical isolation of services allows more flexibility in choosing a programming language and platform of implementation. It helps not only to select technology that is the best for the job but also to hire people able to implement it, not being bound by a certain technological stack of the company. It also helped the recruiters to spread the net wider and use smaller cells for bringing in new talent, which is not a small advantage with a limited number of available specialists and the unlimited demand of the fast-growing data processing industry.

Also, such architecture enforced a discussion and explicit definition of the interfaces between smaller parts of the complex system, thus creating a solid foundation for further growth and tuning of the processing sophistication.

And that is how microservices came into the picture and were put to work by giants of traffic such as Netflix, Google, Twitter, eBay, Amazon, and Uber. Now, let's talk about the results of this effort and the lessons learned.

Building Microservices

Before diving into the building process, let's revisit the characteristics a chunk of code has to possess in order to be qualified as a microservice. We will do it in no particular order:

- The size of the source code of one microservice should be smaller to that of an SOA, and one development team should be able to support several of them.
- It has to be deployed independently of other services.
- Each has to have its own database (or schema or set of tables), although this statement is still under debate, especially in cases when several services modify the same data set or the inter-dependent data sets; if the same team owns all of the related services, it is easier to accomplish. Otherwise, there are several possible strategies we will discuss later.
- It has to be stateless and idempotent. If one instance of the service has failed, another should be able to accomplish what was expected from the service.
- It should provide a way to check its **health**, meaning that the service is up and running and ready to do the job.

Sharing resources has to be considered during the design, development, and, after deployment, monitored for validation of the assumptions. In the previous lesson, we talked about threads synchronization. You could see that this problem was not easy to solve, and we have presented several possible ways to do it. Similar approaches can be applied toward microservices. Although they are run in different processes, they can communicate to each other if need be, so they can coordinate and synchronize their actions.

Special care has to be taken during modification of the same persistent data whether shared across databases, schemas, or tables within the same schema. If an eventual consistency is acceptable (which is often the case for larger sets of data, used for statistical purposes, for example) then no special measures are necessary. However, the need for transactional integrity poses a more difficult problem.

One way to support a transaction across several microservices is to create a service that would play the role of a **Distributed Transaction Manager** (**DTM**). Other services that need coordination would pass to it the new modified values. The DTM service could keep the concurrently modified data temporarily in a database table and would move it into the main table(s) in one transaction after all the data is ready (and consistent).

If the time to access the data is an issue or you need to protect the database from an excessive number of concurrent connections, dedicating a database to some services may be an answer. Alternatively, if you would like to try another option, memory cache could be the way to go. Adding a service that provides access to the cache (and updates it as needed) increases isolation from the services that use it, but requires (sometimes difficult) synchronization between the peers that are managing the same cache too.

After considering all the options and possible solutions for data sharing, it is often helpful to revisit the idea of creating its own database (or schema) for each microservice. One may discover that the effort of the data isolation (and subsequent synchronization on the database level) does not look as daunting as before if compared with the effort to synchronize the data dynamically.

That said, let's look over the field of the frameworks for microservices implementation. One can definitely write the microservices from scratch, but before doing that, it is always worth looking at what is out there already, even if to find eventually that nothing fits your particular needs.

There are more than a dozen frameworks that are currently used for building microservices. Two most popular are Spring Boot (`https://projects.spring.io/spring-boot/`) and raw J2EE. The J2EE community founded the initiative MicroProfile (`https://microprofile.io/`) with a declared goal of **Optimizing Enterprise Java** for a microservices architecture. KumuluzEE (`https://ee.kumuluz.com/`) is a lightweight open-source microservice framework coplined with MicroProfile.

The list of some other frameworks include the following (in alphabetical order):

- **Akka**: This is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala (`akka.io`)
- **Bootique**: This is a minimally opinionated framework for runnable Java apps (`bootique.io`)
- **Dropwizard**: This is a Java framework for developing ops-friendly, high-performance, RESTful web services (`www.dropwizard.io`)
- **Jodd**: This is a set of Java microframeworks, tools, and utilities, under 1.7 MB (`jodd.org`)

- **Lightbend Lagom**: This is an opinionated microservice framework built on Akka and Play (`www.lightbend.com`)
- **Ninja**: This is a full stack web framework for Java (`www.ninjaframework.org`)
- **Spotify Apollo**: This is a set of Java libraries used at Spotify for writing microservices (`spotify.github.io/apollo`)
- **Vert.x**: This is a toolkit for building reactive applications on the JVM (`vertx.io`)

All frameworks support HTTP/JSON communication between microservices; some of them also have an additional way to send messages. If not the latter, any lightweight messaging system can be used. We mentioned it here because, as you may recall, message-driven asynchronous processing is a foundation for elasticity, responsiveness, and resilience of a reactive system composed of microservices.

To demonstrate the process of microservices building, we will use Vert.x, an event-driven, non-blocking, lightweight, and polyglot toolkit (components can be written in Java, JavaScript, Groovy, Ruby, Scala, Kotlin, and Ceylon). It supports an asynchronous programming model and a distributed event bus that reaches even into in-browser JavaScript (thus allowing the creation of real-time web applications).

One starts using Vert.x by creating a `Verticle` class that implements the interface `io.vertx.core.Verticle`:

```
package io.vertx.core;
public interface Verticle {
  Vertx getVertx();
  void init(Vertx vertx, Context context);
  void start(Future<Void> future) throws Exception;
  void stop(Future<Void> future) throws Exception;
}
```

The method names previously mentioned are self-explanatory. The method `getVertex()` provides access to the `Vertx` object the entry point into the Vert.x Core API. It provides access to the following functionality necessary for the microservices building:

- Creating TCP and HTTP clients and servers
- Creating DNS clients
- Creating Datagram sockets
- Creating periodic services
- Providing access to the event bus and file system API
- Providing access to the shared data API
- Deploying and undeploying verticles

Using this Vertx object, various verticles can be deployed, which talk to each other, receive an external request, and process and store data as any other Java application, thus forming a system of microservices. Using RxJava implementation from the package `io.vertx.rxjava`, we will show how one can create a reactive system of microservices.

A verticle is a building block in Vert.x world. It can easily be created by extending the `io.vertx.rxjava.core.AbstractVerticle` class:

```
package io.vertx.rxjava.core;
import io.vertx.core.Context;
import io.vertx.core.Vertx;
public class AbstractVerticle
               extends io.vertx.core.AbstractVerticle {
  protected io.vertx.rxjava.core.Vertx vertx;
  public void init(Vertx vertx, Context context) {
     super.init(vertx, context);
     this.vertx = new io.vertx.rxjava.core.Vertx(vertx);
  }
}
```

The earlier mentioned class, in turn, extends `io.vertx.core.AbstractVerticle`:

```
package io.vertx.core;
import io.vertx.core.json.JsonObject;
import java.util.List;
public abstract class AbstractVerticle
                                implements Verticle {
    protected Vertx vertx;
    protected Context context;
    public Vertx getVertx() { return vertx; }
    public void init(Vertx vertx, Context context) {
        this.vertx = vertx;
        this.context = context;
    }
    public String deploymentID() {
        return context.deploymentID();
    }
    public JsonObject config() {
        return context.config();
    }
    public List<String> processArgs() {
        return context.processArgs();
    }
    public void start(Future<Void> startFuture)
```

```
                            throws Exception {
            start();
            startFuture.complete();
        }
        public void stop(Future<Void> stopFuture)
                            throws Exception {
            stop();
            stopFuture.complete();
        }
        public void start() throws Exception {}
        public void stop() throws Exception {}

    }
```

A verticle can be created by extending the class `io.vertx.core.AbstractVerticle`, too. However, we will write reactive microservices, so we will extend its rx-fied version, `io.vertx.rxjava.core.AbstractVerticle`.

To use Vert.x and run the provided example, all you need to do is to add the following dependencies:

```
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-web</artifactId>
    <version>${vertx.version}</version>
</dependency>

<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-rx-java</artifactId>
    <version>${vertx.version}</version>
</dependency>
```

Other Vert.x functionality can be added as needed by including other Maven dependencies.

What makes `Vert.x Verticle` reactive is the underlying implementation of an event loop (a thread) that receives an event and delivers it a `Handler` (we will show how to write the code for it). When a `Handler` gets the result, the event loop invokes the callback.

> As you see, it is important not to write a code that blocks the event loop, thus the Vert.x golden rule: don't block the event loop.

If not blocked, the event loop works very quickly and delivers a huge number of events in a short period of time. This is called the reactor pattern (`https://en.wikipedia.org/wiki/Reactor_pattern`). Such an event-driven non-blocking programming model is a very good fit for reactive microservices. For certain types of code that are blocking by nature (JDBC calls and long computations are good examples) a worker verticle can be executed asynchronously (not by the event loop, but by a separate thread using the method `vertx.executeBlocking()`), which keeps the golden rule intact.

Let's look at a few examples. Here is a `Verticle` class that works as an HTTP server:

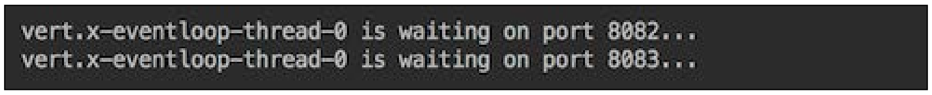
```
import io.vertx.rxjava.core.http.HttpServer;
import io.vertx.rxjava.core.AbstractVerticle;

public class Server extends AbstractVerticle{
  private int port;
  public Server(int port) {
    this.port = port;
  }
  public void start() throws Exception {
    HttpServer server = vertx.createHttpServer();
    server.requestStream().toObservable()
        .subscribe(request -> request.response()
        .end("Hello from " +
           Thread.currentThread().getName() +
                         " on port " + port + "!\n\n")
        );
    server.rxListen(port).subscribe();
    System.out.println(Thread.currentThread().getName()
              + " is waiting on port " + port + "...");
  }
}
```

In the previous code, the server is created, and the stream of data from a possible request is wrapped into an `Observable`. We then subscribed to the data coming from the `Observable` and passed in a function (a request handler) that will process the request and generate a necessary response. We also told the server which port to listen. Using this `Verticle`, we can deploy several instances of an HTTP server listening on different ports. Here is an example:

```
import io.vertx.rxjava.core.RxHelper;
import static io.vertx.rxjava.core.Vertx.vertx;
public class Demo01Microservices {
  public static void main(String... args) {
    RxHelper.deployVerticle(vertx(), new Server(8082));
```

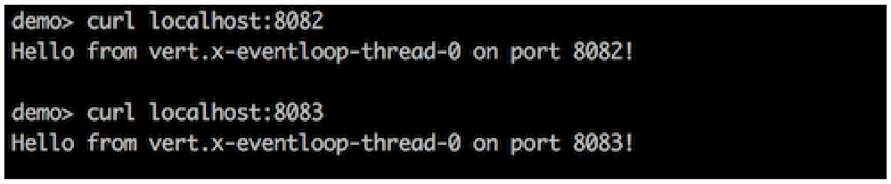
```
      RxHelper.deployVerticle(vertx(), new Server(8083));
    }
  }
```

If we run this application, the output would be as follows:



```
vert.x-eventloop-thread-0 is waiting on port 8082...
vert.x-eventloop-thread-0 is waiting on port 8083...
```

As you can see, the same thread is listening on both ports. If we now place a request to each of the running servers, we will get the response we have hardcoded:



```
demo> curl localhost:8082
Hello from vert.x-eventloop-thread-0 on port 8082!

demo> curl localhost:8083
Hello from vert.x-eventloop-thread-0 on port 8083!
```

We ran our examples from the `main()` method. A plugin `maven-shade-plugin` allows you to specify which verticle you would like to be the starting point of your application. Here is an example from `http://vertx.io/blog/my-first-vert-x-3-application`:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.3</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer
            implementation="org.apache.maven.plugins.shade.resource.
ManifestResourceTransformer">
            <manifestEntries>
              <Main-Class>io.vertx.core.Starter</Main-Class>
              <Main-Verticle>io.vertx.blog.first.MyFirstVerticle</
Main-Verticle>
            </manifestEntries>
```

```
              </transformer>
            </transformers>
            <artifactSet/>
            <outputFile>${project.build.directory}/${project.artifactId}-
${project.version}-fat.jar</outputFile>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

Now, run the following command:

```
mvn package
```

It will generate a specified JAR file (called `target/my-first-app-1.0-SNAPSHOT-fat.jar`, in this example). It is called `fat` because it contains all the necessary dependencies. This file will also contain `MANIFEST.MF` with the following entries in it:

```
Main-Class: io.vertx.core.Starter
Main-Verticle: io.vertx.blog.first.MyFirstVerticle
```

You can use any verticle instead of `io.vertx.blog.first.MyFirstVerticle`, used in this example, but `io.vertx.core.Starter` has to be there because that is the name of the `Vert.x` class that knows how to read the manifest and execute the method `start()` of the specified verticle. Now, you can run the following command:

```
java -jar target/my-first-app-1.0-SNAPSHOT-fat.jar
```

This command will execute the `start()` method of the `MyFirstVerticle` class the same way the `main()` method is executed in our example, which we will continue to use for the simplicity of demonstration.

To compliment the HTTP server, we can create an HTTP client too. However, first, we will modify the method `start()` in the `server` verticle to accept the parameter `name`:

```
public void start() throws Exception {
    HttpServer server = vertx.createHttpServer();
    server.requestStream().toObservable()
       .subscribe(request -> request.response()
       .end("Hi, " + request.getParam("name") +
             "! Hello from " +
             Thread.currentThread().getName() +
                        " on port " + port + "!\n\n")
       );
```

```
        server.rxListen(port).subscribe();
        System.out.println(Thread.currentThread().getName()
                   + " is waiting on port " + port + "...");
    }
```

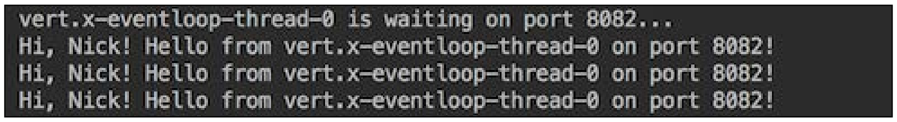
Now, we can create an HTTP `client` verticle that sends a request and prints out the response every second for 3 seconds, then stops:

```
    import io.vertx.rxjava.core.AbstractVerticle;
    import io.vertx.rxjava.core.http.HttpClient;
    import java.time.LocalTime;
    import java.time.temporal.ChronoUnit;

    public class Client extends AbstractVerticle {
      private int port;
      public Client(int port) {
        this.port = port;
      }
      public void start() throws Exception {
        HttpClient client = vertx.createHttpClient();
        LocalTime start = LocalTime.now();
        vertx.setPeriodic(1000, v -> {
           client.getNow(port, "localhost", "?name=Nick",
             r -> r.bodyHandler(System.out::println));
             if(ChronoUnit.SECONDS.between(start,
                                   LocalTime.now()) > 3 ){
               vertx.undeploy(deploymentID());
           }
        });
      }
    }
```

Let's assume we deploy both verticles as follows:

```
    RxHelper.deployVerticle(vertx(), new Server2(8082));
    RxHelper.deployVerticle(vertx(), new Client(8082));
```

The output will be as follows:



```
vert.x-eventloop-thread-0 is waiting on port 8082...
Hi, Nick! Hello from vert.x-eventloop-thread-0 on port 8082!
Hi, Nick! Hello from vert.x-eventloop-thread-0 on port 8082!
Hi, Nick! Hello from vert.x-eventloop-thread-0 on port 8082!
```

In this last example, we demonstrated how to create an HTTP client and periodic service. Now, let's add more functionality to our system. For example, let's add another verticle that will interact with the database and use it via the HTTP server we have already created.

First, we need to add this dependency:

```
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-jdbc-client</artifactId>
    <version>${vertx.version}</version>
</dependency>
```

The newly added JAR file allows us to create an in-memory database and a handler to access it:

```
public class DbHandler {
  private JDBCClient dbClient;
  private static String SQL_CREATE_WHO_CALLED =
    "CREATE TABLE IF NOT EXISTS " +
          "who_called ( name VARCHAR(10), " +
          "create_ts TIMESTAMP(6) DEFAULT now() )";
  private static String SQL_CREATE_PROCESSED =
    "CREATE TABLE IF NOT EXISTS " +
         "processed ( name VARCHAR(10), " +
         "length INTEGER, " +
         "create_ts TIMESTAMP(6) DEFAULT now() )";

  public DbHandler(Vertx vertx){
    JsonObject config = new JsonObject()
      .put("driver_class", "org.hsqldb.jdbcDriver")
      .put("url", "jdbc:hsqldb:mem:test?shutdown=true");
    dbClient = JDBCClient.createShared(vertx, config);
    dbClient.rxGetConnection()
      .flatMap(conn ->
                 conn.rxUpdate(SQL_CREATE_WHO_CALLED)
                       .doAfterTerminate(conn::close) )
      .subscribe(r ->
        System.out.println("Table who_called created"),
                           Throwable::printStackTrace);
    dbClient.rxGetConnection()
      .flatMap(conn ->
                 conn.rxUpdate(SQL_CREATE_PROCESSED)
                      .doAfterTerminate(conn::close) )
      .subscribe(r ->
        System.out.println("Table processed created"),
                          Throwable::printStackTrace);

  }
}
```

Those familiar with RxJava can see that Vert.x code closely follows the style and naming convention of RxJava. Nevertheless, we encourage you to go through Vert.x documentation, because it has a very rich API that covers many more cases than just demonstrated. In the previous code, the operation `flatMap()` receives the function that runs the script and then closes the connection. The operation `doAfterTerminate()` in this case acts as if it was placed inside a finally block in a traditional code and closes the connection either in case of success or if an exception is generated. The `subscribe()` method has several overloaded versions. For our code, we have selected the one that takes two functions one is going to be executed in the case of success (we print a message about the table being created) and another in the case of an exception (we just print the stack trace then).

To use the created database, we can add to `DbHandler` methods `insert()`, `process()`, and `readProcessed()` that will allow us to demonstrate how to build a reactive system. The code for the method `insert()` can look like this:

```
private static String SQL_INSERT_WHO_CALLED =
             "INSERT INTO who_called(name) VALUES (?)";
public void insert(String name, Action1<UpdateResult>
                onSuccess, Action1<Throwable> onError){
  printAction("inserts " + name);
  dbClient.rxGetConnection()
    .flatMap(conn ->
        conn.rxUpdateWithParams(SQL_INSERT_WHO_CALLED,
                            new JsonArray().add(name))
                       .doAfterTerminate(conn::close) )
    .subscribe(onSuccess, onError);
}
```

The `insert()` method, as well as other methods we are going to write, takes full advantage of Java functional interfaces. It creates a record in the table `who_called` (using the passed in parameter `name`). Then, the operation `subscribe()` executes one of the two functions passed in by the code that calls this method. We use the method `printAction()` only for better traceability:

```
private void printAction(String action) {
  System.out.println(this.getClass().getSimpleName()
                                     + " " + action);
}
```

The method `process()` also accepts two functions but does not need other parameters. It processes all the records from the table `who_called` that are not processed yet (not listed in the table `processed`):

```
private static String SQL_SELECT_TO_PROCESS =
  "SELECT name FROM who_called w where name not in " +
  "(select name from processed) order by w.create_ts " +
  "for update";
private static String SQL_INSERT_PROCESSED =
     "INSERT INTO processed(name, length) values(?, ?)";
public void process(Func1<JsonArray, Observable<JsonArray>>
                     process, Action1<Throwable> onError) {
  printAction("process all records not processed yet");
  dbClient.rxGetConnection()
    .flatMapObservable(conn ->
       conn.rxQueryStream(SQL_SELECT_TO_PROCESS)
           .flatMapObservable(SQLRowStream::toObservable)
           .flatMap(process)
           .flatMap(js ->
              conn.rxUpdateWithParams(SQL_INSERT_PROCESSED, js)
                  .flatMapObservable(ur->Observable.just(js)))
           .doAfterTerminate(conn::close))
    .subscribe(js -> printAction("processed " + js), onError);
}
```

If two threads are reading the table `who_called` for the purpose of selecting records not processed yet, the clause `for update` in the SQL query makes sure that only one gets each record, so they are not going to be processed twice. The significant advantage of the method `process()` code is its usage of the `rxQUeryStream()` operation that emits the found records one at a time so that they are processed independently of each other. In the case of a big number of not processed records, such a solution guarantees a smooth delivery of the results without the spiking of the resources consumption. The following `flatMap()` operation does processing using the function passed in. The only requirement for that function is that it must return one integer value (in `JsonArray`) that is going to be used as a parameter for the `SQL_INSERT_PROCESSED` statement. So, it is up to the code that calls this method to decide the nature of the processing. The rest of the code is similar to the method `insert()`. The code indentation helps to follow the nesting of the operations.

The method `readProcessed()` has code that looks very similar to the code of the method `insert()`:

```
private static String SQL_READ_PROCESSED =
  "SELECT name, length, create_ts FROM processed
                       order by create_ts desc limit ?";
```

```
public void readProcessed(String count, Action1<ResultSet>
                  onSuccess, Action1<Throwable> onError) {
  printAction("reads " + count +
                        " last processed records");
  dbClient.rxGetConnection()
   .flatMap(conn ->
      conn.rxQueryWithParams(SQL_READ_PROCESSED,
                      new JsonArray().add(count))
                   .doAfterTerminate(conn::close) )
   .subscribe(onSuccess, onError);
}
```

The preceding code reads the specified number of the latest processed records. The difference from the method `process()` is that the method `readProcessed()` returns all the read records in one result set, so it is up to the user of this method to decide how to process the result in bulk or one at a time. We show all these possibilities just to demonstrate the variety of the possible options. With the `DbHandler` class in place, we are ready to use it and create the `DbServiceHttp` microservice, which allows a remote access to the `DbHandler` capabilities by wrapping around it an HTTP server. Here is the constructor of the new microservice:

```
public class DbServiceHttp extends AbstractVerticle {
  private int port;
  private DbHandler dbHandler;
  public DbServiceHttp(int port) {
    this.port = port;
  }
  public void start() throws Exception {
    System.out.println(this.getClass().getSimpleName() +
                        "(" + port + ") starts...");
    dbHandler = new DbHandler(vertx);
    Router router = Router.router(vertx);
    router.put("/insert/:name").handler(this::insert);
    router.get("/process").handler(this::process);
    router.get("/readProcessed")
                      .handler(this::readProcessed);
    vertx.createHttpServer()
          .requestHandler(router::accept).listen(port);
  }
}
```

In the earlier mentioned code, you can see how the URL mapping is done in Vert.x. For each possible route, a corresponding `Verticle` method is assigned, each accepting the `RoutingContext` object that contains all the data of HTTP context, including the `HttpServerRequest` and `HttpServerResponse` objects. A variety of convenience methods allows us to easily access the URL parameters and other data necessary to process the request. Here is the method `insert()` referred in the `start()` method:

```
private void insert(RoutingContext routingContext) {
  HttpServerResponse response = routingContext.response();
  String name = routingContext.request().getParam("name");
  printAction("insert " + name);
  Action1<UpdateResult> onSuccess =
    ur -> response.setStatusCode(200).end(ur.getUpdated() +
                 " record for " + name + " is inserted");
  Action1<Throwable> onError = ex -> {
    printStackTrace("process", ex);
    response.setStatusCode(400)
        .end("No record inserted due to backend error");
  };
  dbHandler.insert(name, onSuccess, onError);
}
```

All it does is extracts the parameter `name` from the request and constructs the two functions necessary to call method `insert()` of `DbHandler` we discussed earlier. The method `process()` looks similar to the previous method `insert()`:

```
private void process(RoutingContext routingContext) {
  HttpServerResponse response = routingContext.response();
  printAction("process all");
  response.setStatusCode(200).end("Processing...");
  Func1<JsonArray, Observable<JsonArray>> process =
    jsonArray -&gt; {
      String name = jsonArray.getString(0);
      JsonArray js =
            new JsonArray().add(name).add(name.length());
       return Observable.just(js);
  };
  Action1<Throwable> onError = ex -> {
     printStackTrace("process", ex);
     response.setStatusCode(400).end("Backend error");
  };
  dbHandler.process(process, onError);
}
```

The function `process` mentioned earlier defines what should be done with the records coming from the `SQL_SELECT_TO_PROCESS` statement inside the method `process()` in `DbHandler`. In our case, it calculates the length of the caller's name and passes it as a parameter along with the name itself (as a return value) to the next SQL statement that inserts the result into the table `processed`.

Here is the method `readProcessed()`:

```
private void readProcessed(RoutingContext routingContext) {
  HttpServerResponse response = routingContext.response();
  String count = routingContext.request().getParam("count");
  printAction("readProcessed " + count + " entries");
  Action1<ResultSet> onSuccess = rs -> {
     Observable.just(rs.getResults().size() > 0 ?
       rs.getResults().stream().map(Object::toString)
                   .collect(Collectors.joining("\n")) : "")
       .subscribe(s -> response.setStatusCode(200).end(s) );
  };
  Action1<Throwable> onError = ex -> {
      printStackTrace("readProcessed", ex);
      response.setStatusCode(400).end("Backend error");
  };
  dbHandler.readProcessed(count, onSuccess, onError);
}
```

That is where (in the previous code in the function `onSuccess()`) the result set from the query `SQL_READ_PROCESSED` is read and used to construct the response. Notice that we do it by creating an `Observable` first, then subscribing to it and passing the result of the subscription as the response into method `end()`. Otherwise, the response can be returned without waiting for the response to be constructed.

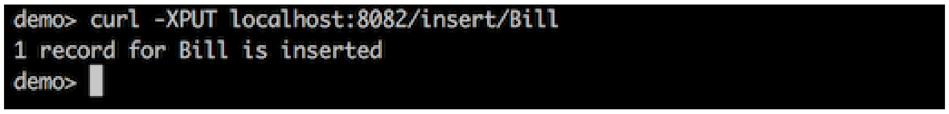
Now, we can launch our reactive system by deploying the `DbServiceHttp` verticle:

```
RxHelper.deployVerticle(vertx(), new DbServiceHttp(8082));
```

If we do that, in the output we will see the following lines of code:

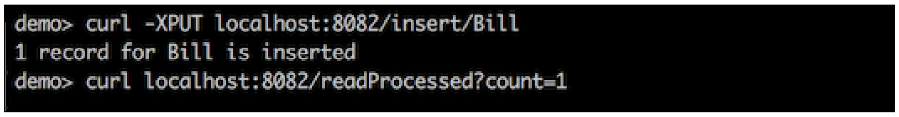
```
DbServiceHttp(8082) starts...
Table processed created
Table who_called created
```

In another window, we can issue the command that generates an HTTP request:



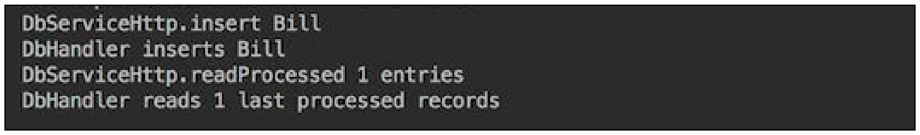
```
demo> curl -XPUT localhost:8082/insert/Bill
1 record for Bill is inserted
demo>
```

If we read the processed records now, there should be none:



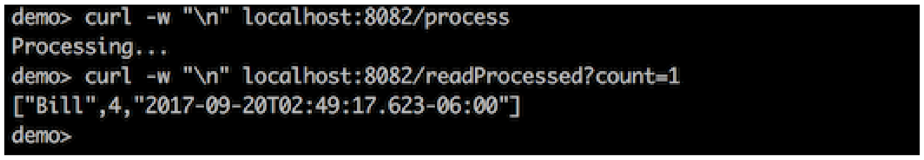
```
demo> curl -XPUT localhost:8082/insert/Bill
1 record for Bill is inserted
demo> curl localhost:8082/readProcessed?count=1
```

The log messages show the following:



```
DbServiceHttp.insert Bill
DbHandler inserts Bill
DbServiceHttp.readProcessed 1 entries
DbHandler reads 1 last processed records
```

Now, we can request processing of the existing records and then read the results again:



```
demo> curl -w "\n" localhost:8082/process
Processing...
demo> curl -w "\n" localhost:8082/readProcessed?count=1
["Bill",4,"2017-09-20T02:49:17.623-06:00"]
demo>
```

In principle, it is enough already to build a reactive system. We can deploy many `DbServiceHttp` microservices on different ports or cluster them to increase processing capacity, resilience, and responsiveness. We can wrap other services inside an HTTP client or an HTTP server and let them talk to each other, processing the input and passing the results along the processing pipeline.

However, Vert.x also has a feature that even better suits the message-driven architecture (without using HTTP). It is called an event bus. Any verticle has access to the event bus and can send any message to any address (which is just a string) using either method `send()` (`rxSend()` in the case of reactive programming) or method `publish()`. One or many verticles can register themselves as a consumer for a certain address.

If many verticles are consumers for the same address, then the method `send()` (`rxSend()`) delivers the message only to one of them (using a round-robin algorithm to pick the next consumer). The method `publish()`, as you would expect, delivers the message to all consumers with the same address. Let's see an example, using the already familiar `DbHandler` as the main working horse.

A microservice, based on an event bus, looks very similar to the one based on the HTTP protocol we discussed already:

```
public class DbServiceBus extends AbstractVerticle {
  private int id;
  private String instanceId;
  private DbHandler dbHandler;
  public static final String INSERT = "INSERT";
  public static final String PROCESS = "PROCESS";
  public static final String READ_PROCESSED
                            = "READ_PROCESSED";
  public DbServiceBus(int id) { this.id = id; }
  public void start() throws Exception {
    this.instanceId = this.getClass().getSimpleName()
                                   + "(" + id + ")";
    System.out.println(instanceId + " starts...");
    this.dbHandler = new DbHandler(vertx);
    vertx.eventBus().consumer(INSERT).toObservable()
      .subscribe(msg -> {
         printRequest(INSERT, msg.body().toString());
         Action1<UpdateResult> onSuccess
                             = ur -> msg.reply(...);
         Action1<Throwable> onError
                   = ex -> msg.reply("Backend error");
         dbHandler.insert(msg.body().toString(),
                               onSuccess, onError);
    });

    vertx.eventBus().consumer(PROCESS).toObservable()
        .subscribe(msg -> {
                  .....
                 dbHandler.process(process, onError);
        });

    vertx.eventBus().consumer(READ_PROCESSED).toObservable()
        .subscribe(msg -> {
                 ...
            dbHandler.readProcessed(msg.body().toString(),
                                      onSuccess, onError);
        });
    }
```

We simplified the preceding code by skipping some sections (that are very similar to the `DbServiceHttp` class) and trying to highlight the code structure. For demo purposes, we will deploy two instances of this class and send three messages to each of the addresses `INSERT`, `PROCESS`, and `READ_PROCESSED`:

```
void demo_DbServiceBusSend() {
  Vertx vertx = vertx();
  RxHelper.deployVerticle(vertx, new DbServiceBus(1));
  RxHelper.deployVerticle(vertx, new DbServiceBus(2));
  delayMs(200);
  String[] msg1 = {"Mayur", "Rohit", "Nick" };
  RxHelper.deployVerticle(vertx,
    new PeriodicServiceBusSend(DbServiceBus.INSERT, msg1, 1));
  String[] msg2 = {"all", "all", "all" };
  RxHelper.deployVerticle(vertx,
    new PeriodicServiceBusSend(DbServiceBus.PROCESS, msg2, 1));
  String[] msg3 = {"1", "1", "2", "3" };
  RxHelper.deployVerticle(vertx,
     new PeriodicServiceBusSend(DbServiceBus.READ_PROCESSED,
                                                       msg3, 1));
}
```

Notice the delay for 200 ms we inserted using the method `delayMs()`:

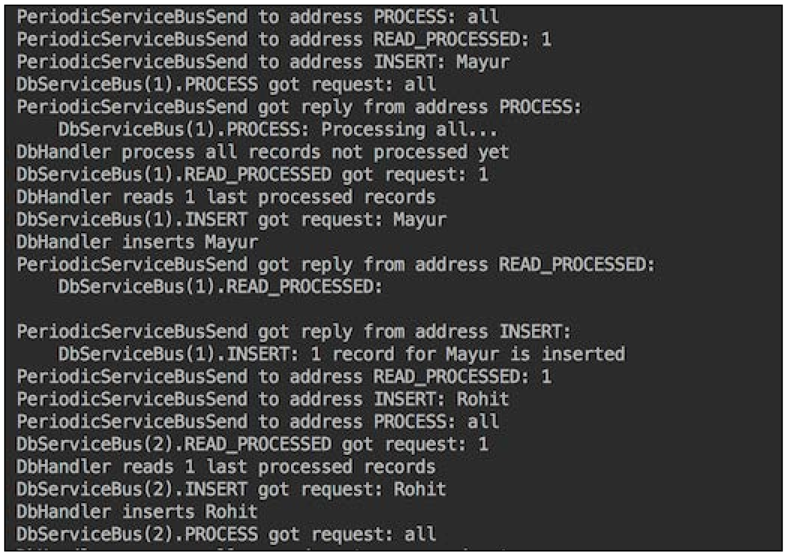
```
void delayMs(int ms){
    try {
        TimeUnit.MILLISECONDS.sleep(ms);
    } catch (InterruptedException e) {}
}
```

The delay is necessary to let the `DbServiceBus` verticle to be deployed and started (and the consumers registered with the address). Otherwise, an attempt to send a message may fail because the consumer is not registered with the address yet. The `PeriodicServiceBusSend()` verticle code is as follows:

```
public class PeriodicServiceBusSend
                           extends AbstractVerticle {
  private EventBus eb;
  private LocalTime start;
  private String address;
  private String[] caller;
  private int delaySec;
  public PeriodicServiceBusSend(String address,
                     String[] caller, int delaySec) {
        this.address = address;
        this.caller = caller;
```

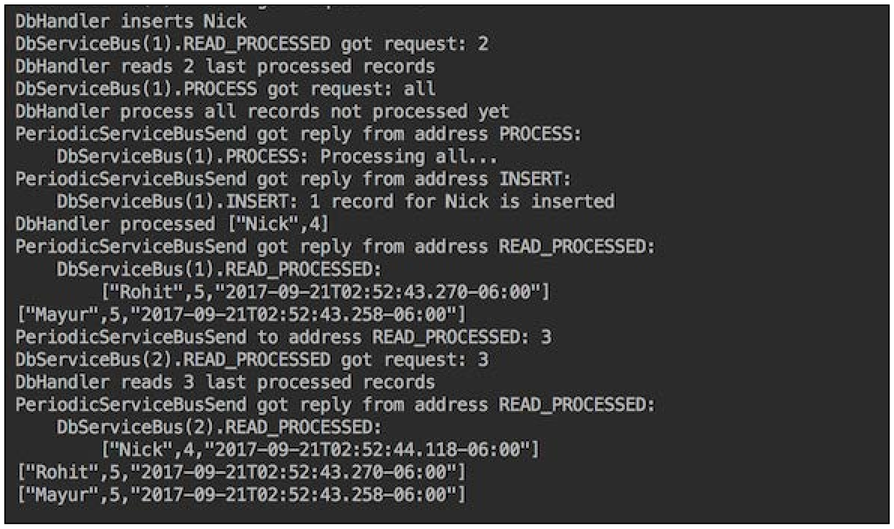
```
            this.delaySec = delaySec;
      }
      public void start() throws Exception {
        System.out.println(this.getClass().getSimpleName()
          + "(" + address + ", " + delaySec + ") starts...");
        this.eb = vertx.eventBus();
        this.start  = LocalTime.now();
        vertx.setPeriodic(delaySec * 1000, v -> {
           int i = (int)ChronoUnit.SECONDS.between(start,
                                         LocalTime.now()) - 1;
           System.out.println(this.getClass().getSimpleName()
              + " to address " + address + ": " + caller[i]);
           eb.rxSend(address, caller[i]).subscribe(reply -> {
             System.out.println(this.getClass().getSimpleName()
                        + " got reply from address " + address
                                   + ":\n    " + reply.body());
              if(i + 1 >= caller.length ){
                   vertx.undeploy(deploymentID());
              }
           }, Throwable::printStackTrace);
        });
      }
    }
```

The previous code sends a message to an address every `delaySec` seconds as many times as the length of the array `caller[]`, and then undeploys the verticle (itself). If we run the demo, the beginning of the output will be as follows:



```
PeriodicServiceBusSend to address PROCESS: all
PeriodicServiceBusSend to address READ_PROCESSED: 1
PeriodicServiceBusSend to address INSERT: Mayur
DbServiceBus(1).PROCESS got request: all
PeriodicServiceBusSend got reply from address PROCESS:
    DbServiceBus(1).PROCESS: Processing all...
DbHandler process all records not processed yet
DbServiceBus(1).READ_PROCESSED got request: 1
DbHandler reads 1 last processed records
DbServiceBus(1).INSERT got request: Mayur
DbHandler inserts Mayur
PeriodicServiceBusSend got reply from address READ_PROCESSED:
    DbServiceBus(1).READ_PROCESSED:

PeriodicServiceBusSend got reply from address INSERT:
    DbServiceBus(1).INSERT: 1 record for Mayur is inserted
PeriodicServiceBusSend to address READ_PROCESSED: 1
PeriodicServiceBusSend to address INSERT: Rohit
PeriodicServiceBusSend to address PROCESS: all
DbServiceBus(2).READ_PROCESSED got request: 1
DbHandler reads 1 last processed records
DbServiceBus(2).INSERT got request: Rohit
DbHandler inserts Rohit
DbServiceBus(2).PROCESS got request: all
```

As you can see, for each address, only `DbServiceBus(1)` was a receiver of the first message. The second message to the same address was received by `DbServiceBus(2)`. That was the round-robin algorithm (which we mentioned earlier) in action. The final section of the output looks like this:



```
DbHandler inserts Nick
DbServiceBus(1).READ_PROCESSED got request: 2
DbHandler reads 2 last processed records
DbServiceBus(1).PROCESS got request: all
DbHandler process all records not processed yet
PeriodicServiceBusSend got reply from address PROCESS:
    DbServiceBus(1).PROCESS: Processing all...
PeriodicServiceBusSend got reply from address INSERT:
    DbServiceBus(1).INSERT: 1 record for Nick is inserted
DbHandler processed ["Nick",4]
PeriodicServiceBusSend got reply from address READ_PROCESSED:
    DbServiceBus(1).READ_PROCESSED:
        ["Rohit",5,"2017-09-21T02:52:43.270-06:00"]
["Mayur",5,"2017-09-21T02:52:43.258-06:00"]
PeriodicServiceBusSend to address READ_PROCESSED: 3
DbServiceBus(2).READ_PROCESSED got request: 3
DbHandler reads 3 last processed records
PeriodicServiceBusSend got reply from address READ_PROCESSED:
    DbServiceBus(2).READ_PROCESSED:
        ["Nick",4,"2017-09-21T02:52:44.118-06:00"]
["Rohit",5,"2017-09-21T02:52:43.270-06:00"]
["Mayur",5,"2017-09-21T02:52:43.258-06:00"]
```

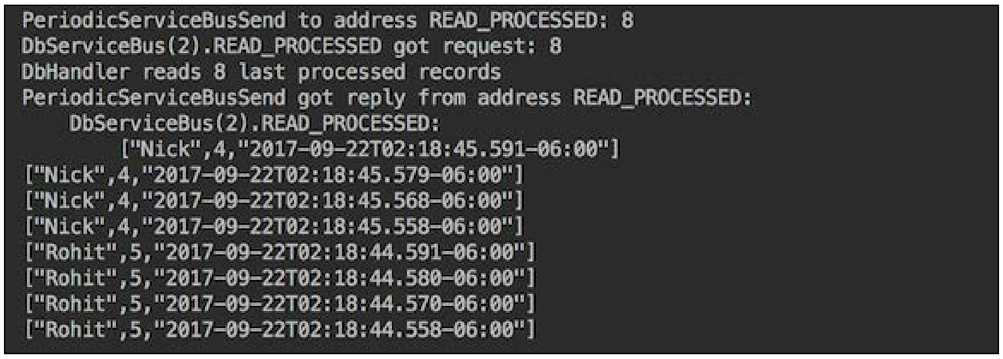
We can deploy as many verticles of the same type as needed. For example, let's deploy four verticles that send messages to the address `INSERT`:

```
String[] msg1 = {"Mayur", "Rohit", "Nick" };
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusSend(DbServiceBus.INSERT, msg1, 1));
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusSend(DbServiceBus.INSERT, msg1, 1));
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusSend(DbServiceBus.INSERT, msg1, 1));
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusSend(DbServiceBus.INSERT, msg1, 1));
```

To see the results, we will also ask the reading Verticle to read the last eight records:

```
String[] msg3 = {"1", "1", "2", "8" };
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusSend(DbServiceBus.READ_PROCESSED,
                                                    msg3, 1));
```

The result (the final section of the output) then will be as expected:



```
PeriodicServiceBusSend to address READ_PROCESSED: 8
DbServiceBus(2).READ_PROCESSED got request: 8
DbHandler reads 8 last processed records
PeriodicServiceBusSend got reply from address READ_PROCESSED:
    DbServiceBus(2).READ_PROCESSED:
        ["Nick",4,"2017-09-22T02:18:45.591-06:00"]
["Nick",4,"2017-09-22T02:18:45.579-06:00"]
["Nick",4,"2017-09-22T02:18:45.568-06:00"]
["Nick",4,"2017-09-22T02:18:45.558-06:00"]
["Rohit",5,"2017-09-22T02:18:44.591-06:00"]
["Rohit",5,"2017-09-22T02:18:44.580-06:00"]
["Rohit",5,"2017-09-22T02:18:44.570-06:00"]
["Rohit",5,"2017-09-22T02:18:44.558-06:00"]
```

Four verticles have sent the same messages, so each name was sent four times and processed that is what we see in the previous output.

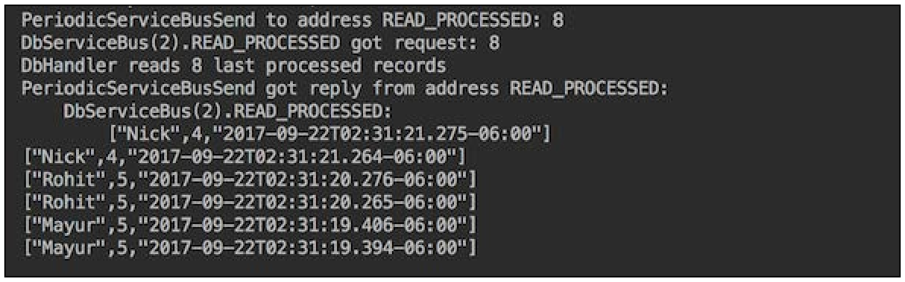
We will now return to one inserting periodic verticle but will change it from using the method `rxSend()` to the method `publish()`:

```
PeriodicServiceBusPublish(String address, String[] caller, int
delaySec) {
  ...
  vertx.setPeriodic(delaySec * 1000, v -> {
    int i = (int)ChronoUnit.SECONDS.between(start,
                                  LocalTime.now()) - 1;
    System.out.println(this.getClass().getSimpleName()
            + " to address " + address + ": " + caller[i]);
    eb.publish(address, caller[i]);
    if(i + 1 == caller.length ){
        vertx.undeploy(deploymentID());
    }
  });
}
```

This change would mean that the message has to be sent to all verticles that are registered as the consumers at that address. Now, let's run the following code:

```
Vertx vertx = vertx();
RxHelper.deployVerticle(vertx, new DbServiceBus(1));
RxHelper.deployVerticle(vertx, new DbServiceBus(2));
delayMs(200);
String[] msg1 = {"Mayur", "Rohit", "Nick" };
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusPublish(DbServiceBus.INSERT,
                                               msg1, 1));
delayMs(200);
String[] msg2 = {"all", "all", "all" };
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusSend(DbServiceBus.PROCESS,
                                               msg2, 1));
String[] msg3 = {"1", "1", "2", "8" };
RxHelper.deployVerticle(vertx,
  new PeriodicServiceBusSend(DbServiceBus.READ_PROCESSED,
                                               msg3, 1));
```

We have included another delay for 200 ms to give the publishing verticle time to send the message. The output (in the final section) now shows that each message was processed twice:



```
PeriodicServiceBusSend to address READ_PROCESSED: 8
DbServiceBus(2).READ_PROCESSED got request: 8
DbHandler reads 8 last processed records
PeriodicServiceBusSend got reply from address READ_PROCESSED:
    DbServiceBus(2).READ_PROCESSED:
        ["Nick",4,"2017-09-22T02:31:21.275-06:00"]
["Nick",4,"2017-09-22T02:31:21.264-06:00"]
["Rohit",5,"2017-09-22T02:31:20.276-06:00"]
["Rohit",5,"2017-09-22T02:31:20.265-06:00"]
["Mayur",5,"2017-09-22T02:31:19.406-06:00"]
["Mayur",5,"2017-09-22T02:31:19.394-06:00"]
```

That is because two consumers `DbServiceBus(1)` and `DbServiceBus(2)` were deployed, and each received a message to the address `INSERT` and inserted it in the table `who_called`.

All the previous examples we have run in one JVM process. If necessary, Vert.x instances can be deployed in different JVM processes and clustered by adding the `-cluster` option to the run command. Therefore, they share the event bus and the addresses are visible to all Vert.x instances. This way, the resources can be added to each address as needed. For example, we can increase the number of processing microservices only and compensate the load's increase.

Other frameworks we mentioned earlier have similar capabilities. They make microservices creation easy and may encourage breaking the application into tiny single-method operations with an expectation of assembling a very resilient and responsive system.

However, these are not the only criteria of good quality. System decomposition increases the complexity of its deployment. Also, if one development team is responsible for many microservices, the complexity of versioning so many pieces in different stages (development, test, integration test, certification, staging, production) may lead to confusion and a very challenging deployment process, which, in turn, may slow down the rate of changes necessary to keep the system in sync with the market requirements.

In addition to the developing of the microservices, many other aspects have to be addressed to support the reactive system:

- A monitoring system has to be designed to provide an insight into the state of the application, but it should not be so complex as to pull the development resources away from the main application.
- Alerts have to be installed to warn the team about possible and actual issues in a timely manner, so they can be addressed before affecting the business.
- If possible, self-correcting automated processes have to be implemented. For example, the system should be able to add and release resources in accordance with the current load; the retry logic has to be implemented with a reasonable upper limit of a attempts before declaring the failure.
- A layer of circuit breakers has to protect the system from the domino effect when failure of one component deprives other components of the necessary resources.
- An embedded testing system should be able to introduce disruptions and simulate processing load to ensure that the application resilience and responsiveness do not degrade over time. For example, the Netflix team has introduced a **chaos monkey** a system that is able to shut down various parts of the production system to test the ability to recover. They use it even in production because a production environment has a specific configuration, and no test in another environment can guarantee that all possible issues are found.

One of the main considerations of a reactive system design is the selection of the deployment methodology that can be either container-less, self-contained, or in-container. We will look into the pros and cons of each of these approaches in the following sections of this lesson.

Container-Less Deployment

People use the term **container** to refer to very different things. In the original usage, a container was something that carried its content from one location to another without changing anything inside. However, when servers were introduced, only one aspect was emphasized the ability to hold an application to contain it. Also, another meaning was added to provide life-supportive infrastructure so that the container's content (an application) can not only survive but also be active and respond to the external requests. Such a redefined notion of a container was applied to web servers (servlet container), application servers (an application container with or without an EJB container), and other software facilities that provided the supportive environment for applications. Sometimes, even the JVM itself was called a container, but this association did not survive, probably, because the ability to actively engage (execute) the content does not align well with the original meaning of a container.

That is why, later, when people started talking about container-less deployment, they typically meant the ability to deploy an application into a JVM directly, without first installing WebSphere, WebLogic, JBoss, or any other mediating software that provides the runtime environment for the application.

In the previous sections, we described many frameworks that allow us to build and deploy an application (or rather a reactive system of microservices) without the need for any other container beyond the JVM itself. All you need to do is to build a fat JAR file that includes all the dependencies (except those that come from the JVM itself) and then run it as a standalone Java process:

```
$ java -jar myfatjar.jar
```

Well, you also need to make sure that `MANIFEST.MF` in your JAR file has an entry `main` class that points to the fully qualified class name that has the `main()` method and will be run at the startup. We have described how to do it in the previous section, *Building Microservices*.

That is the promised compile-once-run-everywhere of Java, everywhere meaning everywhere where JVM of a certain version or higher is installed. There are several advantages and disadvantages of this approach. We will discuss them not relative to the traditional deployment in a server container. The advantages of deployment without using the traditional containers are quite obvious, starting with much fewer (if any) licensing costs and ending up with much a lighter deployment and scalability process, not even mentioning much less consumption of resources. Instead, we will compare container-less deployment not with the traditional one, but with a self-contained and an in-container in a new generation of containers that have been developed a few years ago.

They allow the ability not only to contain and execute the contained code, which the traditional containers did too, but also to move it to a different location without any change to the contained code. From now on, by a container, we are going to mean only the new ones.

The advantages of container-less deployment are as follows:

- It is easy to add more Java processes either inside the same physical (or virtual or in the cloud) machine or on new hardware
- An isolation level between processes is high, which is especially important in the shared environment when you have no control over other co-deployed applications, and it is possible that a rogue application would try to penetrate the neighboring execution environment
- It has a small footprint since it does not include anything else beyond the application itself or a group of microservices

The disadvantages of container-less deployment are as follows:

- Each JAR file requires the JVM of a certain version or higher, which may force you to bring up a new physical or virtual machine just for this reason, to deploy one particular JAR file
- In the case of an environment you do not control, your code might be deployed with a wrong version of JVM, which could lead to unpredictable results
- Processes in the same JVM compete for resources, which are especially hard to manage in the case of the environments shared by different teams or different companies
- When several microservices are bundled into the same JAR file, they might require different versions of a third-party library or even incompatible libraries

Microservices can be deployed one per JAR or bundled together by a team, by related services, by the unit of scale, or using another criterion. Not the least important consideration is the total number of such JAR files. As this number grows (Google today deals with hundreds of thousands of deployment units at a time), it may become impossible to handle deployment via simple bash script and require a complex process that allows account ability for possible incompatibilities. If that is the case, then it is reasonable to consider using virtual machines or containers (in their new incarnation, see the following section) for better isolation and management.

Self-Contained Microservices

Self-contained microservices look much similar to container-less. The only difference is that the JVM (or JRE, actually) or any other external frameworks and servers necessary for the application to run are included in the fat JAR file too. There are many ways to build such an all-inclusive JAR file.

Spring Boot, for example, provides a convenient GUI with checkbox list that allows you to select which parts of your Spring Boot application and the external tools you would like to package. Similarly, WildFly Swarm allows you to choose which parts of the Java EE components you would like to bundle along with your application. Alternatively, you can do it yourself using the `javapackager` tool. It compiles and packages the application and JRE in the same JAR file (it can also be `.exe` or `.dmg`) for distribution. You can read about the tool on the Oracle website `https://docs.oracle.com/javase/9/tools/javapackager.htm` or you can just run the command `javapackager` on a computer where JDK is installed (it comes with Java 8 too) you will get the list of tool options and their brief description.

Basically, to use the `javapackager` tool, all you need to do is to prepare a project with everything you would like to package together, including all the dependencies (packaged in JAR files), and run the `javapackager` command with the necessary options that allow you to specify the type of output you would like to have (`.exe` or `.dmg`, for example), the JRE location you would like to bundle together, the icon to use, the `main` class entry for `MANIFEST.MF`, and so on. There are also Maven plugins that make the packaging command simpler because much of the setup has to be configured in `pom.xml`.

The advantages of self-contained deployment are as follows:

- It is one file (with all the microservices that compose the reactive system or some part of it) to handle, which is simpler for a user and for a distributor
- There is no need to pre-install JRE and no risk of mismatching the required version
- The isolation level is high because your application has a dedicated JRE, so the risk of an intrusion from a co-deployed application is minimal
- You have full control over the dependencies included in the bundle

The disadvantages are as follows:

- The size of the file is bigger, which might be an impediment if it has to be downloaded
- The configuration is more complex than in the case of a container-less JAR file

- The bundle has to be generated on a platform that matches the target one, which might lead to mismatch if you have no control over the installation process
- Other processes deployed on the same hardware or virtual machine can hog the resources critical for your application needs, which are especially hard to manage if your application is downloaded and run not by the team that has developed it

In-Container Deployment

Those who are familiar with **Virtual Machine** (**VM**) and not familiar with modern containers (such as Docker, Rocket by CoreOS, VMware Photon, or similar) could get the impression that we were talking about VM while saying that a container could not only contain and execute the contained code, but also to move it to a different location without any change to the contained code. If so, that would be quite an apt assumption. VM does allow all of that, and a modern container can be considered a lightweight VM as it also allows the allocation of resources and provides the feeling of a separate machine. Yet, a container is not a full-blown isolated virtual computer.

The key difference is that the bundle that can be passed around as a VM includes an entire operating system (with the application deployed). So, it is quite possible that a physical server running two VMs would have two different operating systems running on it. By contrast, a physical server (or a VM) running three containerized applications has only one operating system running, and the two containers share (read-only) the operating system kernel, each having its own access (mount) for writing to the resources they do not share. This means, for example, a much shorter start time, because starting a container does not require us to boot the operating system (as in the case of a VM).

For an example, let's take a closer look at Docker the community leader in container. In 2015, an initiative called **Open Container Project** was announced, later renamed the **Open Container Initiative** (**OCI**), which was supported by Google, IBM, Amazon, Microsoft, Red Hat, Oracle, VMware, HP, Twitter, and many other companies. Its purpose was to develop industry standards for a container format and container runtime software for all platforms. Docker has donated about 5 percent of its code base to the project because its solution was chosen as the starting point.

There is an extensive Docker documentation at: `https://docs.docker.com`. Using Docker, one can include in the package all the Java EE Container and the application as a Docker image, achieving essentially the same result as with a self-contained deployment. Then, you can launch your application by starting the Docker image in the Docker engine using this command:

```
$ docker run mygreatapplication
```

It starts a process that looks like running an OS on a physical computer, although it can also be happening in a cloud inside a VM that is running on the physical Linux server shared by many different companies and individuals. That is why an isolation level (which, in the case of containers, is almost as high as in a VM) may be critical in choosing between different deployment models.

A typical recommendation would be to put one microservice in each container, but nothing prevents you from putting several microservices in one Docker image (or any other container for that matter). However, there are already mature systems of container management (in the world of containers called **orchestration**) that can help you with deployment, so the complexity of having many containers, although a valid consideration, should not be a big obstacle if resilience and responsiveness are at stake. One of the popular orchestrations called **Kubernetes** supports microservice registry, discovery, and load balancing. Kubernetes can be used in any cloud or in a private infrastructure.

Containers allow a fast, reliable, and consistent deployment in practically any of the current deployment environments, whether it is your own infrastructure or a cloud at Amazon, Google, or Microsoft. They also allow the easy movement of an application through the development, testing, and production stages. Such infrastructure independence allows you, if necessary, to use a public cloud for development and testing and your own computers for production.

Once a base operating image is created, each development team can then build their application on top, thus avoiding the complexities of environment configuration. The versions of a container can also be tracked in a version control system.

The advantages of using containers are as follows:

- The level of isolation is the highest if compared with container-less and self-contained deployment. In addition, more efforts were put recently into adding security to containers.
- Each container is managed, distributed, deployed, started, and stopped by the same set of commands.

- There is no need to pre-install JRE and risk of mismatching the required version.
- You have full control over the dependencies included in the container.
- It is straightforward to scale up/down each microservice by adding/removing container instances.

The disadvantages of using containers are as follows:

- You and your team have to learn a whole new set of tools and become involved more heavily in the production stage. On the other hand, that seems to be the general tendency in recent years.

Summary

Microservices is a new architectural and design solution for highly loaded processing systems that became popular after being successfully used in production by such giants as Amazon, Google, Twitter, Microsoft, IBM, and others. It does not mean though that you must adopt it too, but you can consider the new approach and see if some or any of it can help your applications to be more resilient and responsive.

Using microservices can provide a substantial value, but it is not free. It comes with increased complexity of the need to manage many more units through all the lifecycle from requirements and development through testing to production. Before committing to the full-scale microservice architecture, give it a shot by implementing just a few microservices and move them all the way to production. Then, let it run for some time and gauge the experience. It will be very specific to your organization. Any successful solution must not be blindly copied but adopted as fit for your particular needs and abilities.

Better performance and overall efficiency often can be achieved by gradual improvements of what is already in place than by radical redesign and re-architecture.

In the next lesson, we will discuss and demonstrate new API that can improve your code by making it more readable and faster performing.

Assessments

1. Using the _________ object, various verticles can be deployed, which talk to each other, receive an external request, and process and store data as any other Java application, thus forming a system of microservices.
2. Which of the following is advantage of container-less deployment?
    1. Each JAR file requires the JVM of a certain version or higher, which may force you to bring up a new physical or virtual machine just for this reason, to deploy one particular JAR file
    2. In the case of an environment you do not control, your code might be deployed with a right version of JVM, which could lead to unpredictable results
    3. Processes in the same JVM compete for resources, which are especially hard to manage in the case of the environments shared by different teams or different companies
    4. It has a small footprint since it does not include anything else beyond the application itself or a group of microservices
3. State whether True or False: One way to support a transaction across several microservices is to create a service that would play the role of a Parallel Transaction Manager.
4. Which of the following are the Java frameworks that are included in Java 9?
    1. Akka
    2. Ninja
    3. Orange
    4. Selenium
5. State whether True or False: The level of isolation in a container is the highest if compared with container-less and self-contained deployment.

5
Making Use of New APIs to Improve Your Code

In the previous lessons, we talked about possible ways to improve the performance of your Java application--from using the new command and monitoring tools to adding multithreading and introducing reactive programming and even to radically re-architecting your current solution into an unruly and flexible bunch of small independent deployment units and microservices. Without knowing your particular situation, there is no way for us to guess which of the provided recommendations can be helpful to you. That's why, in this lesson, we will describe a few recent additions to the JDK that can be helpful to you too. As we mentioned in the previous lesson, the gain in performance and overall code improvement does not always require us to radically redesign it. Small incremental changes can sometimes bring more significant improvements than we could have expected.

To bring back our analogy of a pyramid building, instead of trying to change the logistics of the delivery of the stones to the final destination--in order to shorten the construction time--it is often prudent to look closer at the tools the builders are using first. If each operation can be completed in half the time, the overall time of the project's delivery can be shortened accordingly, even if each of the stone blocks travels the same, if not a larger, distance.

These are the improvements of the programming tools we will discuss in this lesson:

- Using filters on streams as a way to find what you need and to decrease workload
- A new stack-walking API as the way analyze the stack trace programmatically in order to apply an automatic correction
- New convenient static factory methods that create compact, unmodifiable collection instances

- The new `CompletableFuture` class as a way to access the results of asynchronous processing
- The JDK 9 stream API improvements that can speed up processing while making your code more readable

Filtering Streams

The `java.util.streams.Stream` interface was introduced in Java 8. It emits elements and supports a variety of operations that perform computations based on these elements. A stream can be finite or infinite, slow or fast emitting. Naturally, there is always a concern that the rate of the newly emitted elements may be higher than the rate of the processing. Besides, the ability to keep up with the input reflects the application's performance. The `Stream` implementations address the backpressure (when the rate of the element processing is lower than their emitting rate) by adjusting the emitting and processing rates using a buffer and various other techniques. In addition, it is always helpful if an application developer makes sure that the decision about processing or skipping each particular element is made as early as possible so that the processing resources are not wasted. Depending on the situation, different operations can be used for filtering the data.

Basic Filtering

The first and the most straightforward way to do filtering is using the `filter()` operation. To demonstrate all the following capabilities, we will use the `Senator` class:

```
public class Senator {
    private int[] voteYes, voteNo;
    private String name, party;
    public Senator(String name, String party,
                     int[] voteYes, int[] voteNo) {
        this.voteYes = voteYes;
        this.voteNo = voteNo;
        this.name = name;
        this.party = party;
    }
    public int[] getVoteYes() { return voteYes; }
    public int[] getVoteNo() { return voteNo; }
    public String getName() { return name; }
    public String getParty() { return party; }
    public String toString() {
        return getName() + ", P" +
            getParty().substring(getParty().length() - 1);
        }
    }
```

As you can see, this class captures a senator's name, party, and how they voted for each of the issues (`0` means `No` and `1` means `Yes`). If for a particular issue `i`, `voteYes[i]=0` , and `voteNo[i]=0`, it means that the senator was not present. It is not possible to have `voteYes[i]=1` and `voteNo[i]=1` for the same issue.

Let's assume that there are 100 senators, each belonging to one of the two parties: `Party1` or `Party2`. We can use these objects to collect statistics of how senators voted for the last 10 issues using the `Senate` class:

```
public class Senate {
  public static List<Senator> getSenateVotingStats(){
     List<Senator> results = new ArrayList<>();
     results.add(new Senator("Senator1", "Party1",
                       new int[]{1,0,0,0,0,0,1,0,0,1},
                       new int[]{0,1,0,1,0,0,0,0,1,0}));
     results.add(new Senator("Senator2", "Party2",
                       new int[]{0,1,0,1,0,1,0,1,0,0},
                       new int[]{1,0,1,0,1,0,0,0,0,1}));
     results.add(new Senator("Senator3", "Party1",
                       new int[]{1,0,0,0,0,0,1,0,0,1},
                       new int[]{0,1,0,1,0,0,0,0,1,0}));
     results.add(new Senator("Senator4", "Party2",
                       new int[]{1,0,1,0,1,0,1,0,0,1},
                       new int[]{0,1,0,1,0,0,0,0,1,0}));
     results.add(new Senator("Senator5", "Party1",
                       new int[]{1,0,0,1,0,0,0,0,0,1},
                       new int[]{0,1,0,0,0,0,1,0,1,0}));
     IntStream.rangeClosed(6, 98).forEach(i -> {
       double r1 = Math.random();
       String name = "Senator" + i;
       String party = r1 > 0.5 ? "Party1" : "Party2";
       int[] voteNo = new int[10];
       int[] voteYes = new int[10];
       IntStream.rangeClosed(0, 9).forEach(j -> {
         double r2 = Math.random();
         voteNo[j] = r2 > 0.4 ? 0 : 1;
         voteYes[j] = r2 < 0.6 ? 0 : 1;
       });
       results.add(new Senator(name,party,voteYes,voteNo));
     });
```

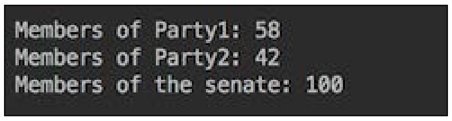
```
        results.add(new Senator("Senator99", "Party1",
                          new int[]{0,0,0,0,0,0,0,0,0,0},
                          new int[]{1,1,1,1,1,1,1,1,1,1}));
            results.add(new Senator("Senator100", "Party2",
                          new int[]{1,1,1,1,1,1,1,1,1,1},
                          new int[]{0,0,0,0,0,0,0,0,0,0}));
            return results;
        }
        public static int timesVotedYes(Senator senator){
            return Arrays.stream(senator.getVoteYes()).sum();
        }
    }
```

We hardcoded statistics for the first five senators so we can get predictable results while testing our filters and verify that the filters work. We also hardcoded voting statistics for the last two senators so we can have a predictable count while looking for senators who voted only `Yes` or only `No` for each of the ten issues. And we added the `timesVotedYes()` method, which provides the count of how many times the given `senator` voted `Yes`.

Now we can collect some data from the `Senate` class. For example, let's see how many members of each party comprise the `Senate` class:

```
List<Senator> senators = Senate.getSenateVotingStats();
long c1 = senators.stream()
   .filter(s -> s.getParty() == "Party1").count();
System.out.println("Members of Party1: " + c1);

long c2 = senators.stream()
   .filter(s -> s.getParty() == "Party2").count();
System.out.println("Members of Party2: " + c2);
System.out.println("Members of the senate: " + (c1 + c2));
```

The result of the preceding code differs from run to run because of the random value generator we used in the `Senate` class, so do not expect to see exactly the same numbers if you try to run the examples. What is important is that the total of the two party members should be equal 100--the total number of the senators in the `Senate` class:



```
Members of Party1: 58
Members of Party2: 42
Members of the senate: 100
```

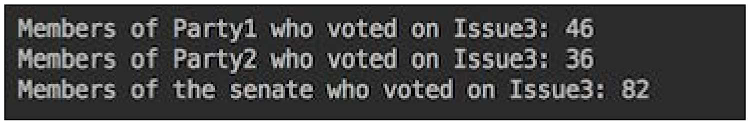
The expression `s -> s.getParty()=="Party1"` is the predicate that filters out only those senators who are members of `Party1`. So, the elements (`Senator` objects) of `Party2` do not get through and are not included in the count. That was pretty straightforward.

Now let's look at a more complex example of filtering. Let's count how many senators of each party voted on `issue 3`:

```
int issue = 3;
c1 = senators.stream()
  .filter(s -> s.getParty() == "Party1")
  .filter(s -> s.getVoteNo()[issue] != s.getVoteYes()[issue])
  .count();
System.out.println("Members of Party1 who voted on Issue" +
                                          issue + ": " + c1);

c2 = senators.stream()
  .filter(s -> s.getParty() == "Party2" &&
               s.getVoteNo()[issue] != s.getVoteYes()[issue])
  .count();
System.out.println("Members of Party2 who voted on Issue" +
                                          issue + ": " + c2);
System.out.println("Members of the senate who voted on Issue"
                               + issue + ": " + (c1 + c2));
```

For `Party1`, we used two filters. For `Party2`, we combined them just to show another possible solution. The important point here is to use the filter by a party (`s -> s.getParty() == "Party1"`) first before the filter that selects only those who voted. This way, the second filter is used only for approximately half of the elements. Otherwise, if the filter that selects only those who voted were placed first, it would be applied to all 100 of `Senate` members.

The result looks like this:



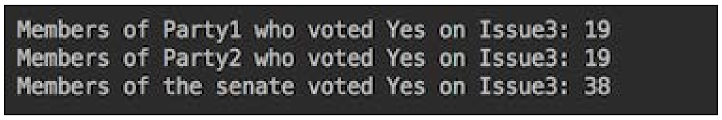
```
Members of Party1 who voted on Issue3: 46
Members of Party2 who voted on Issue3: 36
Members of the senate who voted on Issue3: 82
```

Similarly, we can calculate how many members of each party voted `Yes` on `issue 3`:

```
c1 = senators.stream()
        .filter(s -> s.getParty() == "Party1" &&
                     s.getVoteYes()[issue] == 1)
        .count();
```

```
System.out.println("Members of Party1 who voted Yes on Issue"
                                          + issue + ": " + c1);

c2 = senators.stream()
        .filter(s -> s.getParty() == "Party2" &&
                     s.getVoteYes()[issue] == 1)
        .count();
System.out.println("Members of Party2 who voted Yes on Issue"
                                          + issue + ": " + c2);
System.out.println("Members of the senate voted Yes on Issue"
                                   + issue + ": " + (c1 + c2));
```

The result of the preceding code is as follows:



```
Members of Party1 who voted Yes on Issue3: 19
Members of Party2 who voted Yes on Issue3: 19
Members of the senate voted Yes on Issue3: 38
```

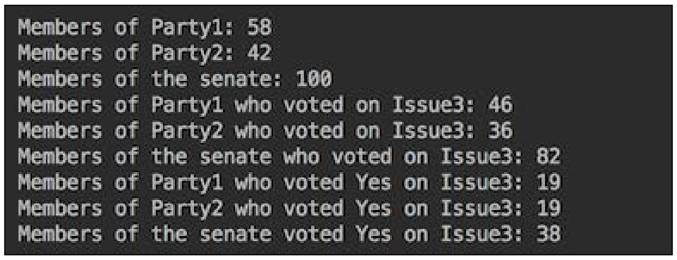
We can refactor the preceding examples by taking advantage of the Java functional programming capability (using lambda expressions) and creating the `countAndPrint()` method:

```
long countAndPrint(List<Senator> senators,
       Predicate<Senator> pred1, Predicate<Senator> pred2,
                                             String prefix) {
    long c = senators.stream().filter(pred1::test)
                              .filter(pred2::test).count();
    System.out.println(prefix + c);
    return c;
}
```

Now all the earlier code can be expressed in a more compact way:

```
int issue = 3;

Predicate<Senator> party1 = s -> s.getParty() == "Party1";
Predicate<Senator> party2 = s -> s.getParty() == "Party2";
Predicate<Senator> voted3 =
       s -> s.getVoteNo()[issue] != s.getVoteYes()[issue];
Predicate<Senator> yes3 = s -> s.getVoteYes()[issue] == 1;

long c1 = countAndPrint(senators, party1, s -> true,
                                   "Members of Party1: ");
long c2 = countAndPrint(senators, party2, s -> true,
                                   "Members of Party2: ");
System.out.println("Members of the senate: " + (c1 + c2));

c1 = countAndPrint(senators, party1, voted3,
   "Members of Party1 who voted on Issue" + issue + ": ");
c2 = countAndPrint(senators, party2, voted3,
   "Members of Party2 who voted on Issue" + issue + ": ");
System.out.println("Members of the senate who voted on Issue"
                                 + issue + ": " + (c1 + c2));

c1 = countAndPrint(senators, party1, yes3,
  "Members of Party1 who voted Yes on Issue" + issue + ": ");
c2 = countAndPrint(senators, party2, yes3,
  "Members of Party2 who voted Yes on Issue" + issue + ": ");
System.out.println("Members of the senate voted Yes on Issue"
                                 + issue + ": " + (c1 + c2));
```

We created four predicates, `party1`, `party2`, `voted3`, and `yes3`, and we used each of them several times as parameters of the `countAndPrint()` method. The output of this code is the same as that of the earlier examples:



```
Members of Party1: 58
Members of Party2: 42
Members of the senate: 100
Members of Party1 who voted on Issue3: 46
Members of Party2 who voted on Issue3: 36
Members of the senate who voted on Issue3: 82
Members of Party1 who voted Yes on Issue3: 19
Members of Party2 who voted Yes on Issue3: 19
Members of the senate voted Yes on Issue3: 38
```

Using the `filter()` method of the `Stream` interface is the most popular way of filtering. But it is possible to use other `Stream` methods to accomplish the same effect.

Using Other Stream Operations for Filtering

Alternatively, or in addition to the basic filtering described in the previous section, other operations (methods of the `Stream` interface) can be used for selection and filtering emitted stream elements.

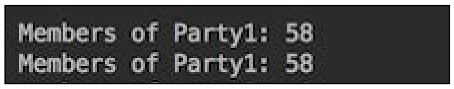
For example, let's use the `flatMap()` method to filter out the members of the Senate by their party membership:

```
long c1 = senators.stream()
        .flatMap(s -> s.getParty() == "Party1" ?
                      Stream.of(s) : Stream.empty())
        .count();
System.out.println("Members of Party1: " + c1);
```

This method takes advantage of the `Stream.of()` (produces a stream of one element) and `Stream.empty()`factory methods (it produces a stream without elements, so nothing is emitted further downstream). Alternatively, the same effect can be achieved using a new factory method (introduced in Java 9) called `Stream.ofNullable()`:

```
c1 = senators.stream().flatMap(s ->
  Stream.ofNullable(s.getParty() == "Party1" ? s : null))
                                                .count();
System.out.println("Members of Party1: " + c1);
```

The `Stream.ofNullable()` method creates a stream of one element if not `null`; otherwise, it creates an empty stream, as in the previous example. Both the preceding code snippets--produce the same output if we run them for the same senate composition:



```
Members of Party1: 58
Members of Party1: 58
```

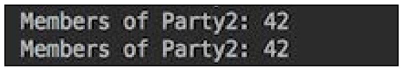
However, the same result can be achieved using a `java.uti.Optional` class that may or may not contain a value. If a value is present (and not `null`), its `isPresent()` method returns `true` and the `get()` method returns the value. Here is how we can use it to filter out the members of one party:

```
long c2 = senators.stream()
  .map(s -> s.getParty() == "Party2" ?
                         Optional.of(s) : Optional.empty())
  .flatMap(o -> o.map(Stream::of).orElseGet(Stream::empty))
  .count();
System.out.println("Members of Party2: " + c2);
```

First, we map (transform) an element (the `Senator` object) to an `Optional` object with or without the value. Next, we use the `flatMap()` method to either generate a stream of a single element or else an empty stream, and then we count the elements that made it through. In Java 9, the `Optional` class acquired a new factory `stream()` method that produces a stream of one element if the `Optional` object carries a non-null value; otherwise, it produces an empty stream. Using this new method, we can rewrite the previous code as follows:

```
long c2 = senators.stream()
  .map(s -> s.getParty() == "Party2" ?
                       Optional.of(s) : Optional.empty())
  .flatMap(Optional::stream)
  .count();
System.out.println("Members of Party2: " + c2);
```

Both the previous examples produce the same output if we run them for the same senate composition:

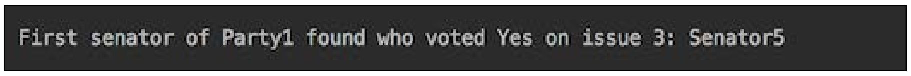


```
Members of Party2: 42
Members of Party2: 42
```

We can apply another kind of filtering when we need to capture the first element emitted by the stream. This means that we terminate the stream after the first element is emitted. For example, let's find the first senator of `Party1`who voted `Yes` on `issue 3`:

```
senators.stream()
  .filter(s -> s.getParty() == "Party1" &&
                          s.getVoteYes()[3] == 1)
  .findFirst()
  .ifPresent(s -> System.out.println("First senator "
         "of Party1 found who voted Yes on issue 3: "
                                 + s.getName()));
```

In the preceding code snippet, we highlighted the `findFirst()` method, which does the described job. It returns the `Optional` object, so we have added another `ifPresent()` operator that is invoked only if the `Optional`object contains a non-null value. The resulting output is as follows:



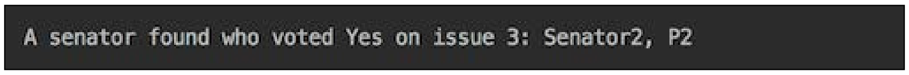
```
First senator of Party1 found who voted Yes on issue 3: Senator5
```

This was exactly what we expected when we seeded data in the `Senate` class.

Similarly, we can use the `findAny()` method to find any `senator` who voted `Yes` on `issue 3`:

```
senators.stream().filter(s -> s.getVoteYes()[3] == 1)
        .findAny()
        .ifPresent(s -> System.out.println("A senator " +
                 "found who voted Yes on issue 3: " + s));
```

The result is also as we expected:



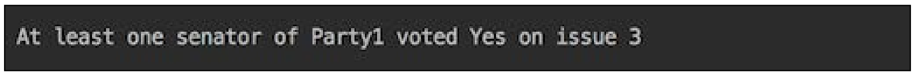
```
A senator found who voted Yes on issue 3: Senator2, P2
```

It is typically (but not necessarily) the first element of the stream. But one should not rely on this assumption, especially in the case of parallel processing.

The `Stream` interface also has three `match` methods that, although they return a Boolean value, can be used for filtering too if the specific object is not required and we only need to establish the fact that such an object exists or not. The names of these methods are `anyMatch()`, `allMatch()`, and `noneMatch()`. Each of them takes a predicate and returns a Boolean. Let's start by demonstrating the `anyMatch()` method. We will use it to find out if there is at least one `senator` of `Party1` who voted `Yes` on `issue 3`:

```
boolean found = senators.stream()
       .anyMatch(s -> (s.getParty() == "Party1" &&
                             s.getVoteYes()[3] == 1));
String res = found ?
  "At least one senator of Party1 voted Yes on issue 3"
  : "Nobody of Party1 voted Yes on issue 3";
System.out.println(res);
```

The result of running the previous code should look like the following:



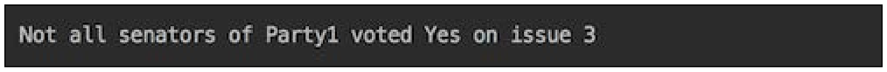
```
At least one senator of Party1 voted Yes on issue 3
```

To demonstrate the `allMatch()` method, we will use it to find out if all the members of `Party1` in the `Senate` class have voted `Yes` on `issue 3`:

```
boolean yes = senators.stream()
    .allMatch(s -> (s.getParty() == "Party1" &&
                           s.getVoteYes()[3] == 1));
```

```
String res = yes ?
  "All senators of Party1 voted Yes on issue 3"
  : "Not all senators of Party1 voted Yes on issue 3";
System.out.println(res);
```

The result of the previous code may look like this:

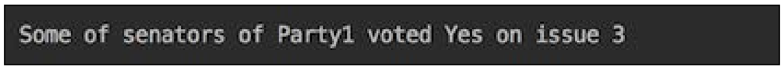


```
Not all senators of Party1 voted Yes on issue 3
```

And the last of the three `match` methods--the `noneMatch()` method--will be used to figure out if some senators of `Party1` have voted `Yes` on `issue 3`:

```
boolean yes = senators.stream()
   .noneMatch(s -> (s.getParty() == "Party1" &&
                        s.getVoteYes()[3] == 1));
String res = yes ?
  "None of the senators of Party1 voted Yes on issue 3"
  : "Some of senators of Party1 voted Yes on issue 3";
System.out.println(res);
```

The result of the earlier example is as follows:



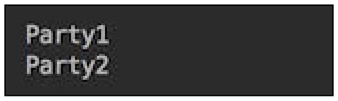
```
Some of senators of Party1 voted Yes on issue 3
```

However, in real life, it could be very different because quite a few issues in the `Senate` class are voted for along party lines.

Yet another type of filtering is required when we need to skip all the duplicate elements in a stream and select only unique ones. The `distinct()` method is designed for the purpose. We will use it to find the names of the parties that have their members in the `Senate` class:

```
senators.stream().map(s -> s.getParty())
        .distinct().forEach(System.out::println);
```

The result, as expected, is as follows:



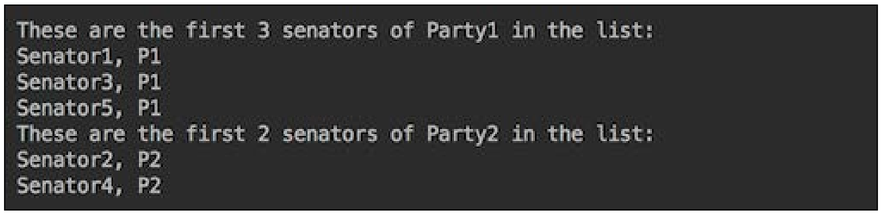
```
Party1
Party2
```

Well, no surprise there?

We can also filter out all the elements of the `stream` except the certain count of the first ones, using the `limit()` method:

```
System.out.println("These are the first 3 senators "
                          + "of Party1 in the list:");
senators.stream()
        .filter(s -> s.getParty() == "Party1")
.limit(3)
        .forEach(System.out::println);

System.out.println("These are the first 2 senators "
                           + "of Party2 in the list:");
senators.stream().filter(s -> s.getParty() == "Party2")
.limit(2)
        .forEach(System.out::println);
```

If you remember how we have set up the first five senators in the list, you could predict that the result will be as follows:

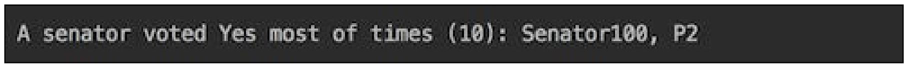


```
These are the first 3 senators of Party1 in the list:
Senator1, P1
Senator3, P1
Senator5, P1
These are the first 2 senators of Party2 in the list:
Senator2, P2
Senator4, P2
```

Now let's find only one element in a stream--the biggest one. To do this, we can use the `max()` method of the `Stream` interface and the `Senate.timeVotedYes()` method (we will apply it on each senator):

```
senators.stream()
   .max(Comparator.comparing(Senate::timesVotedYes))
   .ifPresent(s -> System.out.println("A senator voted "
        + "Yes most of times (" + Senate.timesVotedYes(s)
                                            + "): " + s));
```

In the preceding snippet, we use the result of the `timesVotedYes()` method to select the senator who voted `Yes` most often. You might remember, we have assigned all instances of `Yes` to `Senator100`. Let's see if that would be the result:



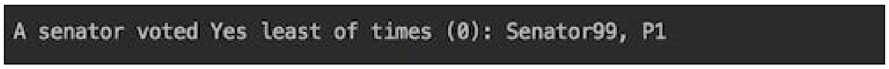
```
A senator voted Yes most of times (10): Senator100, P2
```

Yes, we got `Senator100` filtered as the one who voted `Yes` on all 10 issues.

Similarly, we can find the senator who voted `No` on all 10 issues:

```
senators.stream()
  .min(Comparator.comparing(Senate::timesVotedYes))
  .ifPresent(s -> System.out.println("A senator voted "
       + "Yes least of times (" + Senate.timesVotedYes(s)
                                            + "): " + s));
```

We expect it to be `Senator99`, and here is the result:



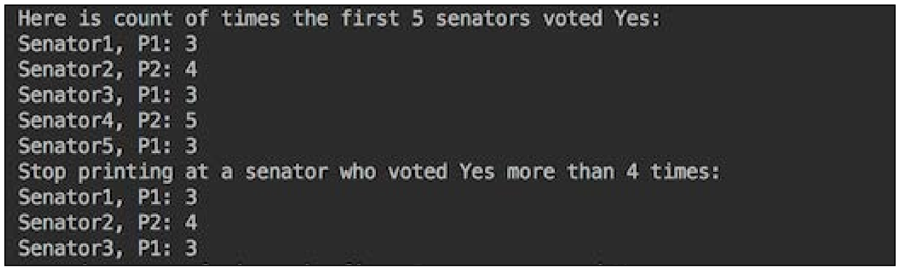
```
A senator voted Yes least of times (0): Senator99, P1
```

That's why we hardcoded several stats in the `Senate` class, so we can verify that our queries work correctly.

As the last two methods can help us with filtering, we will demonstrate the `takeWhile()` and `dropWhile()`methods introduced in JDK 9. We will first print the data of all the first five senators and then use the `takeWhile()`method to print the first senators until we encounter the one who voted `Yes` more than four times, and then stop printing:

```
System.out.println("Here is count of times the first "
                            + "5 senators voted Yes:");
senators.stream().limit(5)
  .forEach(s -> System.out.println(s + ": "
                           + Senate.timesVotedYes(s)));
System.out.println("Stop printing at a senator who "
                     + "voted Yes more than 4 times:");
senators.stream().limit(5)
        .takeWhile(s -> Senate.timesVotedYes(s) < 5)
        .forEach(s -> System.out.println(s + ": "
                           + Senate.timesVotedYes(s)));
```

The result for the previous code is as follows:

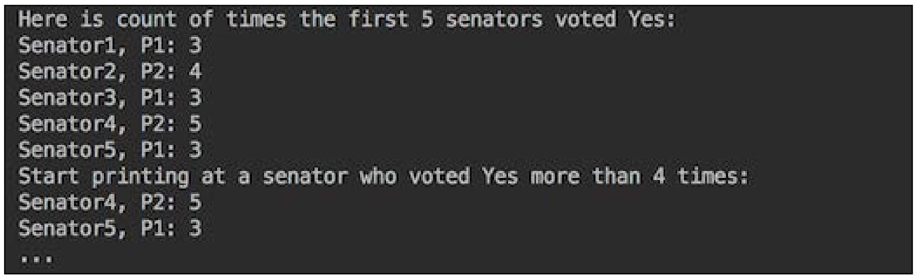


```
Here is count of times the first 5 senators voted Yes:
Senator1, P1: 3
Senator2, P2: 4
Senator3, P1: 3
Senator4, P2: 5
Senator5, P1: 3
Stop printing at a senator who voted Yes more than 4 times:
Senator1, P1: 3
Senator2, P2: 4
Senator3, P1: 3
```

The `dropWhile()` method can be used for the opposite effect, that is, to filter away, to skip the first senators until we encounter the one who voted `Yes` more than four times, then continue printing all the rest of the senators:

```
System.out.println("Here is count of times the first "
                             + "5 senators voted Yes:");
senators.stream().limit(5)
        .forEach(s -> System.out.println(s + ": "
                            + Senate.timesVotedYes(s)));
System.out.println("Start printing at a senator who "
                      + "voted Yes more than 4 times:");
senators.stream().limit(5)
        .dropWhile(s -> Senate.timesVotedYes(s) < 5)
        .forEach(s -> System.out.println(s + ": "
                            + Senate.timesVotedYes(s)));
System.out.println("...");
```

The result will be as follows:



```
Here is count of times the first 5 senators voted Yes:
Senator1, P1: 3
Senator2, P2: 4
Senator3, P1: 3
Senator4, P2: 5
Senator5, P1: 3
Start printing at a senator who voted Yes more than 4 times:
Senator4, P2: 5
Senator5, P1: 3
...
```

This concludes our demonstration of the ways in which a stream of elements can be filtered. We hope you have learned enough to be able to find a solution for any of your filtering needs. Nevertheless, we encourage you to study and experiment with the Stream API on your own, so you can retain what you have learned so far and acquire your own view on the rich APIs of Java 9.

Stack-Walking APIs

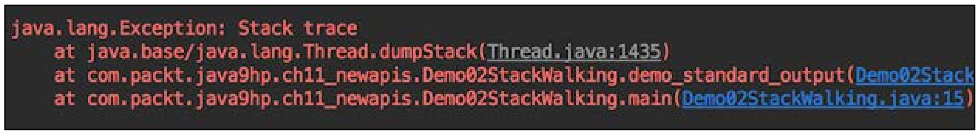
Exceptions do happen, especially during development or the period of software stabilization. But in a big complex system, the chance of getting an exception is possible even in production, especially when several third-party systems are brought together and the need arises to analyze the stack trace programmatically in order to apply an automatic correction. In this section, we will discuss how it can be done.

Stack Analysis before Java 9

The traditional reading of the stack trace, using objects of the `java.lang.Thread` and `java.lang.Throwable`classes, was accomplished by capturing it from the standard output. For example, we can include this line in any section of the code:

```
Thread.currentThread().dumpStack();
```

The previous line will produce the following output:

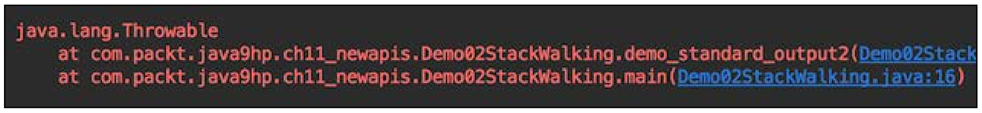


```
java.lang.Exception: Stack trace
    at java.base/java.lang.Thread.dumpStack(Thread.java:1435)
    at com.packt.java9hp.ch11_newapis.Demo02StackWalking.demo_standard_output(Demo02Stack
    at com.packt.java9hp.ch11_newapis.Demo02StackWalking.main(Demo02StackWalking.java:15)
```

Similarly, we can include this line in the code:

```
new Throwable().printStackTrace();
```

The output will then look like this:



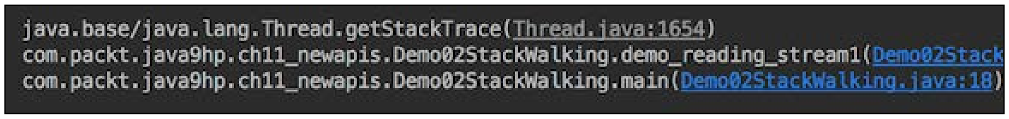
```
java.lang.Throwable
    at com.packt.java9hp.ch11_newapis.Demo02StackWalking.demo_standard_output2(Demo02Stack
    at com.packt.java9hp.ch11_newapis.Demo02StackWalking.main(Demo02StackWalking.java:16)
```

This output can be captured, read, and analyzed programmatically, but requires quite a bit of custom code writing.

JDK 8 made this easier via the usage of streams. Here is the code that allows reading the stack trace from the stream:

```
Arrays.stream(Thread.currentThread().getStackTrace())
        .forEach(System.out::println);
```

The previous line produces the following output:

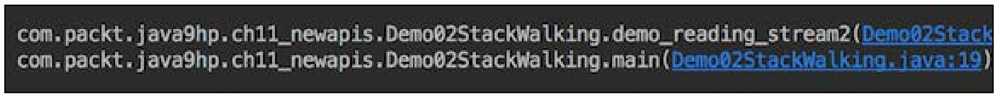


```
java.base/java.lang.Thread.getStackTrace(Thread.java:1654)
com.packt.java9hp.ch11_newapis.Demo02StackWalking.demo_reading_stream1(Demo02Stack
com.packt.java9hp.ch11_newapis.Demo02StackWalking.main(Demo02StackWalking.java:18)
```

Alternatively, we could use this code:

```
Arrays.stream(new Throwable().getStackTrace())
        .forEach(System.out::println);
```

The output of the previous code shows the stack trace in a similar way:



```
com.packt.java9hp.ch11_newapis.Demo02StackWalking.demo_reading_stream2(Demo02Stack
com.packt.java9hp.ch11_newapis.Demo02StackWalking.main(Demo02StackWalking.java:19)
```

If, for example, you would like to find the fully qualified name of the caller class, you can use one of these approaches:

```
new Throwable().getStackTrace()[1].getClassName();

Thread.currentThread().getStackTrace()[2].getClassName();
```

Such coding is possible because the `getStackTrace()` method returns an array of objects of the `java.lang.StackTraceElement` class, each representing a stack frame in a stack trace. Each object carries stack trace information accessible by the `getFileName()`, `getClassName()`, `getMethodName()`, and `getLineNumber()` methods.

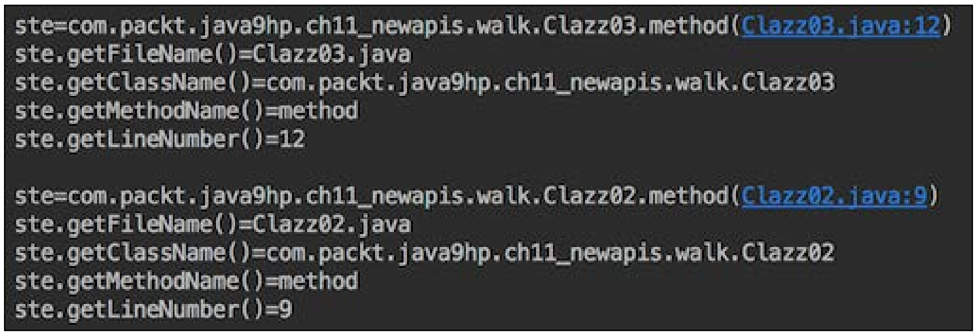
To demonstrate how it works, we have created three classes, `Clazz01`, `Clazz02`, and `Clazz03`, that call each other:

```
public class Clazz01 {
  public void method(){ new Clazz02().method(); }
}
public class Clazz02 {
  public void method(){ new Clazz03().method(); }
}
public class Clazz03 {
  public void method(){
    Arrays.stream(Thread.currentThread()
                        .getStackTrace()).forEach(ste -> {
      System.out.println();
      System.out.println("ste=" + ste);
      System.out.println("ste.getFileName()=" +
                               ste.getFileName());
      System.out.println("ste.getClassName()=" +
                              ste.getClassName());
      System.out.println("ste.getMethodName()=" +
                             ste.getMethodName());
      System.out.println("ste.getLineNumber()=" +
                             ste.getLineNumber());
    });
  }
}
```

Now, let's call the `method()` method of `Clazz01`:

```
public class Demo02StackWalking {
    public static void main(String... args) {
        demo_walking();
    }
    private static void demo_walking(){
        new Clazz01().method();
    }
}
```

Here are two (the second and the third) of the six stack trace frames printed out by the preceding code:



```
ste=com.packt.java9hp.ch11_newapis.walk.Clazz03.method(Clazz03.java:12)
ste.getFileName()=Clazz03.java
ste.getClassName()=com.packt.java9hp.ch11_newapis.walk.Clazz03
ste.getMethodName()=method
ste.getLineNumber()=12

ste=com.packt.java9hp.ch11_newapis.walk.Clazz02.method(Clazz02.java:9)
ste.getFileName()=Clazz02.java
ste.getClassName()=com.packt.java9hp.ch11_newapis.walk.Clazz02
ste.getMethodName()=method
ste.getLineNumber()=9
```

In principle, every called class has access to this information. But to find out which class called the current class may not be so easy because you need to figure out which frame represents the caller. Also, in order to provide this info, JVM captures the entire stack (except for the hidden stack frames), and it may affect performance.

That was the motivation for introducing the `java.lang.StackWalker` class, its nested `Option` class, and the `StackWalker.StackFrame` interface in JDK 9.

New Better Way to Walk the Stack

The `StackWalker` class has four `getInstance()` static factory methods:

- `getInstance()`: This returns a `StackWalker` class instance configured to skip all hidden frames and the caller class reference
- `getInstance(StackWalker.Option option)`: This creates a `StackWalker` class instance with the given option specifying the stack frame information it can access

- `getInstance(Set<StackWalker.Option> options)`: This creates a `StackWalker` class instance with the given set of options
- `getInstance(Set<StackWalker.Option> options, int estimatedDepth)`: This allows you to pass in the `estimatedDepth` parameter that specifies the estimated number of stack frames this instance will traverse so that the Java machine can allocate the appropriate buffer size it might need

The value passed as an option can be one of the following:

- `StackWalker.Option.RETAIN_CLASS_REFERENCE`
- `StackWalker.Option.SHOW_HIDDEN_FRAMES`
- `StackWalker.Option.SHOW_REFLECT_FRAMES`

The other three methods of the `StackWalker` class are as follows:

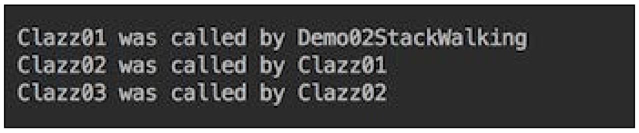
- `T walk(Function<Stream<StackWalker.StackFrame>, T> function)`: This applies the passed in function to the stream of stack frames, the first frame representing the method that called this `walk()` method
- `void forEach(Consumer<StackWalker.StackFrame> action)`: This performs the passed in action on each element (of the `StalkWalker.StackFrame` interface type) of the stream of the current thread
- `Class<?> getCallerClass()`: This gets objects of the `Class` class of the caller class

As you can see, it allows much more straightforward stack trace analysis. Let's modify our demo classes using the following code and access the caller name in one line:

```
public class Clazz01 {
  public void method(){
    System.out.println("Clazz01 was called by " +
      StackWalker.getInstance(StackWalker
        .Option.RETAIN_CLASS_REFERENCE)
        .getCallerClass().getSimpleName());
    new Clazz02().method();
  }
}
public class Clazz02 {
  public void method(){
    System.out.println("Clazz02 was called by " +
      StackWalker.getInstance(StackWalker
        .Option.RETAIN_CLASS_REFERENCE)
```

```
          .getCallerClass().getSimpleName());
      new Clazz03().method();
    }
  }
  public class Clazz03 {
    public void method(){
      System.out.println("Clazz01 was called by " +
        StackWalker.getInstance(StackWalker
          .Option.RETAIN_CLASS_REFERENCE)
          .getCallerClass().getSimpleName());
    }
  }
```

The previous code will produce this output:

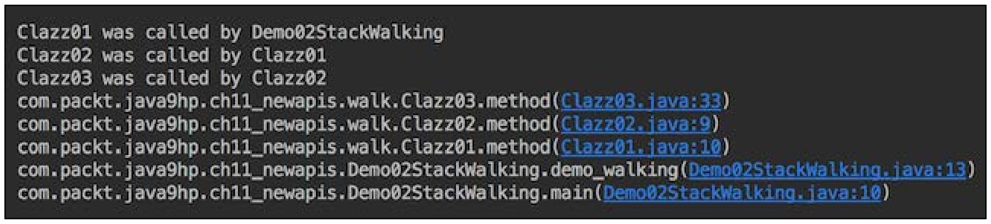


```
Clazz01 was called by Demo02StackWalking
Clazz02 was called by Clazz01
Clazz03 was called by Clazz02
```

You can appreciate the simplicity of the solution. If we need to see the entire stack trace, we can add the following line to the code in `Clazz03`:

```
StackWalker.getInstance().forEach(System.out::println);
```

The resulting output will be as follows:



```
Clazz01 was called by Demo02StackWalking
Clazz02 was called by Clazz01
Clazz03 was called by Clazz02
com.packt.java9hp.ch11_newapis.walk.Clazz03.method(Clazz03.java:33)
com.packt.java9hp.ch11_newapis.walk.Clazz02.method(Clazz02.java:9)
com.packt.java9hp.ch11_newapis.walk.Clazz01.method(Clazz01.java:10)
com.packt.java9hp.ch11_newapis.Demo02StackWalking.demo_walking(Demo02StackWalking.java:13)
com.packt.java9hp.ch11_newapis.Demo02StackWalking.main(Demo02StackWalking.java:10)
```

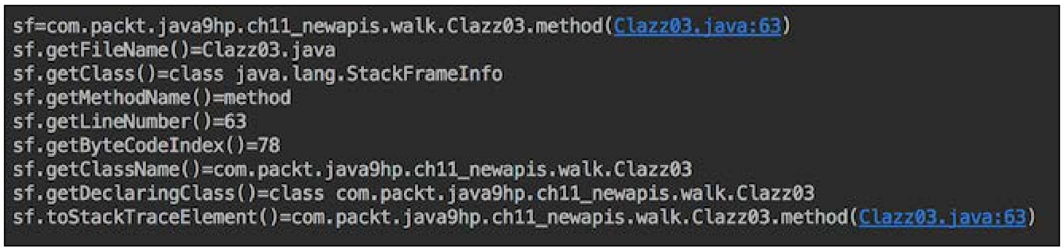
Again, with only one line of code, we have achieved much more readable output. We could achieve the same result by using the `walk()` method:

```
StackWalker.getInstance().walk(sf -> {
  sf.forEach(System.out::println); return null;
});
```

Instead of just printing `StackWalker.StackFrame`, we also could run a deeper analysis on it, if need be, using its API, which is more extensive than the API of `java.lang.StackTraceElement`. Let's run the code example that prints every stack frame and its information:

```
StackWalker stackWalker =
   StackWalker.getInstance(Set.of(StackWalker
                   .Option.RETAIN_CLASS_REFERENCE), 10);
stackWalker.forEach(sf -> {
    System.out.println();
    System.out.println("sf="+sf);
    System.out.println("sf.getFileName()=" +
                                   sf.getFileName());
    System.out.println("sf.getClass()=" + sf.getClass());
    System.out.println("sf.getMethodName()=" +
                                 sf.getMethodName());
    System.out.println("sf.getLineNumber()=" +
                                 sf.getLineNumber());
    System.out.println("sf.getByteCodeIndex()=" +
                              sf.getByteCodeIndex());
    System.out.println("sf.getClassName()=" +
                                  sf.getClassName());
    System.out.println("sf.getDeclaringClass()=" +
                             sf.getDeclaringClass());
    System.out.println("sf.toStackTraceElement()=" +
                           sf.toStackTraceElement());
});
```

The output of the previous code is as follows:



```
sf=com.packt.java9hp.ch11_newapis.walk.Clazz03.method(Clazz03.java:63)
sf.getFileName()=Clazz03.java
sf.getClass()=class java.lang.StackFrameInfo
sf.getMethodName()=method
sf.getLineNumber()=63
sf.getByteCodeIndex()=78
sf.getClassName()=com.packt.java9hp.ch11_newapis.walk.Clazz03
sf.getDeclaringClass()=class com.packt.java9hp.ch11_newapis.walk.Clazz03
sf.toStackTraceElement()=com.packt.java9hp.ch11_newapis.walk.Clazz03.method(Clazz03.java:63)
```

Note the `StackFrameInfo` class that implements the `StackWalker.StackFrame` interface and actually does the job. The API also allows converting back to the familiar `StackTraceElement` object for backward compatibility and for the enjoyment of those who are used to it and do not want to change their code and habits.

In contrast, with the full stack trace generated and stored in the array in the memory (like in the case of the traditional stack trace implementation), the `StackWalker` class brings only the requested elements. This is another motivation for its introduction in addition to the demonstrated simplicity of use. More details about the `StackWalker` class API and its usage can be found at `https://docs.oracle.com/javase/9/docs/api/java/lang/StackWalker.html`.

Convenience Factory Methods for Collections

With the introduction of functional programming in Java, the interest in and need for immutable objects increased. The functions passed into the methods may be executed in substantially different contexts than the one they were created in, so the need to decrease the chances of unexpected side effects made the case for immutability stronger. Besides, the Java way of creating an unmodifiable collection was quite verbose anyway, so the issue was addressed in Java 9. Here is an example of the code that creates an immutable collection of the `Set` interface in Java 8:

```
Set<String> set = new HashSet<>();
set.add("Life");
set.add("is");
set.add("good!");
set = Collections.unmodifiableSet(set);
```

After one does it several times, the need for a convenience method comes up naturally as the basic refactoring consideration that always lingers in the background thinking of any software professional. In Java 8, the previous code could be changed to the following:

```
Set<String> immutableSet =
  Collections.unmodifiableSet(new HashSet<>(Arrays
                        .asList("Life", "is", "good!")));
```

Alternatively, if streams are your friends, you could write the following:

```
Set<String> immutableSet = Stream.of("Life","is","good!")
  .collect(Collectors.collectingAndThen(Collectors.toSet(),
                             Collections::unmodifiableSet));
```

Another version of the previous code is as follows:

```
Set<String> immutableSet =
  Collections.unmodifiableSet(Stream.of("Life","is","good!")
                             .collect(Collectors.toSet()));
```

However, it has more boilerplate code than the values you are trying to encapsulate. So, in Java 9, a shorter version of the previous code became possible:

```
Set<String> immutableSet = Set.of("Life","is","good!");
```

Similar factories were introduced to generate immutable collections of `List` interfaces and `Map` interfaces:

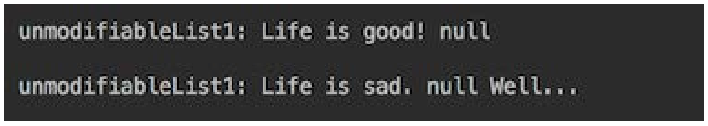
```
List<String> immutableList = List.of("Life","is","good!");

Map<Integer,String> immutableMap1 =
                   Map.of(1, "Life", 2, "is", 3, "good!");

Map<Integer,String> immutableMap2 =
       Map.ofEntries(entry(1, "Life "), entry(2, "is"),
                                        entry(3, "good!");

Map.Entry<Integer,String> entry1 = Map.entry(1,"Life");
Map.Entry<Integer,String> entry2 = Map.entry(2,"is");
Map.Entry<Integer,String> entry3 = Map.entry(3,"good!");
Map<Integer,String> immutableMap3 =
                    Map.ofEntries(entry1, entry2, entry3);
```

Why New Factory Methods?

The ability to express the same functionality in more compact manner is very helpful, but it would probably not be enough motivation to introduce these new factories. It was much more important to address the weakness of the existing implementation of `Collections.unmodifiableList()`, `Collections.unmodifiableSet()`, and `Collections.unmodifiableMap()`. Although the collections created using these methods throw an `UnsupportedOperationException` class when you try to modify or add/remove their elements, they are just wrappers around the traditional modifiable collections and can thus be susceptible to modifications, depending on the way you construct them. Let's walk through examples to illustrate the point. By the way, another weakness of the existing unmodifiable implementation is that it does not change how the source collection is constructed, so the difference between `List`, `Set`, and `Map`--the ways in which they can be constructed--remains in place, which may be a source of bugs or even frustration when a programmer uses them. The new factory methods address this issue too, providing a more unified approach using the `of()` factory method (and the additional `ofEntries()` method for `Map`) only. Having said that, let's get back to the examples. Look at the following code snippet:

```
List<String> list = new ArrayList<>();
list.add("unmodifiableList1: Life");
list.add(" is");
list.add(" good! ");
list.add(null);
list.add("\n\n");
List<String> unmodifiableList1 =
                      Collections.unmodifiableList(list);
//unmodifiableList1.add(" Well..."); //throws exception
//unmodifiableList1.set(2, " sad."); //throws exception
unmodifiableList1.stream().forEach(System.out::print);

list.set(2, " sad. ");
list.set(4, " ");
list.add("Well...\n\n");
unmodifiableList1.stream().forEach(System.out::print);
```

Attempts of direct modification of the elements of `unmodifiableList1` lead to `UnsupportedOperationException`. Nevertheless, we can modify them via the underlying `list` object. If we run the previous example, the output will be as follows:

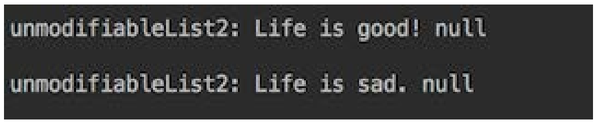


```
unmodifiableList1: Life is good! null

unmodifiableList1: Life is sad. null Well...
```

Even if we use `Arrays.asList()` for the source list creation, it will only protect the created collection from adding a new element, but not from modifying the existing one. Here is a code example:

```
List<String> list2 =
           Arrays.asList("unmodifiableList2: Life",
                        " is", " good! ", null, "\n\n");
List<String> unmodifiableList2 =
                    Collections.unmodifiableList(list2);
//unmodifiableList2.add(" Well..."); //throws exception
//unmodifiableList2.set(2, " sad."); //throws exception
unmodifiableList2.stream().forEach(System.out::print);

list2.set(2, " sad. ");
//list2.add("Well...\n\n");  //throws exception
unmodifiableList2.stream().forEach(System.out::print);
```

If we run the previous code, the output will be as follows:



```
unmodifiableList2: Life is good! null

unmodifiableList2: Life is sad. null
```

We also included a `null` element to demonstrate how the existing implementation treats them, because, by contrast, the new factories of immutable collections do not allow `null` to be included. By the way, they do not allow duplicate elements in `Set` either (while the existing implementation just ignores them), but we will demonstrate this aspect later while using the new factory methods in code examples.

To be fair, there is a way to create a truly immutable collection of `List` interfaces with the existing implementation too. Look at the following code:

```
List<String> immutableList1 =
        Collections.unmodifiableList(new ArrayList<>() {{
            add("immutableList1: Life");
            add(" is");
            add(" good! ");
            add(null);
            add("\n\n");
        }});
//immutableList1.set(2, " sad.");     //throws exception
//immutableList1.add("Well...\n\n");  //throws exception
immutableList1.stream().forEach(System.out::print);
```

Another way to create an immutable list is as follows:

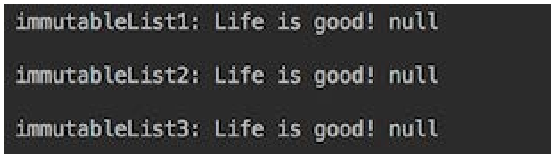
```
List<String> immutableList2 =
  Collections.unmodifiableList(Stream
   .of("immutableList2: Life"," is"," good! ",null,"\n\n")
   .collect(Collectors.toList()));
//immutableList2.set(2, " sad.");     //throws exception
//immutableList2.add("Well...\n\n");  //throws exception
immutableList2.stream().forEach(System.out::print);
```

The following is a variation of the earlier code:

```
List<String> immutableList3 =
  Stream.of("immutableList3: Life",
                            " is"," good! ",null,"\n\n")
  .collect(Collectors.collectingAndThen(Collectors.toList(),
                            Collections::unmodifiableList));
```

```
//immutableList3.set(2, " sad.");      //throws exception
//immutableList3.add("Well...\n\n");  //throws exception
immutableList3.stream().forEach(System.out::print);
```

If we run the previous three examples, we will see the following output:



```
immutableList1: Life is good! null

immutableList2: Life is good! null

immutableList3: Life is good! null
```

Note that although we cannot modify the content of these lists, we can put `null` in them.

The situation with `Set` is quite similar to what we have seen with the lists earlier. Here is the code that shows how an unmodifiable collection of `Set` interfaces can be modified:

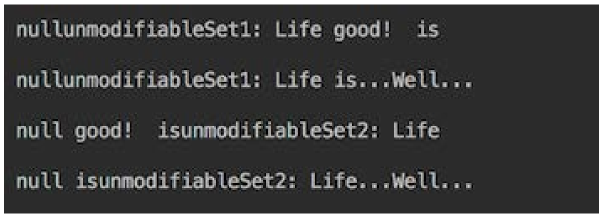
```
Set<String> set = new HashSet<>();
set.add("unmodifiableSet1: Life");
set.add(" is");
set.add(" good! ");
set.add(null);
Set<String> unmodifiableSet1 =
                       Collections.unmodifiableSet(set);
//unmodifiableSet1.remove(" good! "); //throws exception
//unmodifiableSet1.add("...Well..."); //throws exception
unmodifiableSet1.stream().forEach(System.out::print);
System.out.println("\n");

set.remove(" good! ");
set.add("...Well...");
unmodifiableSet1.stream().forEach(System.out::print);
System.out.println("\n");
```

The resulting collection of `Set` interfaces can be modified even if we convert the original collection from an array to a list and then to a set, as follows:

```
Set<String> set2 =
   new HashSet<>(Arrays.asList("unmodifiableSet2: Life",
                             " is", " good! ", null));
Set<String> unmodifiableSet2 =
                       Collections.unmodifiableSet(set2);
//unmodifiableSet2.remove(" good! "); //throws exception
//unmodifiableSet2.add("...Well..."); //throws exception
```

```
unmodifiableSet2.stream().forEach(System.out::print);
System.out.println("\n");

set2.remove(" good! ");
set2.add("...Well...");
unmodifiableSet2.stream().forEach(System.out::print);
System.out.println("\n");
```

Here is the output of running the previous two examples:



```
nullunmodifiableSet1: Life good!  is

nullunmodifiableSet1: Life is...Well...

null good!  isunmodifiableSet2: Life

null isunmodifiableSet2: Life...Well...
```

If you have not worked with sets in Java 9, you may be surprised to see the unusually messed up order of the set elements in the output. In fact, it is another new feature of set and maps introduced in JDK 9. In the past, `Set` and `Map` implementations did not guarantee to preserve the elements' order. But more often than not, the order was preserved and some programmers wrote code that relied on it, thus introducing an annoyingly inconsistent and not easily reproducible defect into an application. The new `Set` and `Map` implementations change the order more often, if not at every new run of the code. This way, it exposes potential defects early in development and decreases the chance of its propagation into production.

Similar to the lists, we can create immutable sets even without using Java 9's new immutable set factory. One way to do it is as follows:

```
Set<String> immutableSet1 =
     Collections.unmodifiableSet(new HashSet<>() {{
            add("immutableSet1: Life");
            add(" is");
            add(" good! ");
            add(null);
        }});
//immutableSet1.remove(" good! "); //throws exception
//immutableSet1.add("...Well..."); //throws exception
immutableSet1.stream().forEach(System.out::print);
System.out.println("\n");
```

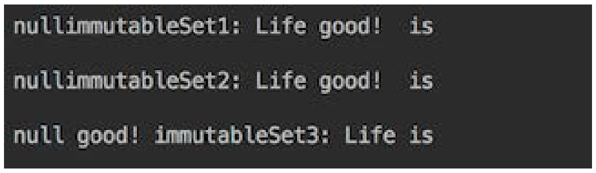
Also, as in the case with lists, here is another way to do it:

```
Set<String> immutableSet2 =
     Collections.unmodifiableSet(Stream
        .of("immutableSet2: Life"," is"," good! ", null)
                            .collect(Collectors.toSet()));
//immutableSet2.remove(" good!");  //throws exception
//immutableSet2.add("...Well..."); //throws exception
immutableSet2.stream().forEach(System.out::print);
System.out.println("\n");
```

Another variant of the previous code is as follows:

```
Set<String> immutableSet3 =
  Stream.of("immutableSet3: Life"," is"," good! ", null)
  .collect(Collectors.collectingAndThen(Collectors.toSet(),
                             Collections::unmodifiableSet));
//immutableList5.set(2, "sad.");  //throws exception
//immutableList5.add("Well...");  //throws exception
immutableSet3.stream().forEach(System.out::print);
System.out.println("\n");
```

If we run all three examples of creating an immutable collection of `iSet` interfaces that we have just introduced, the result would be as follows:



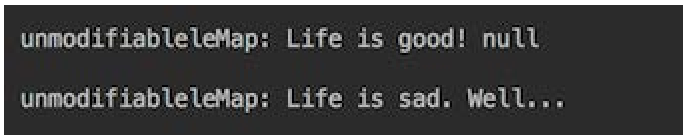
```
nullimmutableSet1: Life good!  is

nullimmutableSet2: Life good!  is

null good! immutableSet3: Life is
```

With `Map` interfaces, we were able to come up with only one way to modify the `unmodifiableMap` object:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "unmodifiableleMap: Life");
map.put(2, " is");
map.put(3, " good! ");
map.put(4, null);
map.put(5, "\n\n");
Map<Integer, String> unmodifiableleMap =
                       Collections.unmodifiableMap(map);
//unmodifiableleMap.put(3, " sad.");   //throws exception
//unmodifiableleMap.put(6, "Well..."); //throws exception
unmodifiableleMap.values().stream()
```

```
                                    .forEach(System.out::print);
    map.put(3, " sad. ");
    map.put(4, "");
    map.put(5, "");
    map.put(6, "Well...\n\n");
    unmodifiableleMap.values().stream()
                                    .forEach(System.out::print);
```

The output of the previous code is as follows:



```
unmodifiableleMap: Life is good! null

unmodifiableleMap: Life is sad. Well...
```

We found four ways to create an immutable collection of `Map` interfaces without using Java 9 enhancements. Here is the first example:

```
    Map<Integer, String> immutableMap1 =
            Collections.unmodifiableMap(new HashMap<>() {{
                put(1, "immutableMap1: Life");
                put(2, " is");
                put(3, " good! ");
                put(4, null);
                put(5, "\n\n");
            }});
    //immutableMap1.put(3, " sad. ");   //throws exception
    //immutableMap1.put(6, "Well...");  //throws exception
    immutableMap1.values().stream().forEach(System.out::print);
```

The second example has a bit of a complication:

```
    String[][] mapping =
           new String[][] {{"1", "immutableMap2: Life"},
                           {"2", " is"}, {"3", " good! "},
                              {"4", null}, {"5", "\n\n"}};

    Map<Integer, String> immutableMap2 =
      Collections.unmodifiableMap(Arrays.stream(mapping)
        .collect(Collectors.toMap(a -> Integer.valueOf(a[0]),
                               a -> a[1] == null? "" : a[1])));
    immutableMap2.values().stream().forEach(System.out::print);
```

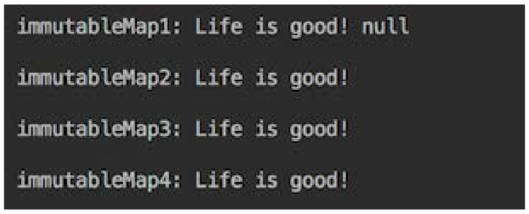
We tried first to use `Collectors.toMap(a -> Integer.valueOf(a[0]), a -> a[1])`, but the `toMap()` method uses the `merge()` functions which does not allow `null` as a value. So, we had to add a check for `null` and replace it with an empty `String` value. This, in effect, brought us to the next version of the previous code snippet--without a `null` value in the source array:

```
String[][] mapping =
    new String[][]{{"1", "immutableMap3: Life"},
       {"2", " is"}, {"3", " good! "}, {"4", "\n\n"}};
Map<Integer, String> immutableMap3 =
   Collections.unmodifiableMap(Arrays.stream(mapping)
     .collect(Collectors.toMap(a -> Integer.valueOf(a[0]),
a -> a[1])));
//immutableMap3.put(3, " sad.");   //throws Exception
//immutableMap3.put(6, "Well..."); //throws exception
immutableMap3.values().stream().forEach(System.out::print);
```

A variant of the previous code is as follows:

```
mapping[0][1] = "immutableMap4: Life";
Map<Integer, String> immutableMap4 = Arrays.stream(mapping)
           .collect(Collectors.collectingAndThen(Collectors
             .toMap(a -> Integer.valueOf(a[0]), a -> a[1]),
                             Collections::unmodifiableMap));
//immutableMap4.put(3, " sad.");    //throws exception
//immutableMap4.put(6, "Well...");  //throws exception
immutableMap4.values().stream().forEach(System.out::print);
```

After we run all the four last examples, the output is as follows:



```
immutableMap1: Life is good! null

immutableMap2: Life is good!

immutableMap3: Life is good!

immutableMap4: Life is good!
```

With that revision of the existing collections implementations, we can now discuss and appreciate the new factory methods of collections in Java 9.

The New Factory Methods in Action

After revisiting the existing methods of collection creation, we can now review and enjoy the related API introduced in Java 9. As in a previous section, we start with the `List` interface. Here is how simple and consistent the immutable list creation can be using the new `List.of()` factory method:

```
List<String> immutableList =
  List.of("immutableList: Life",
      " is", " is", " good!\n\n"); //, null);
//immutableList.set(2, "sad.");    //throws exception
//immutableList.add("Well...");    //throws exception
immutableList.stream().forEach(System.out::print);
```

As you can see from the previous code comments, the new factory method does not allow including `null` as the list value.

The `immutableSet` creation looks similar to this:

```
Set<String> immutableSet =
    Set.of("immutableSet: Life", " is", " good!");
                                   //, " is" , null);
//immutableSet.remove(" good!\n\n");  //throws exception
//immutableSet.add("...Well...\n\n"); //throws exception
immutableSet.stream().forEach(System.out::print);
System.out.println("\n");
```

As you can see from the previous code comments, the `Set.of()` factory method does not allow adding `null` or a duplicate element when creating an immutable collection of `Set` interfaces.

The immutable collection of `Map` interfaces has similar format too:

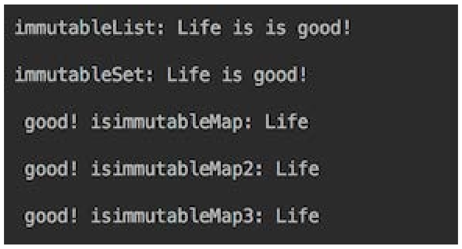
```
Map<Integer, String> immutableMap =
   Map.of(1</span>, "immutableMap: Life", 2, " is", 3, " good!");
                                 //, 4, null);
//immutableMap.put(3, " sad.");    //throws exception
//immutableMap.put(4, "Well...");  //throws exception
immutableMap.values().stream().forEach(System.out::print);
System.out.println("\n");
```

The `Map.of()` method does not allow `null` as a value either. Another feature of the `Map.of()` method is that it allows a compile-time check of the element type, which decreases the chances of a runtime problem.

For those who prefer more compact code, here is another way to express the same functionality:

```
Map<Integer, String> immutableMap3 =
            Map.ofEntries(entry(1, "immutableMap3: Life"),
                     entry(2, " is"), entry(3, " good!"));
immutableMap3.values().stream().forEach(System.out::print);
System.out.println("\n");
```

And here is the output if we run all the previous examples of the usage of the new factory methods:



```
immutableList: Life is is good!

immutableSet: Life is good!

 good! isimmutableMap: Life

 good! isimmutableMap2: Life

 good! isimmutableMap3: Life
```

As we mentioned already, the ability to have immutable collections, including empty ones, is very helpful for functional programming as this feature makes sure that such a collection cannot be modified as a side effect and cannot introduce unexpected and difficult to trace defects. The full variety of the new factories methods includes up to 10 explicit entries plus one with an arbitrary number of elements. Here's how it looks for `List` interface:

```
static <E> List<E> of()
static <E> List<E> of(E e1)
static <E> List<E> of(E e1, E e2)
static <E> List<E> of(E e1, E e2, E e3)
static <E> List<E> of(E e1, E e2, E e3, E e4)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8,
E e9)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8,
E e9, E e10)
static <E> List<E> of(E... elements)
```

The `Set` factory methods look similar:

```
static <E> Set<E> of()
static <E> Set<E> of(E e1)
static <E> Set<E> of(E e1, E e2)
static <E> Set<E> of(E e1, E e2, E e3)
static <E> Set<E> of(E e1, E e2, E e3, E e4)
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5)
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6)
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E
e9)
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E
e9, E e10)
static <E> Set<E> of(E... elements)
```

Also, the `Map` factory methods follow suit:

```
static <K,V> Map<K,V> of()
static <K,V> Map<K,V> of(K k1, V v1)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V
v4)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V
v4, K k5, V   v5
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V
v4, K k5, V v5, K k6, V v6)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V
v4, K k5, V v5, K k6, V v6, K k7, V v7
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V
v4, K k5, V v5, K k6, V v6, K k7, V v7,
K k8, V v8)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V
v4, K k5, V v5, K k6, V v6, K k7, V v7,
K k8, V v8, K k9, V v9)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V
v4, K k5, V v5, K k6, V v6, K k7, V v7,
K k8, V v8, K k9, V v9, K k10, V v10)
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>...
entries
```

The decision not to add new interfaces for immutable collections left them susceptible to causing occasional confusion when programmers assumed they could call `add()` or `put()` on them. Such an assumption, if not tested, will cause a runtime error that throws an `UnsupportedOperationException`. Despite this potential pitfall, the new factory methods for immutable collection creation are very useful additions to Java.

CompletableFuture in Support of Asynchronous Processing

The `java.util.concurrent.CompletableFuture<T>` class was first introduced in Java 8. It is the next level of asynchronous call control over `java.util.concurrent.Future<T>` interface. It actually implements `Future`, as well as `java.util.concurrent.CompletionStage<T>`. In Java 9, `CompletableFuture` was enhanced by adding new factory methods, support for delays and timeouts, and improved subclassing--we will discuss these features in more details in the sections to follow. But first, let's have an overview of the `CompletableFuture` API.

The CompletableFuture API Overview

The `CompletableFuture` API consists of more than 70 methods, 38 of which are implementations of the `CompletionStage` interface, and five are the implementations of `Future`. Because the `CompletableFuture`class implements the `Future` interface, it can be treated as `Future` and will not break the existing functionality based on the `Future` API.

So, the bulk of the API comes from `CompletionStage`. Most of the methods return `CompletableFuture` (in the `CompletionStage` interface, they return `CompletionStage`, but they are converted to `CompletableFuture`when implemented in `CompletableFuture` class), which means that they allow chaining the operations similar to how the `Stream` methods do when only one element goes through a pipe. Each method has a signature that accepts a function. Some methods accept `Function<T,U>`, which is going to be applied to the passed-in value `T` and return the result `U`. Other methods accept `Consumer<T>`, which takes the passed-in value and returns `void`. Yet other methods accept `Runnable`, which does not take any input and returns `void`. Here is one group of these methods:

```
thenRun(Runnable action)
thenApply(Function<T,U> fn)
thenAccept(Consumer<T> action)
```

They all return `CompletableFuture`, which carries the result of the function or void (in the case of `Runnable`and `Consumer`). Each of them has two companion methods that perform the same function asynchronously. For example, let's take the `thenRun(Runnable action)` method. The following are its companions:

- The `thenRunAsync(Runnable action)` method, which runs the action in another thread from the default `ForkJoinPool.commonPool()` pool
- The `thenRun(Runnable action, Executor executor)` method, which runs the action in another thread from the pool passed in as the parameter executor

With that, we have covered nine methods of the `CompletionStage` interface.

Another group of methods consists of the following:

```
thenCompose(Function<T,CompletionStage<U>> fn)
applyToEither(CompletionStage other, Function fn)
acceptEither(CompletionStage other, Consumer action)
runAfterBoth(CompletionStage other, Runnable action)
runAfterEither(CompletionStage other, Runnable action)
thenCombine(CompletionStage<U> other, BiFunction<T,U,V> fn)
thenAcceptBoth(CompletionStage other, BiConsumer<T,U> action)
```

These methods execute the passed in action after one or both the `CompletableFuture` (or `CompletionStage`) objects produce a result that is used as an input to the action. By both, we mean the `CompletableFuture` that provides the method and the one that is passed in as a parameter of the method. From the name of these methods, you can quite reliably guess what their intent is. We will demonstrate some of them in the following examples. Each of these seven methods has two companions for asynchronous processing, too. This means that we have already described 30 (out of 38) methods of the `CompletionStage` interface.

There is a group of two methods that are typically used as terminal operations because they can handle either the result of the previous method (passed in as `T`) or an exception (passed in as `Throwable`):

```
handle(BiFunction<T,Throwable,U> fn)
whenComplete(BiConsumer<T,Throwable> action)
```

We will see an example of the use of these methods later. When an exception is thrown by a method in the chain, all the rest of the chained methods are skipped until the first `handle()` method or `whenComplete()` is encountered. If neither of these two methods are present in the chain, then the exception will bubble up as any other Java exception. These two also have asynchronous companions, which means that we talked about 36 (out of 38) methods of `CompletionStage` interface already.

There is also a method that handles exceptions only (similar to a catch block in the traditional programming):

```
exceptionally(Function<Throwable,T> fn)
```

This method does not have asynchronous companions, just like the last remaining method:

```
toCompletableFuture()
```

It just returns a `CompletableFuture` object with the same properties as this stage. With that, we have described all 38 methods of the `CompletionStage` interface.

There are also some 30 methods in the `CompletableFuture` class that do not belong to any of the implemented interfaces. Some of them return the `CompletableFuture` object after asynchronously executing the provided function:

```
runAsync(Runnable runnable)
runAsync(Runnable runnable, Executor executor)
supplyAsync(Supplier<U> supplier)
supplyAsync(Supplier<U> supplier, Executor executor)
```

Others execute several objects of `CompletableFuture` in parallel:

```
allOf(CompletableFuture<?>... cfs)
anyOf(CompletableFuture<?>... cfs)
```

There is also a group of the methods that generate completed futures, so the `get()` method on the returned `CompletableFuture` object will not block any more:

```
complete(T value)
completedStage(U value)
completedFuture(U value)
failedStage(Throwable ex)
failedFuture(Throwable ex)
completeAsync(Supplier<T> supplier)
completeExceptionally(Throwable ex)
completeAsync(Supplier<T> supplier, Executor executor)
completeOnTimeout(T value, long timeout, TimeUnit unit)
```

The rest of the methods perform various other functions that can be helpful:

```
join()
defaultExecutor()
newIncompleteFuture()
getNow(T valueIfAbsent)
getNumberOfDependents()
minimalCompletionStage()
```

```
isCompletedExceptionally()
obtrudeValue(T value)
obtrudeException(Throwable ex)
orTimeout(long timeout, TimeUnit unit)
delayedExecutor(long delay, TimeUnit unit)
```

Refer to the official Oracle documentation, which describes these and other methods of the `CompletableFuture` API at `http://download.java.net/java/jdk9/docs/api/index.html?java/util/concurrent/CompletableFuture.html`.

The CompletableFuture API Enhancements in Java 9

Java 9 introduces several enhancements to `CompletableFuture`:

- The `CompletionStage<U> failedStage(Throwable ex)` factory method returns the `CompletionStage` object completed with the given exception
- The `CompletableFuture<U> failedFuture(Throwable ex)` factory method returns the `CompletableFuture`object completed with the given exception
- The new `CompletionStage<U> completedStage(U value)` factory method returns the `CompletionStage` object completed with the given `U` value
- `CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)` completes `CompletableFuture` task with the given `T` value if not otherwise completed before the given timeout
- `CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)` completes `CompletableFuture` with `java.util.concurrent.TimeoutException` if not completed before the given timeout
- It is possible now to override the `defaultExecutor()` method to support another default executor
- A new method, `newIncompleteFuture()`, makes it easier to subclass the `CompletableFuture` class

The Problem and the Solution using Future

To demonstrate and appreciate the power of `CompletableFuture`, let's start with a problem implemented using just `Future` and then see how much more effectively it can be solved with `CompletableFuture`. Let's imagine that we are tasked with modeling a building that consists of four stages:

- Collecting materials for the foundation, walls, and roof
- Installing the foundation

- Raising up the walls
- Constructing and finishing the roof

In the traditional sequential programming for the single thread, the model would look like this:

```
StopWatch stopWatch = new StopWatch();
Stage failedStage;
String SUCCESS = "Success";

stopWatch.start();
String result11 = doStage(Stage.FoundationMaterials);
String result12 = doStage(Stage.Foundation, result11);
String result21 = doStage(Stage.WallsMaterials);
String result22 = doStage(Stage.Walls,
                       getResult(result21, result12));
String result31 = doStage(Stage.RoofMaterials);
String result32 = doStage(Stage.Roof,
                       getResult(result31, result22));
System.out.println("House was" +
       (isSuccess(result32)?"":" not") + " built in "
                + stopWatch.getTime()/1000. + " sec");
```

Here, `Stage` is an enumeration:

```
enum Stage {
    FoundationMaterials,
    WallsMaterials,
    RoofMaterials,
    Foundation,
    Walls,
    Roof
}
```

The `doStage()` method has two overloaded versions. Here is the first one:

```
String doStage(Stage stage) {
    String result = SUCCESS;
    boolean failed = stage.equals(failedStage);
    if (failed) {
        sleepSec(2);
        result = stage + " were not collected";
        System.out.println(result);
    } else {
        sleepSec(1);
```

```
            System.out.println(stage + " are ready");
        }
        return result;
    }
```

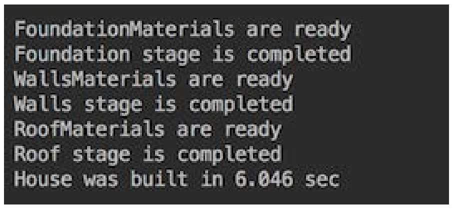
The second version is as follows:

```
    String doStage(Stage stage, String previousStageResult) {
      String result = SUCCESS;
      boolean failed = stage.equals(failedStage);
      if (isSuccess(previousStageResult)) {
        if (failed) {
          sleepSec(2);
          result = stage + " stage was not completed";
          System.out.println(result);
        } else {
          sleepSec(1);
          System.out.println(stage + " stage is completed");
        }
      } else {
          result = stage + " stage was not started because: "
                                        + previousStageResult;
          System.out.println(result);
      }
      return result;
    }
```

The `sleepSec()`, `isSuccess()`, and `getResult()` methods look like this:

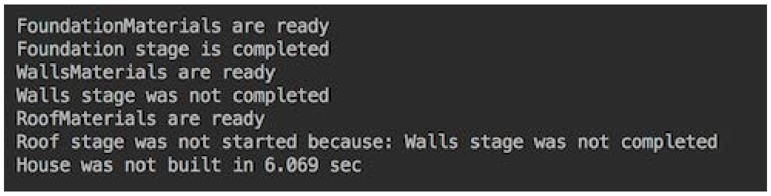
```
    private static void sleepSec(int sec) {
        try {
            TimeUnit.SECONDS.sleep(sec);
        } catch (InterruptedException e) {
        }
    }
    boolean isSuccess(String result) {
        return SUCCESS.equals(result);
    }
    String getResult(String result1, String result2) {
        if (isSuccess(result1)) {
            if (isSuccess(result2)) {
                return SUCCESS;
            } else {
                return result2;
            }
        } else {
            return result1;
        }
    }
```

The successful house construction (if we run the previous code without assigning any value to the `failedStage` variable) looks like this:



```
FoundationMaterials are ready
Foundation stage is completed
WallsMaterials are ready
Walls stage is completed
RoofMaterials are ready
Roof stage is completed
House was built in 6.046 sec
```

If we set `failedStage=Stage.Walls`, the result will be as follows:



```
FoundationMaterials are ready
Foundation stage is completed
WallsMaterials are ready
Walls stage was not completed
RoofMaterials are ready
Roof stage was not started because: Walls stage was not completed
House was not built in 6.069 sec
```

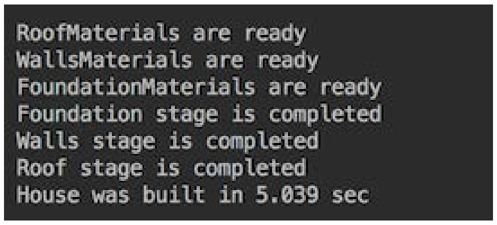
Using `Future`, we can shorten the time it takes to build the house:

```
ExecutorService execService = Executors.newCachedThreadPool();
Callable<String> t11 =
                     () -> doStage(Stage.FoundationMaterials);
Future<String> f11 = execService.submit(t11);
List<Future<String>> futures = new ArrayList<>();
futures.add(f11);

Callable<String> t21 = () -> doStage(Stage.WallsMaterials);
Future<String> f21 = execService.submit(t21);
futures.add(f21);

Callable<String> t31 = () -> doStage(Stage.RoofMaterials);
Future<String> f31 = execService.submit(t31);
futures.add(f31);

String result1 = getSuccessOrFirstFailure(futures);
String result2 = doStage(Stage.Foundation, result1);
String result3 =
       doStage(Stage.Walls, getResult(result1, result2));
String result4 =
        doStage(Stage.Roof, getResult(result1, result3));
```

Here, the `getSuccessOrFirstFailure()` method looks like this:

```
String getSuccessOrFirstFailure(
                      List<Future<String>> futures) {
    String result = "";
    int count = 0;
    try {
        while (count < futures.size()) {
            for (Future<String> future : futures) {
                if (future.isDone()) {
                    result = getResult(future);
                    if (!isSuccess(result)) {
                        break;
                    }
                    count++;
                } else {
                    sleepSec(1);
                }
            }
            if (!isSuccess(result)) {
                break;
            }
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return result;
}
```

The successful building of the house now is faster because material collection happens in parallel:

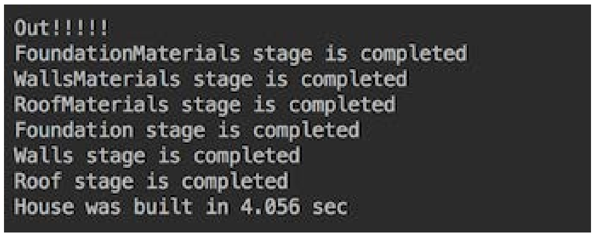


```
RoofMaterials are ready
WallsMaterials are ready
FoundationMaterials are ready
Foundation stage is completed
Walls stage is completed
Roof stage is completed
House was built in 5.039 sec
```

By taking advantage of Java functional programming, we can change the second half of our implementation to the following:

```
Supplier<String> supplier1 =
                 () -> doStage(Stage.Foundation, result1);
Supplier<String> supplier2 =
                () -> getResult(result1, supplier1.get());
Supplier<String> supplier3 =
              () -> doStage(Stage.Walls, supplier2.get());
Supplier<String> supplier4 =
                () -> getResult(result1, supplier3.get());
Supplier<String> supplier5 =
               () -> doStage(Stage.Roof, supplier4.get());
System.out.println("House was" +
              (isSuccess(supplier5.get()) ? "" : " not") +
      " built in " + stopWatch.getTime() / 1000. + " sec");
```

The chain of the previous nested functions is triggered by `supplier5.get()` in the last line. It blocks until all the functions are completed sequentially, so there is no performance improvement:



```
Out!!!!!
FoundationMaterials stage is completed
WallsMaterials stage is completed
RoofMaterials stage is completed
Foundation stage is completed
Walls stage is completed
Roof stage is completed
House was built in 4.056 sec
```

And that is as far as we can go with `Future`. Now let's see if we can improve the previous code using `CompletableFuture`.

The Solution with CompletableFuture

Here's how we can chain the same operations using the `CompletableFuture` API:

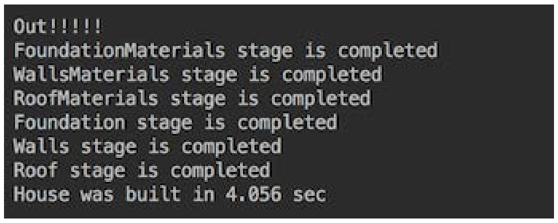
```
stopWatch.start();
ExecutorService pool = Executors.newCachedThreadPool();
CompletableFuture<String> cf1 =
   CompletableFuture.supplyAsync(() ->
           doStageEx(Stage.FoundationMaterials), pool);
CompletableFuture<String> cf2 =
   CompletableFuture.supplyAsync(() ->
                doStageEx(Stage.WallsMaterials), pool);
```

```
CompletableFuture<String> cf3 =
   CompletableFuture.supplyAsync(() ->
                 doStageEx(Stage.RoofMaterials), pool);
CompletableFuture.allOf(cf1, cf2, cf3)
  .thenComposeAsync(result ->
      CompletableFuture.supplyAsync(() -> SUCCESS), pool)
  .thenApplyAsync(result ->
                 doStage(Stage.Foundation, result), pool)
  .thenApplyAsync(result ->
                      doStage(Stage.Walls, result), pool)
  .thenApplyAsync(result ->
                       doStage(Stage.Roof, result), pool)
  .handleAsync((result, ex) -> {
       System.out.println("House was" +
         (isSuccess(result) ? "" : " not") + " built in "
                 + stopWatch.getTime() / 1000. + " sec");
       if (result == null) {
         System.out.println("Because: " + ex.getMessage());
         return ex.getMessage();
       } else {
         return result;
       }
  }, pool);
System.out.println("Out!!!!!");
```

To make it work, we had to change the implementation of one of the `doStage()` to `doStageEx()` methods:

```
String doStageEx(Stage stage) {
  boolean failed = stage.equals(failedStage);
  if (failed) {
    sleepSec(2);
    throw new RuntimeException(stage +
                          " stage was not completed");
  } else {
    sleepSec(1);
    System.out.println(stage + " stage is completed");
  }
  return SUCCESS;
}
```

The reason we do this is because the `CompletableFuture.allOf()` method returns `CompletableFuture<Void>`, while we need to communicate to the further stages the result of the first three stages of collecting materials. The result looks now as follows:

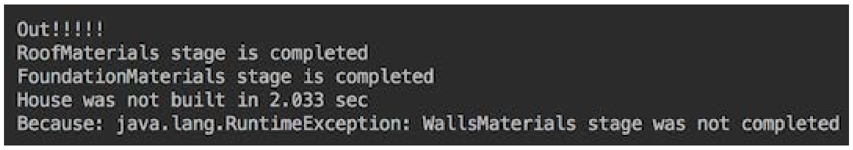


```
Out!!!!!
FoundationMaterials stage is completed
WallsMaterials stage is completed
RoofMaterials stage is completed
Foundation stage is completed
Walls stage is completed
Roof stage is completed
House was built in 4.056 sec
```

There are two points to note:

- We used a dedicated pool of threads to run all the operations asynchronously; if there were several CPUs or some operations use IO while others do not, the result could be even better
- The last line of the code snippet (`Out!!!!!`) came out first, which means that all the chains of the operations related to building the house were executed asynchronously

Now, let's see how the system behaves if one of the first stages of collecting materials fails (`failedStage = Stage.WallsMaterials`):



```
Out!!!!!
RoofMaterials stage is completed
FoundationMaterials stage is completed
House was not built in 2.033 sec
Because: java.lang.RuntimeException: WallsMaterials stage was not completed
```

The exception was thrown by the `WallsMaterials` stage and caught by the `handleAsync()` method, as expected. And, again, the processing was done asynchronously after the `Out!!!!!` message was printed.

Other Useful Features of CompletableFuture

One of the great advantages of `CompletableFuture` is that it can be passed around as an object and used several times to start different chains of operations. To demonstrate this capability, let's create several new operations:

```
String getData() {
  System.out.println("Getting data from some source...");
  sleepSec(1);
  return "Some input";
}
SomeClass doSomething(String input) {
  System.out.println(
    "Doing something and returning SomeClass object...");
  sleepSec(1);
  return new SomeClass();
}
AnotherClass doMore(SomeClass input) {
  System.out.println("Doing more of something and " +
                   "returning AnotherClass object...");
  sleepSec(1);
  return new AnotherClass();
}
YetAnotherClass doSomethingElse(AnotherClass input) {
  System.out.println("Doing something else and " +
               "returning YetAnotherClass object...");
  sleepSec(1);
  return new YetAnotherClass();
}
int doFinalProcessing(YetAnotherClass input) {
  System.out.println("Processing and finally " +
                               "returning result...");
  sleepSec(1);
  return 42;
}
AnotherType doSomethingAlternative(SomeClass input) {
  System.out.println("Doing something alternative " +
              "and returning AnotherType object...");
  sleepSec(1);
  return new AnotherType();
}
YetAnotherType doMoreAltProcessing(AnotherType input) {
  System.out.println("Doing more alternative and " +
                  "returning YetAnotherType object...");
  sleepSec(1);
  return new YetAnotherType();
}
int doFinalAltProcessing(YetAnotherType input) {
  System.out.println("Alternative processing and " +
                       "finally returning result...");
  sleepSec(1);
  return 43;
}
```

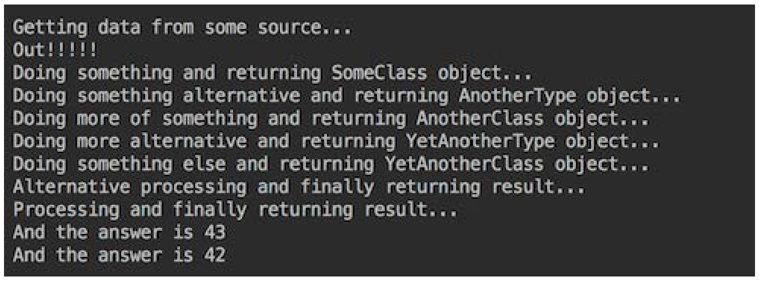
The results of these operations are going to be handled by the `myHandler()` method:

```
int myHandler(Integer result, Throwable ex) {
    System.out.println("And the answer is " + result);
    if (result == null) {
        System.out.println("Because: " + ex.getMessage());
        return -1;
    } else {
        return result;
    }
}
```

Note all the different types returned by the operations. Now we can build a chain that forks in two at some point:

```
ExecutorService pool = Executors.newCachedThreadPool();
CompletableFuture<SomeClass> completableFuture =
   CompletableFuture.supplyAsync(() -> getData(), pool)
     .thenApplyAsync(result -> doSomething(result), pool);

completableFuture
   .thenApplyAsync(result -> doMore(result), pool)
   .thenApplyAsync(result -> doSomethingElse(result), pool)
   .thenApplyAsync(result -> doFinalProcessing(result), pool)
   .handleAsync((result, ex) -> myHandler(result, ex), pool);

completableFuture
   .thenApplyAsync(result -> doSomethingAlternative(result), pool)
   .thenApplyAsync(result -> doMoreAltProcessing(result), pool)
   .thenApplyAsync(result -> doFinalAltProcessing(result), pool)
   .handleAsync((result, ex) -> myHandler(result, ex), pool);

System.out.println("Out!!!!!");
```

The result of this example is as follows:



```
Getting data from some source...
Out!!!!!
Doing something and returning SomeClass object...
Doing something alternative and returning AnotherType object...
Doing more of something and returning AnotherClass object...
Doing more alternative and returning YetAnotherType object...
Doing something else and returning YetAnotherClass object...
Alternative processing and finally returning result...
Processing and finally returning result...
And the answer is 43
And the answer is 42
```

The `CompletableFuture` API provides a very rich and well-thought-through API that supports, among other things, the latest trends in reactive microservices because it allows processing data fully asynchronously as it comes in, splitting the flow if needed, and scaling to accommodate the increase of the input. We encourage you to study the examples (many more are provided in the code that accompanies this book) and look at the API at `http://download.java.net/java/jdk9/docs/api/index.html?java/util/concurrent/CompletableFuture.html`.

Stream API Improvements

Most of the new `Stream` API features in Java 9 have already been demonstrated in the section that describes `Stream` filtering. To remind you, here are the examples we have demonstrated based on the `Stream` API improvements in JDK 9:

```
long c1 = senators.stream()
        .flatMap(s -> Stream.ofNullable(s.getParty()
                              == "Party1" ? s : null))
        .count();
System.out.println("OfNullable: Members of Party1: " + c1);

long c2 = senators.stream()
        .map(s -> s.getParty() == "Party2" ? Optional.of(s)
                                        : Optional.empty())
        .flatMap(Optional::stream)
        .count();
System.out.println("Optional.stream(): Members of Party2: "
                                                     + c2);

senators.stream().limit(5)
        .takeWhile(s -> Senate.timesVotedYes(s) < 5)
        .forEach(s -> System.out.println("takeWhile(<5): "
                     + s + ": " + Senate.timesVotedYes(s)));

senators.stream().limit(5)
         .dropWhile(s -> Senate.timesVotedYes(s) < 5)
        .forEach(s -> System.out.println("dropWhile(<5): "
                     + s + ": " + Senate.timesVotedYes(s)));
```

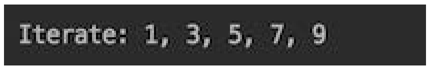
The only one we have not mentioned yet is the new overloaded `iterate()` method:

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

An example of its usage is as follows:

```
String result =

    IntStream.iterate(1, i -> i + 2)

             .limit(5)

             .mapToObj(i -> String.valueOf(i))

             .collect(Collectors.joining(", "));

System.out.println("Iterate: " + result);
```

We had to add `limit(5)` because this version of the `iterate()` method creates an unlimited stream of integer numbers. The result of the previous code is as follows:



```
Iterate: 1, 3, 5, 7, 9
```

In Java 9, an overloaded `iterate()` method was added:

```
static <T> Stream<T> iterate(T seed,
     Predicate<? super T> hasNext, UnaryOperator<T> next)
```

As you see, it has now a `Predicate` functional interface as a parameter that allows limiting the stream as needed. For example, the following code produces exactly the same result as the previous example with `limit(5)`:

```
String result =
   IntStream.iterate(1, i -> i < 11, i -> i + 2)
            .mapToObj(i -> String.valueOf(i))
            .collect(Collectors.joining(", "));
System.out.println("Iterate: " + result);
```

Note that the type of the stream element does not need to be an integer. It can be any type produced by the source. So, the new `iterate()` method can be used to provide criteria for the termination of the stream of any type of data.

Summary

In this lesson, we covered a lot of ground in the area of the new features introduced with Java 9. First, we looked at many ways to stream filtering, starting with the basic `filter()` method and ending up using the `Stream` API additions of JDK 9. Then, you learned a better way to analyze the stack trace using the new `StackWalker` class. The discussion was illustrated by specific examples that help you to see the real working code.

We used the same approach while presenting new convenient factory methods for creating immutable collections and new capabilities for asynchronous processing that came with the `CompletableFuture` class and its enhancements in JDK 9.

We ended this lesson by enumerating the improvements to the `Stream` API--those we have demonstrated in the filtering code examples and the new `iterate()` method.

With this, we come to the end of this book. You can now try and apply the tips and techniques you have learned to your project or, if it is not suitable for that, to build your own Java project for high performance. While doing that, try to solve real problems. That will force you to learn new skills and frameworks instead of just applying the knowledge you have already, although the latter is helpful too--it keeps your knowledge fresh and practical.

The best way to learn is to do it yourself. As Java continues to improve and expand, watch out for new editions of this and similar books by Packt.

Assessments

1. The _______ interface was introduced in Java 8 to emit elements and supports a variety of operations that perform computations based on stream elements.
2. Which of the following factory methods of the `StackWalker` class creates a `StackWalker` class instance with the given option of specifying the stack frame information that it can access?
    1. `getInstance()`
    2. `getInstance(StackWalker.Option option)`
    3. `getInstance(Set<StackWalker.Option> options)`
    4. `getInstance(Set<StackWalker.Option> options, int estimatedDepth)`

3. State whether True or False: The `CompletableFuture` API consists of many methods which are implementations of the `CompletionStage` interface, and are the implementations of `Future`.
4. Which among the following methods is used when a type of filtering is required to skip all the duplicate elements in a stream and select only unique element.
    1. `distinct()`
    2. `unique()`
    3. `selectall()`
    4. `filtertype()`
5. State whether True or False: One of the great advantages of `CompletableFuture` is that it can be passed around as an object and used several times to start different chains of operations.

Assessment Answers

Lesson 1: Learning Java 9 Underlying Performance Improvements

Question Number	Answer
1	tool
2	1
3	True
4	3
5	3

Lesson 2: Tools for Higher Productivity and Faster Application

Question Number	Answer
1	Ahead-of-Time
2	1
3	False
4	1
5	3

Lesson 3: Multithreading and Reactive Programming

Question Number	Answer
1	`calculateAverageSqrt()`
2	3
3	False
4	2
5	RxJava

Lesson 4: Microservices

Question Number	Answer
1	Vertx
2	4
3	False
4	1,2
5	True

Lesson 5: Making Use of New APIs to Improve Your Code

Question Number	Answer
1	`java.util.streams.Stream`
2	2
3	True
4	1
5	True