Mohamed Sanaulla, Nick Samoylov

Java 9 Cookbook

Over 100 practical recipes to develop modern applications in Java



Packt>

Mohamed Sanaulla, Nick Samoylov

Java 9 Cookbook

Over 100 practical recipes to develop modern applications in Java



Packt>

Java 9 Cookbook

Over 100 practical recipes to develop modern applications in Java

Mohamed Sanaulla
Nick Samoylov



BIRMINGHAM - MUMBAI

Java 9 Cookbook

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2017

Production reference: 1180817

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street

Birmingham
B3 2PB, UK.

ISBN 978-1-78646-140-7

www.packtpub.com

Credits

Author Mohamed Sanaulla Nick Samoylov	**Copy Editor** Muktikant Garimella
Reviewer Aristides Villarreal Bravo	**Project Coordinator** Ulhas Kambali
Commissioning Editor Kunal Parikh	**Proofreader** Safis Editing

Acquisition Editor Denim Pinto	**Indexer** Rekha Nair
Content Development Editor Nikhil Borkar	**Graphics** Abhinash Sahu
Technical Editor Subhalaxmi Nadar	**Production Coordinator** Melwyn Dsa

About the Authors

Mohamed Sanaulla is a software developer with over 7 years, experience in backend and full stack development. He is also one of the moderators on Code Ranch (formerly known as Java Ranch).

I would like to thank everyone who has helped me in the process of writing this book.

Nick Samoylov was born in Moscow, raised in Ukraine, and lived in the Crimea. He graduated as an engineer-physicist from Moscow Institute of Physics and Technologies, has worked as a theoretical physicist, and 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. In 1999, with his wife Luda and two daughters, he emigrated to the USA and has been living in Colorado since then.

Nick adopted Java in 1997 and used it for working as a software developer-contractor for a variety of companies, including BEA Systems, Warner Telecom, and Boeing. For Boeing, he and his wife, also a Java programmer,

developed a system of loading application data to the airplane via the internet.

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 webservices on Linux deployed in a cloud.

Nick and Luda have two daughters who graduated from Harvard and Tufts universities, respectively. One has also received a doctoral degree from Brown University and now works as a professor in the University of California in Chico. The other daughter is an executive director of the investment bank, JPMorgan, in Madrid, Spain.

In his free time, Nick likes to read (mostly non-fiction), write (fiction novels and blogs), and hike the Rocky Mountains.

About the Reviewer

Aristides Villarreal Bravo is a Java developer, a member of the NetBeans Dream Team, and a Java User Groups leader. He lives in Panama. He has organized and participated in various conferences and seminars related to Java, JavaEE, NetBeans, the NetBeans platform, free software, and mobile devices. He is the author of jmoordb and tutorials, and he blogs about Java, NetBeans, and web development.

Aristides has participated in several interviews on sites about topics such as NetBeans, NetBeans DZone, and JavaHispano. He is a developer of plugins for
NetBeans. He is the CEO of Javscaz Software Developers. He has also worked on *Developers of jmoordb*.

I would like to thank my mother, father, and all family and friends.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1786461404.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

- Preface
  - What this book covers
  - What you need for this book
  - Who this book is for
  - Sections
    - Getting ready
    - How to do it…
    - How it works…
    - There's more…
    - See also
  - Conventions
  - Reader feedback
  - Customer support
    - Downloading the example code
    - Errata
    - Piracy
    - Questions

1. Installation and a Sneak Peek into Java 9
   - Introduction
   - Installing JDK 9 on Windows and setting up the PATH variable
     - How to do it...
   - Installing JDK 9 on Linux (Ubuntu, x64) and configuring the PATH variable
     - How to do it...
   - Compiling and running a Java application
     - Getting ready
     - How to do it...
   - New features in Java 9
     - How to do it...
       - JEP 102 -- Process API updates
       - JEP 110 -- HTTP/2 client
       - JEP 213 -- milling project coin
       - JEP 222: jshell -- the Java shell (Read-Eval-Print Loop)

JEP 238 -- multi-release JAR files
JEP 266 -- more concurrency updates
Project Jigsaw
There's more...
Using new tools in JDK 9
Getting ready
How to do it...
jdeprscan
jdeps
jlink
jmod
JShell
Comparing JDK 8 and JDK 9
Getting ready
How to do it...
See also
2. Fast Track to OOP - Classes and Interfaces
Introduction
Implementing object-oriented design using classes
Getting ready
How to do it...
How it works...
There's more...
See also
Using inner classes
Getting ready
How to do it...
How it works...
There's more...
See also
Using inheritance and composition to make the design extensible
Getting ready
How to do it...
How it works...
Composition makes the design more extensible
See also

Coding to an interface
Getting ready
How to do it...
How it works...
There's more...
See also
Creating interfaces with default and static methods
Getting ready
How to do it...
How it works...
See also
Creating interfaces with private methods
Getting ready
How to do it...
How it works...
There's more...
See also
Using enums to represent constant entities
Getting ready
How to do it...
How it works...
There's more...
Using the @Deprecated annotation to deprecate APIs
Getting ready
How to do it...
How it works...
There's more...
Using HTML5 in Javadocs
Getting ready
How to do it...
3. Modular Programming
Introduction
Using jdeps to find dependencies in a Java application
Getting ready
How to do it...
How it works...

There's more...
Creating a simple modular application
Getting ready
How to do it...
How it works...
See also
Creating a modular JAR
Getting ready
How to do it...
Using a module JAR with pre-JDK 9 applications
Getting ready
How to do it...
See also
Bottom-up migration
Getting ready
How to do it...
Modularizing banking.util.jar
Modularizing math.util.jar
Modularizing calculator.jar
How it works...
Top-down migration
Getting ready
How to do it...
Modularizing the calculator
Modularizing banking.util
Modularizing math.util
Using services to create loose coupling between consumer and provider mod
ules
Getting ready
How to do it...
Creating a custom modular runtime image using jlink
Getting ready
How to do it...
Compiling for older platform versions
Getting ready
How to do it...

How it works...
Creating multirelease JARs
How to do it...
How it works...
Using Maven to develop a modular application
Getting ready
How to do it...
4. Going Functional
Introduction
Understanding and creating a functional interface
Getting ready
How to do it...
How it works...
There's more...
See also
Understanding lambda expressions
Getting ready
How to do it...
How it works...
There's more...
See also
Using method references
Getting ready
How to do it...
There's more...
See also
Creating and invoking lambda-friendly APIs
Getting ready
How it works...
There's more...
See also
Leveraging lambda expressions in your programs
Getting ready
How to do it...
There's more...
See also

5. Stream Operations and Pipelines
- Introduction
- Using the new factory methods to create collection objects
  - Getting ready
  - How to do it...
  - There's more...
  - See also
- Creating and operating on streams
  - Getting ready
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Creating an operation pipeline on streams
  - Getting ready
  - How to do it...
  - There's more...
  - See also
- Parallel computations on streams
  - Getting ready
  - How to do it...
  - See also

6. Database Programming
- Introduction
- Connecting to a database using JDBC
  - How to do it...
  - How it works...
  - There's more...
  - See also
- Setting up the tables required for DB interactions
  - Getting ready
  - How it works...
  - There's more...
  - See also
- Performing CRUD operations
  - Getting ready

How to do it...
There's more...
See also
Using prepared statements
Getting ready
How to do it...
There's more...
See also
Using transactions
Getting ready
How to do it...
There's more...
Working with large objects
Getting ready
How to do it...
There's more...
Executing stored procedures
Getting ready
How to do it...
There's more...
7. Concurrent and Multithreaded Programming
Introduction
Using the basic element of concurrency - thread
Getting ready
How to do it...
There's more...
See also
Different synchronization approaches
Getting ready
How to do it...
There's more...
See also
Immutability as a means to achieve concurrency
Getting ready
How to do it...
There's more...

See also
Using concurrent collections
Getting ready
How to do it...
How it works...
See also
Using the executor service to execute async tasks
Getting ready
How to do it...
How it works...
There's more...
See also
Using fork/join to implement divide-and-conquer
Getting ready
How to do it...
Using flow to implement the publish-subscribe pattern
Getting ready
How to do it...
8. Better Management of the OS Process
Introduction
Spawning a new process
Getting ready
How to do it...
How it works...
Redirecting the process output and error streams to file
Getting ready
How to do it...
There is more...
Changing the working directory of a subprocess
Getting ready
How to do it...
How it works...
Setting the environment variable for a subprocess
How to do it...
How it works...
Running shell scripts

Getting ready
How to do it...
How it works...
Obtaining the process information of the current JVM
How to do it...
How it works...
Obtaining the process information of the spawned process
Getting ready
How to do it...
Managing the spawned process
How to do it...
Enumerating live processes in the system
How to do it...
Connecting multiple processes using pipe
Getting ready
How to do it...
How it works...
Managing subprocesses
Getting ready
How to do it...
How it works...
9. GUI Programming Using JavaFX
Introduction
Creating a GUI using JavaFX controls
Getting ready
How to do it...
How it works...
Using the FXML markup to create a GUI
Getting ready
How to do it...
How it works...
See also
Using CSS to style elements in JavaFX
Getting ready
How to do it...
How it works...

Creating a bar chart
Getting ready
How to do it...
How it works...
See also
Creating a pie chart
Getting ready
How to do it...
How it works...
Creating a line chart
Getting ready
How to do it...
How it works...
See also
Creating an area chart
Getting ready
How to do it...
How it works...
See also
Creating a bubble chart
Getting ready
How to do it...
How it works...
See also
Creating a scatter chart
Getting ready
How to do it...
How it works...
See also
Embedding HTML in an application
Getting ready
How to do it...
How it works...
There's more...
Embedding media in an application
Getting ready

How to do it...
How it works...
There's more...
Adding effects to controls
How to do it...
How it works...
There's more...
Using the new TIFF I/O API to read TIFF images
Getting ready
How to do it...
10. RESTful Web Services Using Spring Boot
Introduction
Creating a simple Spring Boot application
Getting ready
How to do it...
How it works...
Interacting with the database
Getting ready
Installing MySQL tools
Creating a sample database
Creating a person table
Populating sample data
How to do it...
How it works...
Creating a RESTful web service
Getting ready
How to do it...
How it works...
Creating multiple profiles for Spring Boot
Getting ready
How to do it...
How it works...
There's more...
Deploying RESTful web services to Heroku
Getting ready
Setting up a Heroku account

Creating a new app from the UI

Creating a new app from the CLI

How to do it...

There's more...

Containerizing the RESTful web service using Docker

Getting ready

How to do it...

How it works...

11. Networking

Introduction

Making an HTTP GET request

How to do it...

How it works...

Making an HTTP POST request

How to do it...

Making an HTTP request for a protected resource

How to do it...

How it works...

Making an asynchronous HTTP request

How to do it...

Making an HTTP request using Apache HttpClient

Getting ready

How to do it...

There is more...

Making an HTTP request using the Unirest HTTP client library

Getting ready

How to do it...

There's more...

12. Memory Management and Debugging

Introduction

Understanding the G1 garbage collector

Getting ready

How to do it...

How it works...

See also

Unified logging for JVM

Getting ready
How to do it...
See also
Using the new diagnostic commands for the JVM
How to do it...
How it works...
See also
Try with resources for better resource handling
How to do it...
How it works...
See also
Stack walking for improved debugging
Getting ready
How to do it...
How it works...
See also
Some best practices for better memory usage
How it works...
See also
13. The Read-Evaluate-Print Loop (REPL) Using JShell
Introduction
Getting familiar with REPL
Getting ready
How to do it...
How it works...
Navigating JShell and its commands
How to do it...
Evaluating code snippets
How to do it...
There's more...
Object-oriented programming in JShell
How to do it...
Saving and restoring the JShell command history
How to do it...
Using the JShell Java API
How to do it...

How it works...
14. Scripting Using Oracle Nashorn
Introduction
Using the jjs command-line tool
Getting ready
How to do it...
There's more...
Embedding the Oracle Nashorn engine
Getting ready
How to do it...
Invoking Java from Oracle Nashorn
How to do it...
How it works...
There's more...
Using the ES6 features implemented in Oracle Nashorn
How to do it...
15. Testing
Introduction
Unit testing of an API using JUnit
Getting ready
How to do it...
How it works...
See also
Unit testing by mocking dependencies
Getting ready
How to do it...
How it works...
There's more...
See also
Using fixtures to populate data for testing
How to do it...
How it works...
See also
Behavioral testing
Getting ready
How to do it...

How it works...

Preface

This cookbook offers a range of software development examples in simple and straightforward Java 9 code, providing step-by-step resources and time-saving methods to help you solve data problems efficiently. Starting with the installation of Java, each recipe addresses a specific problem, with a discussion that explains the solution, and offers insight into how it works. We cover major concepts about the core programming language as well as common tasks to build a wide variety of software. You will learn new features in the form of recipes, to make your application modular, secure, and fast.

What this book covers

Chapter 1, *Installation and Sneak Peek into Java 9*, helps you to set up the development environment for running your Java programs and gives a brief overview of the new features and tools in Java 9

Chapter 2, *Fast Track to OOP - Classes and Interfaces*, covers object-oriented programming principles and design solutions, including inner classes, inheritance, composition, interfaces, enumerations, and the Java 9 changes to Javadocs.

Chapter 3, *Modular Programming,* introduces jigsaw as a major feature and a huge leap for the Java ecosystem. This chapter demonstrates how to use tools, such as jdeps and jlink, to create simple modular applications, and related artifacts such as modular JARs, and finally how to modularize your pre-Java 9 applications.

Chapter 4, *Going Functional*, introduces a programming paradigm called functional programming and its applicability in Java 9. Topics covered include functional interfaces, lambda expressions, and lambda-friendly APIs.

Chapter 5, *Stream Operations and Pipelines*, shows how to leverage streams and chain multiple operations on a collection to create a pipeline, use factory methods to create collection objects, create and operate on streams, and create an operation pipeline on streams, including parallel computations.

Chapter 6, *Database Programming*, covers both basic and commonly used interactions between a Java application and a database, right from connecting to the database and performing CRUD operations to creating transactions, storing procedures, and working with large objects.

Chapter 7, *Concurrent and Multithreaded Programming*, presents different ways of incorporating concurrency and some best practices, such as synchronization and immutability. We will also discuss the implementation of some commonly used patterns, such as divide-conquer and publish-subscribe, using the constructs provided by Java.

Chapter 8, *Better Management of the OS Process*, elaborates on the new API enhancements around the Process API.

Chapter 9, *GUI Programming Using JavaFX*, shows how to get started with creating JavaFX applications, leverage CSS styling into your applications, build a GUI in a declarative way using FXML, and use the graph, media, and browser components of JavaFX.

Chapter 10, *RESTful Webservices Using Spring Boot*, deals with creating simple RESTful webservices using Spring boot, deploying them to Heroku, and finally dockerizing Spring boot-based RESTful webservice applications.

Chapter 11, *Networking*, shows you how to use different HTTP client API libraries, namely the API provided in Java 9 as an incubator module, the Apache HTTP client, and the Unirest HTTP client API.

Chapter 12, *Memory Management and Debugging*, explores managing the memory of a Java application, including an introduction to the garbage collection algorithm used in Java 9, and some new features, which help in advanced application diagnostics. We'll also show how to manage resources by using the new try-with-resources construct and the new stack walking API.

Chapter 13, *The Read-Evaluate-Print Loop (REPL) Using JShell*, shows you how to work with the new REPL tool and JShell, provided as part of the JDK.

Chapter 14, *Scripting Using Oracle Nashorn*, shows how to interoperate between JavaScript and Java using the Oracle Nashorn JavaScript engine

and how to use the jjs command-line tool to run JavaScript. It also explores Oracle Nashorn's support for the new ECMAScript 6.

Chapter 15, *Testing*, explains how to unit-test your APIs before they are integrated with other components, including stubbing dependencies with some dummy data and mocking dependencies. We will also show you how to write fixtures to populate the test data and then how to test your application behavior by integrating different APIs and testing them.

What you need for this book

To run the code examples, you will need a computer with 2GB RAM at least, 10GB free disk space, and Windows or Linux OS. The following software/libraries are required:

- JDK 9 (for all chapters)
- PostgreSQL 9.4 DB (for `Chapter 6`, *Database Programming*)
- Junit 4.12 (for `Chapter 15`, *Testing*)
- Mockito 2.7.13 (for `Chapter 15`, *Testing*)
- Maven 3.5.0 (for `Chapter 3`, *Modular Programming*)
- MySQL 5.7.19 DB (for `Chapter 10`, *RESTful Webservices Using Spring Boot*)
- Heroku CLI (for `Chapter 10`, *RESTful Webservices Using Spring Boot*)
- Docker (for `Chapter 10`, *RESTful Webservices Using Spring Boot*)

Who this book is for

The book is for intermediate to advanced Java programmers who want to make their applications fast, secure, and scalable.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it…, How it works…, There's more…, and See also). To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the `reader` more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "When run, it produces the same value as with an object of the `Vehicle` class." A block of code is set as follows:

```
module newfeatures{
  requires jdk.incubator.httpclient;
}
```

Any command-line input or output is written as follows:

```
$> vim ~/.bash_aliases
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Right-click on My Computer and then click on Properties."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors` .

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account. Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Java-9-Cookbook. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Installation and a Sneak Peek into Java 9

In this chapter, we will cover the following recipes:

- Installing JDK 9 on Windows and setting up the PATH variable
- Installing JDK 9 on Linux (Ubuntu, x64) and configuring the PATH variable
- Compiling and running a Java application
- New features in Java 9
- Using new tools in JDK 9
- Comparing JDK 8 with JDK 9

Introduction

Every quest for learning a programming language begins with setting up the environment to experiment our learning. Keeping in sync with this philosophy, in this chapter, we will show you how to set up your development environment and then run a simple modular application to test our installation. After that, we'll give you an introduction to the new features and tools in JDK 9. Then, we'll end the chapter with a comparison between the JDK 8 and JDK 9 installations.

Installing JDK 9 on Windows and setting up the PATH variable

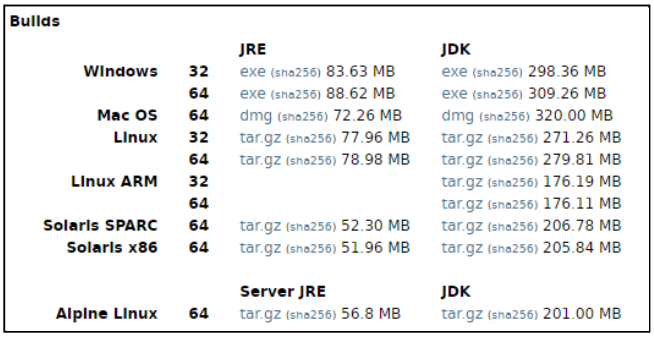
In this recipe, we will look at installing JDK on Windows and how to set up the `PATH` variable to be able to access the Java executables (such as `javac`, `java`, and `jar`, among others) from anywhere within the command shell.

How to do it...

1. Visit `https://jdk9.java.net/download/` and accept the early adopter license agreement, which looks like this:



2. After accepting the license, you will get a grid of the available JDK bundles based on the OS and architecture (32/64 bit), as shown here:



3. Click to download the relevant JDK executable (`.exe`) for your Windows platform.
4. Run the JDK executable (`.exe`) and follow the onscreen instructions to install JDK on your system.
5. If you have chosen all the defaults during the installation, you will find JDK installed in `C:/Program Files/Java` for 64 bit and `C:/Program Files (x86)/Java` for 32 bit.

Now that we have finished installing JDK, let's see how we can set the `PATH` variable.

The tools provided with JDK, namely `javac`, `java`, `jconsole`, and `jlink`, among others, are available in the bin directory of your JDK installation. There are two ways you could run these tools from the command prompt:

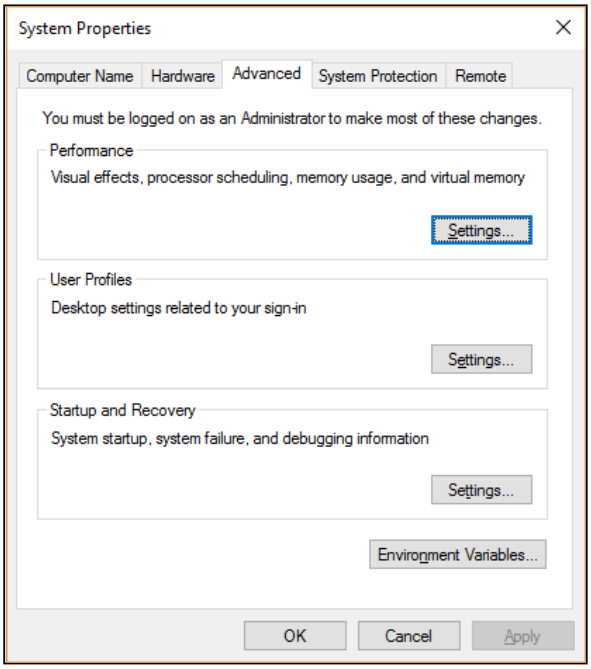
1. Navigate to the directory where the tools are installed and then run them, as follows:

```
cd "C:\Program Files\Java\jdk-9\bin"
javac -version
```

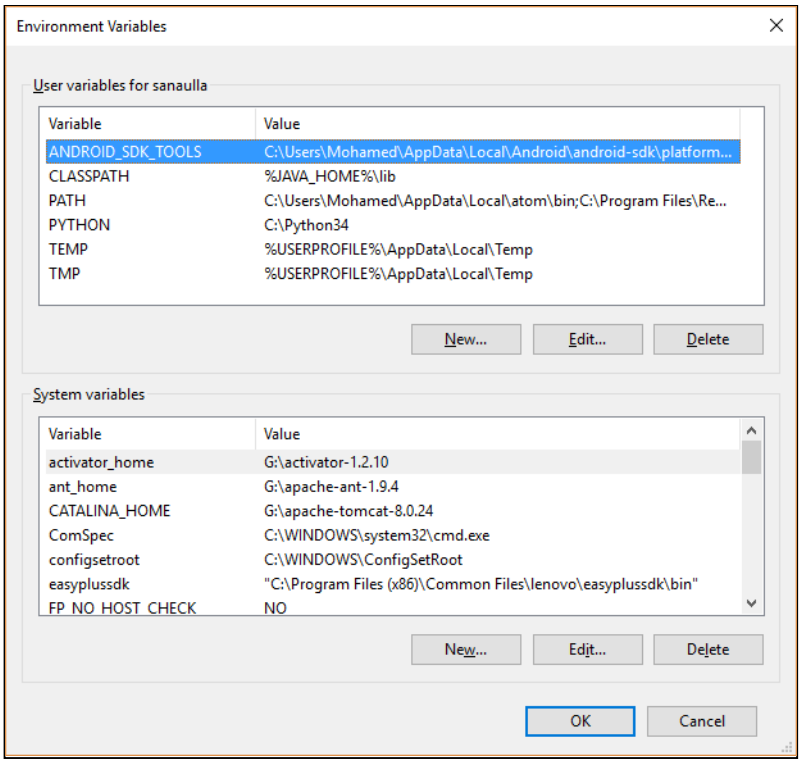
2. Export the path to the directory so that the tools are available from any directory in the command prompt. To achieve this, we have to add the path to the JDK tools in the `PATH` environment variable. The command prompt will search for the relevant tool in all the locations declared in the `PATH` environment variable.

Let's see how you can add the JDK bin directory to the `PATH` variable:

1. Right click on My Computer and then click on Properties. You will see your system information. Search for Advanced system settings and click on it to get a window, as shown in the following screenshot:

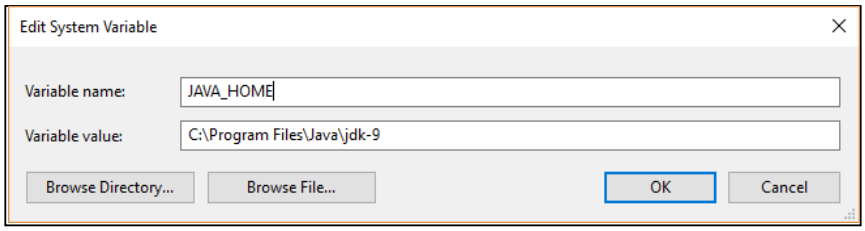


2. Click on Environment Variables to view the variables defined in your system. You will see that there are quite a few environment variables already defined, as shown in the following screenshot (the variables will differ across systems; in the following screenshot, there are a few predefined variables and a few variables added by me):

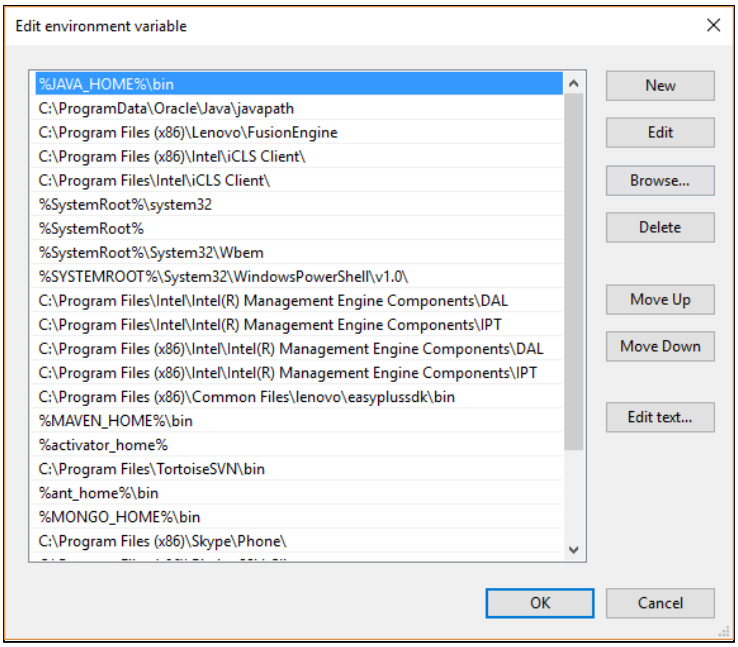


The variables defined under System variables are available across all the users of the system, and those defined under User variables for sanaulla are available only to the user, `sanaulla`.

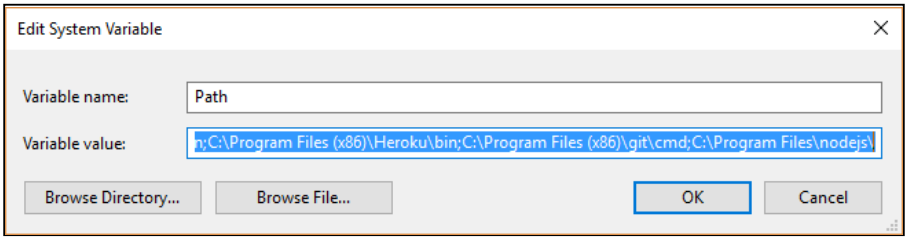
3. Click on New under User variables for <your username> to add a new variable, with the name `JAVA_HOME`, and its value as the location of the JDK 9 installation. For example, it would be `C:/Program Files/Java/jdk-9` (for 64 bit) or `C:/Program Files (x86)/Java/jdk-9` (for 32 bit):



4. The next step is to update the `PATH` environment variable with the location of the bin directory of your JDK installation (defined in the `JAVA_HOME` environment variable). If you already see the `PATH` variable defined in the list, then you need to select that variable and click on Edit. If the `PATH` variable is not seen, then click on New.
5. Any of the actions in the previous step will give you a popup, as shown in the following screenshot (on Windows 10):



The following image shows the other Windows versions:



6. You can either click on New in the first picture and insert the value, `%JAVA_HOME%/bin`, or you can append the value against the Variable value field by adding `;` `%JAVA_HOME%/bin`. The semicolon (`;`) in Windows is used to separate multiple values for a given variable name.
7. After setting the values, open the command prompt and then run `javac -version`, and you should be able to see `javac 9-ea` as the output. If you don't see it, then it means that the bin directory of your JDK installation has not been correctly added to the `PATH` variable.

Installing JDK 9 on Linux (Ubuntu, x64) and configuring the PATH variable

In this recipe, we will look at installing JDK on Linux (Ubuntu, x64) and also how to configure the `PATH` variable to make the JDK tools (such as `javac`, `java`, `jar`, and others) available from any location within the terminal.

How to do it...

1. Follow the Steps 1 and 2 of the *Installing JDK 9 on Windows and setting up the PATH variable* recipe to reach the downloads page.
2. Copy the download link (`tar.gz`) for the JDK for the Linux x64 platform from the downloads page.
3. Download the JDK by using `$> wget <copied link>`, for example, `$> wget http://download.java.net/java/jdk9/archive/180/binaries/jdk-9+180_linux-x64_bin.tar.gz`.
4. Once the download completes, you should have the relevant JDK available, for example, `jdk-9+180_linux-x64_bin.tar.gz`. You can list the contents by using `$> tar -tf jdk-9+180_linux-x64_bin.tar.gz`. You can even pipe it to `more` to paginate the output: `$> tar -tf jdk-9+180_linux-x64_bin.tar.gz | more`.
5. Extract the contents of the `tar.gz` file under `/usr/lib` by using `$> tar -xvzf jdk-9+180_linux-x64_bin.tar.gz -C /usr/lib`. This will extract the contents into a directory, `/usr/lib/jdk-9`. You can then list the contents of JDK 9 by using `$> ls /usr/lib/jdk-9`.

6. Update the `JAVA_HOME` and `PATH` variables by editing the `.bash_aliases` file in your Linux home directory:

```
$> vim ~/.bash_aliases
export JAVA_HOME=/usr/lib/jdk-9
export PATH=$PATH:$JAVA_HOME/bin
```

Source the `.bashrc` file to apply the new aliases:

```
$> source ~/.bashrc
$> echo $JAVA_HOME
/usr/lib/jdk-9
$>javac -version
javac 9
$> java -version
java version "9"
Java(TM) SE Runtime Environment (build 9+180)
Java HotSpot(TM) 64-Bit Server VM (build 9+180, mixed mode)
```



All the examples in this book are run against JDK installed on Linux (Ubuntu, x64), except for places where we have specifically mentioned that these are run on Windows. We have tried to provide run scripts for both platforms.
The recipes on JavaFX are completely executed on Windows.

Compiling and running a Java application

In this recipe, we will write a very simple modular `Hello world` program to test our JDK installation. This simple example prints `Hello world` in XML; after all it's the world of web services.

Getting ready

You should have JDK installed and the `PATH` variable updated to point to the JDK installation.

How to do it...

1. Let's define the model object with the relevant properties and annotations that will be serialized into XML:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
class Messages{
  @XmlElement
  public final String message = "Hello World in XML";
}
```

   In the preceding code, `@XmlRootElement` is used to define the root tag, `@XmlAccessorType` is used to define the type of source for the tag name and tag values, and `@XmlElement` is used to identify the sources that become the tag name and tag values in the XML.

2. Now, let's serialize an instance of the `Message` class into XML using JAXB:

```
public class HelloWorldXml{
  public static void main(String[] args) throws JAXBException{
    JAXBContext jaxb = JAXBContext.newInstance(Messages.class);
    Marshaller marshaller = jaxb.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FRAGMENT,
                           Boolean.TRUE);
    StringWriter writer = new StringWriter();
    marshaller.marshal(new Messages(), writer);
    System.out.println(writer.toString());
  }
}
```

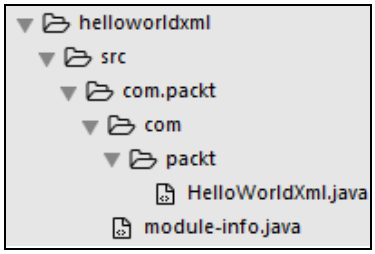
3. We will now create a module named `com.packt`. To create a module, we need to create a file named `module-info.java`, which contains the module definition. The module definition contains the dependencies of the module and the packages exported by the module to other modules:

```
module com.packt{
  //depends on the java.xml.bind module
  requires java.xml.bind;
  //need this for Messages class to be available to java.xml.bind
  exports  com.packt to java.xml.bind;
}
```



We will explain modules in detail in Chapter 3, Modular Programming. But this example is just to give you a taste of modular programming and also to test your JDK installation.

The directory structure with the preceding files is as follows:



4. Let's now compile and run the code. From the directory, `hellowordxml`, create a new directory in which to place your compiled class files:

```
mkdir -p mods/com.packt
```

Compile the source, `HelloWorldXml.java` and `module-info.java`, into the `mods/com.packt` directory:

```
javac -d mods/com.packt/ src/com.packt/module-info.java
src/com.packt/com/packt/HelloWorldXml.java
```

5. Run the compiled code by using `java --module-path mods -m com.packt/com.packt.HelloWorldXml`. You will see the following output:

```
<messages><message>Hello World in XML</message></messages>
```

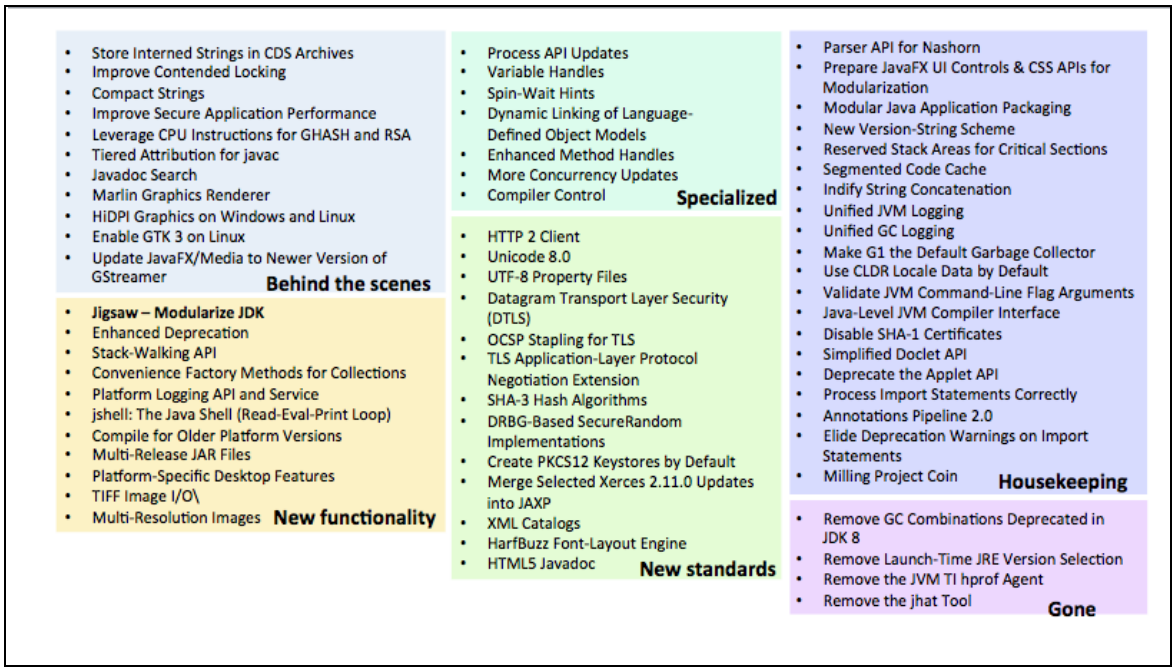
Do not worry if you are not able to understand the options passed with the `java` or `javac` commands. You will learn about them in Chapter 3, *Modular Programming*.

New features in Java 9

The release of Java 9 is a milestone in the Java ecosystem. The much awaited modular framework developed under Project Jigsaw will be part of this Java SE release. Another major feature in this is the JShell tool, which is an REPL tool for Java. Apart from this, there are other important API changes and JVM-level changes to improve the performance and debuggability of the JVM. In a blog post (https://blogs.oracle.com/java/jdk-9-categories), Yolande Poirier categorizes JDK 9 features into the following:

1. Behind the scenes
2. New functionality
3. Specialized
4. New standards
5. Housekeeping
6. Gone

The same blog post has summarized the preceding categorization into the following image:



In this recipe, we will discuss a few important features of JDK 9 and, wherever possible, also show a small code snippet of that feature in action. Every new feature in JDK is introduced by means of **JDK Enhancement Proposals**, also called **JEPs**. More information about the different JEPs part of JDK 9 and the release schedule of JDK 9 can be found on the official project page:

http://openjdk.java.net/projects/jdk9/.

How to do it...

We have picked a few features, which we feel are amazing and worth knowing about. In the following few sections, we'll briefly introduce you to those features.

JEP 102 -- Process API updates

Java's Process API has been quite primitive, with support only to launch new processes, redirect the processes' output, and error streams. In this release, the updates to the Process API enable the following:

- Get the PID of the current JVM process and any other processes spawned by the JVM
- Enumerate the processes running in the system to get information such as PID, name, and resource usage
- Managing process trees
- Managing sub processes

Let's look at a sample code, which prints the current PID as well as the current process information:

```
//NewFeatures.java
public class NewFeatures{
  public static void main(String [] args) {
    ProcessHandle currentProcess = ProcessHandle.current();
    System.out.println("PID: " + currentProcess.getPid());
    ProcessHandle.Info currentProcessInfo = currentProcess.info();
    System.out.println("Info: " + currentProcessInfo);
  }
}
```

JEP 110 -- HTTP/2 client



This feature is being included in the incubator module. This means that the feature is expected to change in the subsequent releases and may even be removed completely. So, we advise you to use this on an experimental basis.

Java's HTTP API has been the most primitive. Developers often resort to using third-party libraries, such as Apache HTTP, RESTlet, Jersey, and so on. In addition to this, Java's HTTP API predates the HTTP/1.1 specification and is synchronous and hard to maintain. These limitations called for the need to add a new API. The new HTTP client API provides the following:

- A simple and concise API to deal with most HTTP requests
- Support for HTTP/2 specification
- Better performance
- Better security
- A few more enhancements

Let's see a sample code to make an HTTP GET request using the new APIs. Below is the module definition defined within the file `module-info.java`:

```
//module-info.java
module newfeatures{
    requires jdk.incubator.httpclient;
}
```

The following code uses the HTTP Client API, which is part of `jdk.incubator.httpclient` module:

```
import jdk.incubator.http.*;
import java.net.URI;
public class Http2Feature{
  public static void main(String[] args) throws Exception{
```

```
    HttpClient client = HttpClient.newBuilder().build();
    HttpRequest request = HttpRequest
      .newBuilder(new URI("http://httpbin.org/get"))
      .GET()
      .version(HttpClient.Version.HTTP_1_1)
      .build();
    HttpResponse<String> response = client.send(request,
      HttpResponse.BodyHandler.asString());
    System.out.println("Status code: " + response.statusCode());
    System.out.println("Response Body: " + response.body());
  }
}
```

JEP 213 -- milling project coin

In Java SE 7, _ were introduced as part of the numbers, whereby a large number could be conveniently written by introducing _ between the digits. This helped in increasing the readability of the number, for example:

```
Integer large_Number = 123_123_123;
System.out.println(large_Number);
```

In Java SE 8, the use of _ in the variable names, as shown earlier, resulted in a warning, but in Java SE 9, this use results in an error, which means that the variables can no longer have _ in their names.

The other changed part of this JEP is to support private methods in interfaces. Java started with interfaces with absolutely no method implementations. Then, Java SE 8 introduced default methods that allowed interfaces to have methods with implementations, called default methods. So any class implementing this interface could choose not to override the default methods and use the implementation provided in the interface.

Java SE 9 is introducing private methods, wherein the default methods in the interfaces can share code between them by refactoring the common code into private methods.

Another useful feature is the allowing of effectively final variables to be used with try-with-resources. As of Java SE 8, we needed to declare a variable within the try-with-resources block, such as the following:

```
try(Connection conn = getConnection()){}catch(Exception ex){}.
```

However, with Java SE 9, we can do the following:

```
try(conn){}catch(Exception ex){}
```

Here, `conn` is effectively final; that is, it has been declared and defined before, and will never be reassigned during out the course of the program

execution.

JEP 222: jshell -- the Java shell (Read-Eval-Print Loop)

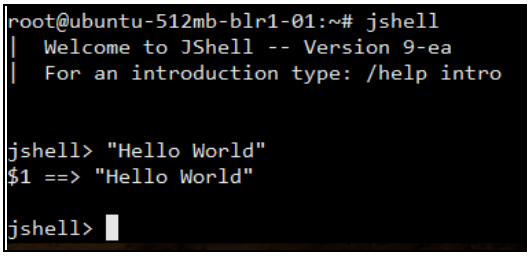
You must have seen languages, such as Ruby, Scala, Groovy, Clojure, and others shipping with a tool, which is often called **REPL** (**Read-Eval-Print-Loop**). This REPL tool is extremely useful in trying out the language features. For example, in Scala, we can write a simple `Hello World` program as

```
scala> println("Hello World");
```

Some of the advantages of the JShell REPL are as follows:

- Help language learners to quickly try out the language features
- Help experienced developers to quickly prototype and experiment before adopting it in their main code base
- Java developers can now boast of an REPL

Let's quickly spawn our command prompts/terminals and run the JShell command, as shown in the following image:



There is a lot more we can do, but we will keep that for `Chapter 13`, *The Read-Evaluate-Print Loop (REPL) Using JShell*.

JEP 238 -- multi-release JAR files

As of now, JAR files can contain classes that can only run on the Java version they were compiled for. To leverage the new features of the Java platform on newer versions, the library developers have to release a newer version of their library. Soon, there will be multiple versions of the library being maintained by the developers, which can be a nightmare. To overcome this limitation, the new feature of multirelease JAR files allows developers to build JAR files with different versions of class files for different Java versions. The following example makes it more clear.

Here is an illustration of the current JAR files:

```
jar root
  - A.class
  - B.class
  - C.class
```

Here is how multirelease JAR files look:

```
jar root
  - A.class
  - B.class
  - C.class
  - META-INF
    - versions
      - 9
        - A.class
      - 10
        - B.class
```

In the preceding illustration, the JAR files support class files for two Java versions--9 and 10. So, when the earlier JAR is executed on Java 9, the `A.class` under the `versions - 9` folder is picked for execution. On a platform that doesn't support multirelease JAR files, the classes under the versions directory are never used. So, if you run the multirelease JAR file on Java 8, it's as good as running a simple JAR file.

JEP 266 -- more concurrency updates

In this update, a new class, `java.util.concurrent.Flow`, has been introduced, which has nested interfaces supporting the implementation of a publish-subscribe framework. The publish-subscribe framework enables developers to build components that can asynchronously consume a live stream of data by setting up publishers that produce the data and subscribers that consume the data via subscription, which manages them. The four new interfaces are as follows:

- `java.util.concurrent.Flow.Publisher`
- `java.util.concurrent.Flow.Subscriber`
- `java.util.concurrent.Flow.Subscription`
- `java.util.concurrent.Flow.Processor` (which acts as both `Publisher` and `Subscriber`).

Project Jigsaw

The main aim of this project is to introduce the concept of modularity; support for creating modules in Java and then apply the same to the JDK; that is, modularize the JDK. Some of the benefits of modularity are as follows:

- Stronger encapsulation: The modules can access only those parts of the module that have been made available for use. So, the public classes in a package are not public unless the package is explicitly exported in the module info file. This encapsulation cannot be broken by using reflection (except in cases where the module is an open module or specific packages in the module have been made open).
- Clear dependencies: Modules must declare which other modules they would be using via the `requires` clause.
- Combining modules to create a smaller runtime, which can be easily scaled to smaller computing devices.
- Make the applications more reliable by eliminating runtime errors. For example, you must have experienced your application failing during runtime due to missing classes, resulting in `ClassNotFoundException`.

There are various JEPs, which are part of this project, as follows:

- **JEP 200 - modular JDK**: This applies the Java platform module system to modularize the JDK into a set of modules that can be combined at compile time, build time, or runtime.
- **JEP 201 - modular source code**: This modularizes the JDK source code into modules and enhances the build tools to compile the modules.
- **JEP 220 - modular runtime images**: This restructures the JDK and JRE runtime images to accommodate modules and to improve performance, security, and maintainability.
- **JEP 260 - encapsulate most internal APIs**: This allows a lot of internal APIs to be accessed directly or via reflection. Accessing

internal APIs that are bound to change is quite risky. To prevent its use, they are being encapsulated into modules and only those internal APIs that are widely used are being made available until a proper API is in its place.

- **JEP 261 - module system**: This implements the module system Java specification by changing the Java programming language, JVM, and other standard APIs (`http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html`). This includes the introduction of a new construct called module, `{ }`, with its supported keywords, such as `requires`, `exports`, `opens`, and `uses`.
- **JEP 282: jlink, the Java linker**: This allows packaging modules and their dependencies into smaller run times.

More details about Project Jigsaw can be found from the Project Jigsaw homepage (`http://openjdk.java.net/projects/jigsaw/`).

There's more...

There are quite a few features listed that are significant for developers, and we thought of grouping them together for your benefit:

- Enhance the Javadoc tool to generate HTML 5 output and the generated Javadoc should support the local search for classes and other elements.
- Make G1 as the default garbage collector and remove GC combinations that have been deprecated in Java 8. G1 is the new garbage collector (which has been in existence since Java SE 7), which focuses on reducing the pause times of the garbage collection activity. These pause times are very critical to latency-critical applications and, hence, such applications are going towards adopting the new garbage collector.
- Changing the internal representation of `String` to make use of a byte array rather than a character array. In a character array, each array element is 2 bytes, and it was observed that a majority of strings use 1 byte. This resulted in wasteful allocation. The new representation would also introduce a flag to indicate the type of encoding used.
- The new stackwalking API to support navigating the stack trace, which will help to do much more than just print the stack trace.
- Allow the image I/O plugin to support the TIFF image format.

Using new tools in JDK 9

There are a few new command-line tools introduced in JDK 9 to support new features. We will give you a quick overview of these tools and the same will be explained with recipes of their own in the later chapters.

Getting ready

You should have JDK 9 installed and the `PATH` environment variable updated to add the path to the `bin` directory of your JDK installation. Also, you need to have tried out `HelloWorldXml` explained in the recipe, *Compiling and running a Java application*.

How to do it...

We will look at a few interesting new command-line tools introduced.

jdeprscan

This tool is used for scanning the usage of deprecated APIs in a given JAR file, classpath, or source directory. Suppose we have a simple class that makes use of the deprecated method, `addItem`, of the `java.awt.List` class, as follows:

```
import java.awt.List;
public class Test{
  public static void main(String[] args){
    List list = new List();
    list.addItem("Hello");
  }
}
```

Compile the preceding class and then use `jdeprscan`, as follows:

```
C:Program FilesJavajdk-9bin>jdeprscan.exe -cp . Test
```

You will notice that this tool prints out `class Test uses method java/awt/List addItem (Ljava/lang/String;)V deprecated`, which is exactly what we expected.

jdeps

This tool analyses your code base specified by the path to the `.class` file, directory, or JAR, lists the package-wise dependency of your application, and also lists the JDK module in which the package exists. This helps in identifying the JDK modules that the application depends on and is the first step in migrating to modular applications.

We can run the tool on our `HelloWorldXml` example written earlier and see what `jdeps` provides:

```
$> jdeps mods/com.packt/
com.packt -> java.base
com.packt -> java.xml.bind
com.packt -> java.io                          java.base
com.packt -> java.lang                        java.base
com.packt -> javax.xml.bind                   java.xml.bind
com.packt -> javax.xml.bind.annotation        java.xml.bind
```

jlink

This tool is used to select modules and create a smaller runtime image with the selected modules. For example, we can create a runtime image by adding the `com.packt` modules created in our `HelloWorldXml` example:

```
$> jlink --module-path mods/:$JAVA_HOME/jmods/ --add-modules com.packt --output
img
```

Looking at the contents of the `img` folder, we should find that it has the `bin`, `conf`, `include`, and `lib` directories. We will learn more about `jlink` under `Chapter 3`, *Modular Programming*.

jmod

JMOD is a new format for packaging your modules. This format allows including native code, configuration files, and other data that do not fit into JAR files. The JDK modules have been packaged as JMOD files.

The `jmod` command-line tool allows `create`, `list`, `describe`, and `hash` JMOD files:

- `create`: This is used to create a new `jmod` file
- `list`: This is used to list the contents of a `jmod` file
- `describe`: This is used to describe module details
- `hash`: This is used to record hashes of tied modules

JShell

This tool has been briefly explained earlier, under the title, *JEP 222: jshell - the Java shell (Read-Eval-Print Loop)*

Comparing JDK 8 and JDK 9

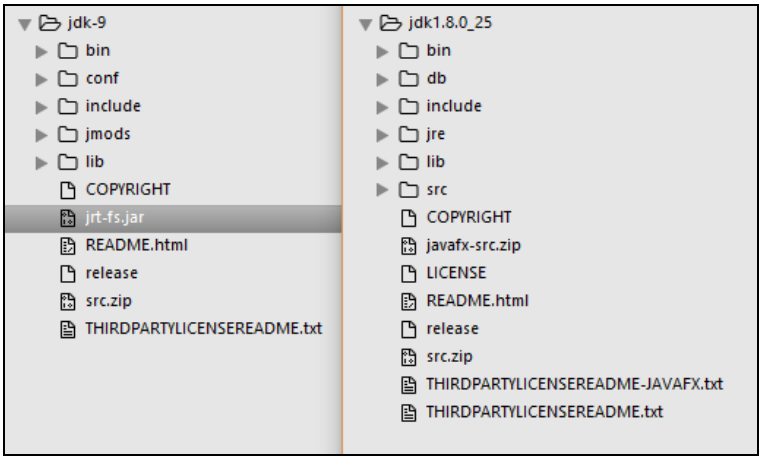
Due to the application of a modular system to JDK under Project Jigsaw, there have been a few changes in the JDK directory structure installed in your systems. In addition to these, there were a few changes undertaken to fix the JDK installation structure, which dates back to the times of Java 1.2. This has been deemed to be a golden opportunity by the JDK team to fix the issues with the JDK directory structure.

Getting ready

To see the difference in the JDK 9 directory structure, you will need to install a pre-JDK 9 version. We have chosen to use JDK 8 to compare with JDK 9. So, go ahead and install JDK 8 before you proceed further.

How to do it...

1. We did a side-by-side comparison of both the JDK installation directories as shown in the following:



2. Following are our observations from the preceding comparison:
    - The `jre` directory has been completely removed and has been replaced by `jmods` and `conf`. The `jmods` directory contains the runtime images of the JDK modules, the `conf` directory contains configuration and property files, which were earlier under the `jre` directory.
    - The contents of `jrebin` and `jrelib` have been moved to the lib and bin directories of JDK installation.

See also

Refer to the following recipe of this chapter:

- Using new tools in JDK 9

Fast Track to OOP - Classes and Interfaces

In this chapter, we will cover the following recipes:

- Implementing object-oriented design using classes
- Using inner classes
- Using inheritance and composition to make the design extensible
- Coding to an interface
- Creating interfaces with default and static methods
- Creating interfaces with private methods
- Using enums to represent constant entities
- Using the @Deprecated annotation to deprecate APIs
- Using HTML5 in Javadocs

Introduction

This chapter gives you a quick introduction to the components of OOP and covers the new enhancements in these components in Java 8 and Java 9. We will also try to cover a few good **object-oriented design** (**OOD**) practices wherever applicable.

Throughout the recipes, we will use the new (introduced in Java 8 and Java 9) enhancements, define and demonstrate the concepts of OOD using specific code examples, and present new capabilities for better code documentation.

One can spend many hours reading articles and practical advice on OOD in books and on the Internet. Some of this activity can be beneficial for some people. But, in our experience, the fastest way to get hold of OOD is to try its principles early in your own code. This is exactly the goal of this chapter: to give you a chance to see and use the OOD principles so that the formal definition makes sense immediately.

One of the main criteria of well-written code is its clarity of expressing its intent. A well-motivated and clear design helps achieve this. The code is run by a computer, but it is maintained and extended by humans. Keeping this in mind will assure longevity of the code written by you and perhaps even a few thanks and mentions with appreciation.

In this chapter, you will learn how to use the five basic OOD concepts:

- Object/Class - Keeping data and procedures together
- Encapsulation - Hiding data and/or procedures
- Inheritance - Extending another class data and/or procedures
- Interface - Deferring the implementation and coding for a type
- Polymorphism - Using the base class type for all its extensions when a parent class reference is used to refer to a child class object

These concepts will be defined and demonstrated in the code snippets presented in this chapter. If you search the Internet, you may notice that many other concepts and additions to them can be derived from the five points we just discussed.

Although the following text does not require prior knowledge of OOD, some experience of writing code in Java would be beneficial. The code samples are fully functional and compatible with Java 9. For better understanding, we recommend that you try to run the presented examples.

We also encourage you to adapt the tips and recommendations in this chapter to your needs in the context of your team experience. Consider sharing your new-found knowledge with your colleagues and discuss how the described principles can be applied to your domain, for your current project.

Implementing object-oriented design using classes

In this recipe, you will learn about the first two OOD concepts: object/class and encapsulation.

Getting ready

The term *object* usually refers to the coupling of data and procedures that can be applied to these data. Neither data, nor procedures are required, but one of them is--and, typically, both are--always present. The data are called object properties, while procedures are called methods. Properties capture the state of the object. Methods describe objects' behavior. Every object has a type, which is defined by its class (see the following). An object also is said to be an instance of a class.



The term class is a collection of the definitions of properties and methods that will be present in each of its instances--the objects created based on this class.
Encapsulation is the hiding of object properties and methods that should not be accessible by other objects.

Encapsulation is achieved by the Java keywords `private` or `protected` in the declaration of properties and methods.

How to do it...

1. Create an `Engine` class with the `horsePower` property, the `setHorsePower()` method that sets this property's value, and the `getSpeedMph()` method that calculates the speed of a vehicle, based on the time since the vehicle started moving, the vehicle weight, and the engine power:

```
public class Engine {
  private int horsePower;
  public void setHorsePower(int horsePower) {
    this.horsePower = horsePower;
  }
  public double getSpeedMph(double timeSec,
                            int weightPounds) {
    double v = 2.0*this.horsePower*746;
    v = v*timeSec*32.17/weightPounds;
    return Math.round(Math.sqrt(v)*0.68);
  }
}
```

2. Create the `Vehicle` class:

```
public class Vehicle {
  private int weightPounds;
  private Engine engine;
  public Vehicle(int weightPounds, Engine engine) {
    this.weightPounds = weightPounds;
    this.engine = engine;
  }
  public double getSpeedMph(double timeSec){
    return this.engine.getSpeedMph(timeSec, weightPounds);
  }
}
```

3. Create the application that will use these classes:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int horsePower = 246;
  int vehicleWeight = 4000;
  Engine engine = new Engine();
  engine.setHorsePower(horsePower);
  Vehicle vehicle = new Vehicle(vehicleWeight, engine);
  System.out.println("Vehicle speed (" + timeSec + " sec)=" +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

How it works...

The preceding application produces the following output:

```
Vehicle speed (10.0 sec) = 117.0 mph
```

As you can see, an `engine` object was created by invoking the default constructor of the `Engine` class without parameters and with the Java keyword `new` that allocated memory for the newly created object on the heap.

The second object, namely `vehicle`, was created with the explicitly defined constructor of the `Vehicle` class with two parameters. The second parameter of the constructor is an `engine` object that carries the `horsePower` property with the value set as `246`, using the `setHorsePower()` method.

It also contains the `getSpeedMph()` method that can be called by any other object that has access to the `engine` object, as it is done in the `getSpeedMph()` method of the `Vehicle` class.

It's worth noticing that the `getSpeedMph()` method of the `Vehicle` class relies on the presence of a value assigned to the `engine` property. The object of the `Vehicle` class delegates the speed calculation to the object of the `Engine` class. If the latter is not set (`null` passed in the `Vehicle()` constructor, for example), we will get `NullPointerException` at runtime. To avoid this, we can place a check for the presence of this value in the `Vehicle()` constructor:

```
if(engine == null){
   throw new RuntimeException("Engine" + " is required parameter.");
}
```

Alternatively, we can place a check in the `getSpeedMph()` method of the `Vehicle` class:

```
if(getEngine() == null){
  throw new RuntimeException("Engine value is required.");
```

```
}
```

This way, we avoid the ambiguity of `NullPointerException` and tell the user exactly what the source of the problem was.

As you may have noticed, the `getSpeedMph()` method can be removed from the `Engine` class and fully implemented in the `Vehicle`: class:

```
public double getSpeedMph(double timeSec){
  double v = 2.0 * this.engine.getHorsePower() * 746;
  v = v * timeSec * 32.174 / this.weightPounds;
  return Math.round(Math.sqrt(v) * 0.68);
}
```

To do this, we would need to add the public method `getHorsePower()` to the `Engine` class in order to make it available for usage by the `getSpeedMph()` method in the `Vehicle` class. For now, we leave the `getSpeedMph()` method in the `Engine`. class.

This is one of the design decisions you need to make. If you think that an object of the `Engine` class is going to be passed around to the objects of different classes (not only `Vehicle`), then you would need to keep the `getSpeedMph()` method in the `Engine` class. Otherwise, if you think that the `Vehicle` class is going to be responsible for the speed calculation (which makes sense, since it is the speed of a vehicle, not of an engine), then you should implement the method inside the `Vehicle` class.

There's more...

Java provides a capability to extend a class and allow the subclass to access all of the functionality of the base class. For example, you can decide that every object that could be asked about its speed belongs to a subclass that is derived from the `Vehicle` class. In such a case, `Car` may look like this:

```
public class Car extends Vehicle {
  private int passengersCount;
  public Car(int passengersCount, int weightPounds,
             Engine engine){
    super(weightPounds, engine);
    this.passengersCount = passengersCount;
  }
  public int getPassengersCount() {
    return this.passengersCount;
  }
}
```

Now we can change our test code by replacing the `Vehicle` class with the `Car` class:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int horsePower = 246;
  int vehicleWeight = 4000;
  Engine engine = new Engine();
  engine.setHorsePower(horsePower);
  Vehicle vehicle = new Car(4, vehicleWeight, engine);
  System.out.println("Car speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

When run, it produces the same value as with an object of the `Vehicle` class:

```
Car speed (10.0 sec) = 117.0 mph
```

Because of polymorphism, a reference to the object of the `Car` class can be assigned to the reference of the base class, that is, `Vehicle`. The object of the `Car` class has two types: its own type, that is, `Car` and the type of the base class, namely `Vehicle`.

In Java, a class can also implement multiple interfaces, and the object of such a class would have a type of each of the implemented interfaces too. We will talk about this in subsequent recipes.

There are usually several ways to design an application for the same functionality. It all depends on the needs of your project and the style adopted by the development team. But in any context, clarity of design will help you communicate the intent. Good design contributes to the quality and longevity of the code.

See also

Refer to the following recipe in this chapter:

- Using inheritance and composition to make the design extensible

Using inner classes

In this recipe, you will learn about three types of inner classes:

- **Inner class**: This is a class defined inside another (enclosing) class. Its accessibility from outside the enclosing class is regulated by the `public`, `protected`, and `private` keywords. It can access the private members of the enclosing class, and the enclosing class can access the private members of its inner class.
- **Method-local inner class**: This is a class defined inside a method. Its scope is restricted to within the method.
- **Anonymous inner class**: This is an anonymous class defined during object instantiation.

Getting ready

When a class is used by one, and only one, other class, the designer might decide that there is no need to make such a class public. For example, let's assume that the Engine class is used by the `Vehicle` class only.

How to do it...

1. Create the `Engine` class as an inner class of the `Vehicle` class:

```
public class Vehicle {
  private int weightPounds;
  private Engine engine;
  public Vehicle(int weightPounds, int horsePower) {
    this.weightPounds = weightPounds;
    this.engine = new Engine(horsePower);
  }
  public double getSpeedMph(double timeSec){
    return this.engine.getSpeedMph(timeSec);
  }
  private int getWeightPounds(){ return weightPounds; }
  private class Engine {
    private int horsePower;
    private Engine(int horsePower) {
      this.horsePower = horsePower;
    }
    private double getSpeedMph(double timeSec){
      double v = 2.0 * this.horsePower * 746;
      v = v * timeSec * 32.174 / getWeightPounds();
      return Math.round(Math.sqrt(v) * 0.68);
    }
  }
}
```

2. Notice that the `getSpeedMph()` method of the `Vehicle` class has access to the `Engine` class (although it is declared `private`) and even to the private `getSpeedMph()` method of the `Engine` class. The inner class has access to all the private elements of the enclosing class too. This is why `getSpeedMph()` of the `Engine` class has access to the private `getWeightPounds()` method of the enclosing `Vehicle` class.
3. Look closer at the usage of the inner class `Engine`. Only the `getSpeedMph()` method uses it. If the designer believes that it is going to be the case in future too, it would be reasonable to make it method-local inner class, which is the second type of an inner class:

```
public class Vehicle {
  private int weightPounds;
  private int horsePower;
  public Vehicle(int weightPounds, int horsePower) {
    this.weightPounds = weightPounds;
    this.horsePower = horsePower;
  }
```

```
    private int getWeightPounds() { return weightPounds; }
    public double getSpeedMph(double timeSec){
      class Engine {
        private int horsePower;
        private Engine(int horsePower) {
          this.horsePower = horsePower;
        }
        private double getSpeedMph(double timeSec){
          double v = 2.0 * this.horsePower * 746;
          v = v * timeSec * 32.174 / getWeightPounds();
          return Math.round(Math.sqrt(v) * 0.68);
        }
      }
      Engine engine = new Engine(this.horsePower);
      return engine.getSpeedMph(timeSec);
    }
  }
```

Encapsulation--hiding the state and behavior of objects--helps avoid unexpected side effects resulting from an accidental change or overriding. It makes the behavior more predictable and easier to understand. That's why a good design exposes only the functionality that must be accessible from the outside. Typically, this is the main functionality that motivated the class creation in the first place.

How it works...

Whether the `Engine` class is implemented as an inner class or a method-local inner class, the test code looks the same:

```
public static void main(String arg[]) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;
  Vehicle vehicle = new Vehicle(vehicleWeightPounds, engineHorsePower);
  System.out.println("Vehicle speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

If we run this program, we get the same output:

```
Car speed (10.0 sec) = 117.0 mph
```

Now, let's assume we need to test different implementations of the `getSpeedMph()` method:

```
public double getSpeedMph(double timeSec){ return -1.0d; }
```

If this speed calculation formula does not make sense to you, you are correct. It does not. We did it for making the result predictable and different from the result of the previous implementation.

There are many ways to introduce this new implementation. We can change the implementation of the `getSpeedMph()` method in the `Engine` class, for example. Or, we can change the implementation of the same method in the `Vehicle` class.

In this recipe, we will do this using the third type of inner class called anonymous inner class. This approach is especially handy when you want to write as little new code as possible or you want to quickly test the new behavior by temporarily overriding the old one. The code would then look like this:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;
  Vehicle vehicle = new Vehicle(vehicleWeightPounds, engineHorsePower) {
    public double getSpeedMph(double timeSec){
      return -1.0d;
    }
  };
  System.out.println("Vehicle speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

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

```
Vehicle speed (10.0 sec) = -1.0 mph
```

We have overridden the `Vehicle` class implementation by leaving only one method in it--the `getSpeedMph()` method that returns hardcoded value. We could override other methods or add new ones, but we will keep it simple for demonstration purposes.

By definition, anonymous inner class has to be an expression that is part of a statement that ends (as any statement) with a semicolon. The expression is composed of the following:

- The `new` operator
- The name of the implemented interface (followed by parentheses `()` that represent the default constructor) or a constructor of the extended class (the latter is our case, the extended class being `Vehicle`)
- The class body with method declarations (statements are not allowed in the body of an anonymous inner class)

Like any inner classes, anonymous inner class can access any member of the enclosing class and can capture the values of its variables. To be able to do this, these variables have to be declared `final`. Otherwise, they become `final` implicitly, which means their values cannot be changed (a good modern IDE will warn you about the violation of this constraint if you try to change such a value).

Using these features, we can modify our sample code and provide more input data for the newly implemented `getSpeedMph()` method without passing them as method parameters:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;
  Vehicle vehicle = new Vehicle(vehicleWeightPounds, engineHorsePower){
    public double getSpeedMph(double timeSec){
      double v = 2.0 * engineHorsePower * 746;
      v = v * timeSec * 32.174 / vehicleWeightPounds;
      return Math.round(Math.sqrt(v) * 0.68);
    }
  };
  System.out.println("Vehicle speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

Notice that the variables `timeSec`, `engineHorsePower`, and `vehicleWeightPounds` are accessible by the `getSpeedMph()` method of the inner class and cannot be modified. If we run this code, the result will be the same as before:

```
Car speed (10.0 sec) = 117.0 mph
```

In the case of an interface with only one abstract method (called functional interface), a particular type of anonymous inner class, called **lambda expression**, it allows you to have a shorter notation but provides the interface implementation. We are going to discuss functional interface and lambda expression in the next chapter.

There's more...

An inner class is a non-static nested class. Java also allows you to create a static nested class that can be used when an inner class does not require access to non-public properties and methods of the enclosing class. Here is an example (the keyword `static` is added to the `Engine` class):

```
public class Vehicle {
  private Engine engine;
  public Vehicle(int weightPounds, int horsePower) {
    this.engine = new Engine(horsePower, weightPounds);
  }
  public double getSpeedMph(double timeSec){
    return this.engine.getSpeedMph(timeSec);
  }
  private static class Engine {
    private int horsePower;
    private int weightPounds;
    private Engine(int horsePower, int weightPounds) {
      this.horsePower = horsePower;
    }
    private double getSpeedMph(double timeSec){
      double v = 2.0 * this.horsePower * 746;
      v = v * timeSec * 32.174 / this.weightPounds;
      return Math.round(Math.sqrt(v) * 0.68);
    }
  }
}
```

Because a static class couldn't access a non-static member (the `getWeightPounds()` method of the enclosing class `Vehicle`), we were forced to pass the weight value to the `Engine` class during its construction (and we removed the `getWeightPounds()` method as it was not needed anymore).

See also

Refer to the following recipe in this chapter:

- Going functional

Using inheritance and composition to make the design extensible

In this recipe, you will learn about two important OOD concepts, namely Inheritance and Polymorphism, which have been mentioned already and used in the examples of the previous recipes.

Getting ready

Inheritance is the ability of one class to extend (and, optionally, override) the properties and/or methods of another class. The extended class is called the base class, superclass, or parent class. The new extension of the class is called a subclass or child class.



Polymorphism is the ability to use the base class as a type for the references to the objects of its subclasses.

To demonstrate the power of these two concepts, let's create classes that represent cars and trucks, each having weight, engine power, and speed it can reach (as a function of time) with maximum load. In addition, a car, in this case, will be characterized by the number of passengers, while a truck's important feature will be its payload.

How to do it...

1. Look at the `Vehicle` class:

```
public class Vehicle {
  private int weightPounds, horsePower;
  public Vehicle(int weightPounds, int horsePower) {
    this.weightPounds = weightPounds;
    this.horsePower = horsePower;
  }
  public double getSpeedMph(double timeSec){
    double v = 2.0 * this.horsePower * 746;
    v = v * timeSec * 32.174 / this.weightPounds;
    return Math.round(Math.sqrt(v) * 0.68);
  }
}
```

   There is an obvious commonality between a car and a truck that can be encapsulated in the `Vehicle` class as the base class.

2. Create a subclass, called `Car`:

```
public class Car extends Vehicle {
  private int passengersCount;
  public Car(int passengersCount, int weightPounds,
             int horsepower){
    super(weightPounds, horsePower);
    this.passengersCount = passengersCount;
  }
  public int getPassengersCount() {
    return this.passengersCount;
  }
}
```

3. Create another subclass, called `Truck`:

```
 public class Truck extends Vehicle {
   private int payload;
   public Truck(int payloadPounds, int weightPounds,
                int horsePower){
     super(weightPounds, horsePower);
     this.payload = payloadPounds;
   }
   public int getPayload() {
     return this.payload;
   }
 }
```

Since the base class `Vehicle` has neither an implicit or explicit constructor without parameters (because we have chosen to use an explicit constructor with parameters only), we will have to call the base class constructor `super()` as the first line of the constructor of every subclass.

How it works...

Let's write a test program:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;
  Vehicle vehicle = new Car(4, vehicleWeightPounds, engineHorsePower);
  System.out.println("Passengers count=" +
                     ((Car)vehicle).getPassengersCount());
  System.out.println("Car speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
  vehicle = new Truck(3300, vehicleWeightPounds, engineHorsePower);
  System.out.println("Payload=" +
                     ((Truck)vehicle).getPayload() + " pounds");
  System.out.println("Truck speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

Notice that the reference `vehicle` to the base class `Vehicle` points to the object of the subclass `Car`. This is made possible by polymorphism, according to which an object has a type of every class in its line of inheritance (including all the interfaces, which we will discuss a bit later).

One needs to cast such a reference to the subclass type, as you can see in the preceding example, in order to invoke a method that exists only in the subclass.

If we run the preceding example, the results will be as follows:

```
Passengers count=4
Car speed (10.0 sec) = 117.0 mph
Payload=3300 pounds
Truck speed (10.0 sec) = 117.0 mph
```

We should not be surprised to see the same speed, that is, `117.0 mph`, calculated for both--the car and the truck--because the same weight and engine power are used to calculate the speed of each. But, intuitively, we feel that a heavily loaded truck should not be able to reach the same speed as a car. To verify this, we need to include the total weight of the car (with

the passengers and their luggage) and the truck (with the payload) in the `getSpeedMph()` method. One way to do this is to override the `getSpeedMph()` method of the base class `Vehicle` in each of the subclasses.

Now, add the `horsePower` and `weightPounds` properties and the following method to the `Car` class (we assume that a passenger with a luggage weighs 250 pounds total on average):

```
public double getSpeedMph(double timeSec) {
  int weight = this.weightPounds + this.passengersCount * 250;
  double v = 2.0 * this.horsePower * 746;
  v = v * timeSec * 32.174 / weight;
  return Math.round(Math.sqrt(v) * 0.68);
}
```

Also, add the `horsePower` and `weightPounds` properties and the following method to the `Truck` class:

```
public double getSpeedMph(double timeSec) {
  int weight = this.weightPounds + this.payload;
  double v = 2.0 * this.horsePower * 746;
  v = v * timeSec * 32.174 / weight;
  return Math.round(Math.sqrt(v) * 0.68);
}
```

The results of these two additions (if we run the same test class) will be as follows:

```
Passengers count=4
Car speed (10.0 sec) = 105.0 mph
Payload=3300 pounds
Truck speed (10.0 sec) = 86.0 mph
```

These results do confirm our intuition: a fully loaded car or truck does not reach the same speed as an empty one.

The new methods in the subclasses override `getSpeedMph()` of the base class `Vehicle`, although we access it via the base class reference:

```
Vehicle vehicle =  new Car(4, vehicleWeightPounds, engineHorsePower);
System.out.println("Car speed (" + timeSec + " sec) = " +
                   vehicle.getSpeedMph(timeSec) + " mph");
```

The overridden method is dynamically bound, which means that the context of the method invocation is determined by the type of the actual object being referred to. Since, in our example, the reference `vehicle` points to an object of the subclass `Car`, the `vehicle.getSpeedMph()` construct invokes the method of the subclass, not the method of the base class.

There is obvious code redundancy in the two new methods, which we can refactor by creating a method in the base class, that is, `Vehicle`:

```
protected double getSpeedMph(double timeSec, int weightPounds) {
  double v = 2.0 * this.horsePower * 746;
  v = v * timeSec * 32.174 / weightPounds;
  return Math.round(Math.sqrt(v) * 0.68);
}
```

Since this method is used by subclasses only, it can be protected (and thus, accessible only to the subclasses).

Here's how the `getSpeedMph()` method of the `Car` subclass would look now:

```
public double getSpeedMph(double timeSec) {
  int weightPounds = this.weightPounds + this.passengersCount * 250;
  return getSpeedMph(timeSec, weightPounds);
}
```

This is how it will appear in the `Truck` subclass:

```
public double getSpeedMph(double timeSec) {
  int weightPounds = this.weightPounds + this.payload;
  return getSpeedMph(timeSec, weightPounds);
}
```

Now we need to modify the test class by adding casting. Otherwise, there will be a runtime error because the `getSpeedMph(int timeSec)` method does not exist in the base class `Vehicle`:

```
public static void main(String... arg) {
    double timeSec = 10.0;
    int engineHorsePower = 246;
    int vehicleWeightPounds = 4000;
    Vehicle vehicle = new Car(4, vehicleWeightPounds, engineHorsePower);
    System.out.println("Passengers count=" +
                       ((Car)vehicle).getPassengersCount());
    System.out.println("Car speed (" + timeSec + " sec) = " +
                       ((Car)vehicle).getSpeedMph(timeSec) + " mph");
    vehicle = new Truck(3300, vehicleWeightPounds, engineHorsePower);
    System.out.println("Payload=" +
```

```
                        ((Truck)vehicle).getPayload() + " pounds");
    System.out.println("Truck speed (" + timeSec + " sec) = " +
                        ((Truck)vehicle).getSpeedMph(timeSec) + " mph");
  }
}
```

As you may have expected, the test class produces the same values:

```
Passengers count=4
Car speed (10.0 sec) = 105.0 mph
Payload=3300 pounds
Truck speed (10.0 sec) = 86.0 mph
```

To simplify the test code, we can drop casting and write the following instead:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;
  Car car = new Car(4, vehicleWeightPounds, engineHorsePower);
  System.out.println("Passengers count=" + car.getPassengersCount());
  System.out.println("Car speed (" + timeSec + " sec) = " +
                     car.getSpeedMph(timeSec) + " mph");
  Truck truck = new Truck(3300, vehicleWeightPounds, engineHorsePower);
  System.out.println("Payload=" + truck.getPayload() + " pounds");
  System.out.println("Truck speed (" + timeSec + " sec) = " +
                     truck.getSpeedMph(timeSec) + " mph");
}
```

The speed values produced by this code remain the same.

Yet, there is an even simpler way to achieve the same effect. We can add the `getMaxWeightPounds()` method to the base class and each of the subclasses. The `Car` class will now look as follows:

```
public class Car extends Vehicle {
  private int passengersCount, weightPounds;
  public Car(int passengersCount, int weightPounds, int horsePower){
    super(weightPounds, horsePower);
    this.passengersCount = passengersCount;
    this.weightPounds = weightPounds;
  }
  public int getPassengersCount() {
    return this.passengersCount;
  }
  public int getMaxWeightPounds() {
    return this.weightPounds + this.passengersCount * 250;
  }
}
```

Here's how the new version of the `Truck` class will look:

```
public class Truck extends Vehicle {
  private int payload, weightPounds;
  public Truck(int payloadPounds, int weightPounds, int horsePower) {
    super(weightPounds, horsePower);
    this.payload = payloadPounds;
    this.weightPounds = weightPounds;
  }
  public int getPayload() { return this.payload; }
  public int getMaxWeightPounds() {
    return this.weightPounds + this.payload;
  }
}
```

We also need to add the `getMaxWeightPounds()` method to the base class so it can be used for the speed calculations:

```
public abstract class Vehicle {
  private int weightPounds, horsePower;
  public Vehicle(int weightPounds, int horsePower) {
    this.weightPounds = weightPounds;
    this.horsePower = horsePower;
  }
  public abstract int getMaxWeightPounds();
  public double getSpeedMph(double timeSec){
    double v = 2.0 * this.horsePower * 746;
    v = v * timeSec * 32.174 / getMaxWeightPounds();
    return Math.round(Math.sqrt(v) * 0.68);
  }
}
```

Adding an abstract method `getMaxWeightPounds()` to the `Vehicle` class makes the class abstract. This has a positive side effect: it enforces the implementation of the `getMaxWeightPounds()` method in each subclass.

The test class remains the same and produces the same results:

```
Passengers count=4
Car speed (10.0 sec) = 105.0 mph
Payload=3300 pounds
Truck speed (10.0 sec) = 86.0 mph
```

There is an even simpler code change for the same effect--to use the maximum weight in the speed calculations in the base class. If we get back to the original version of the classes, all we need to do is to pass the maximum weight to the constructor of the base class `Vehicle`. The resulting classes will look like this:

```
public class Car extends Vehicle {
  private int passengersCount;
  public Car(int passengersCount, int weightPounds, int horsepower){
    super(weightPounds + passengersCount * 250, horsePower);
    this.passengersCount = passengersCount;
  }
  public int getPassengersCount() {
    return this.passengersCount; }
}
```

We added the weight of the passengers to the value we pass to the constructor of the superclass; this is the only change in this subclass. There was a similar change in the `Truck` subclass:

```
public class Truck extends Vehicle {
  private int payload;
  public Truck(int payloadPounds, int weightPounds, int horsePower) {
    super(weightPounds + payloadPounds, horsePower);
    this.payload = payloadPounds;
  }
  public int getPayload() { return this.payload; }
}
```

The base class `Vehicle` remains the same:

```
public class Vehicle {
  private int weightPounds, horsePower;
  public Vehicle(int weightPounds, int horsePower) {
    this.weightPounds = weightPounds;
    this.horsePower = horsePower;
  }
  public double getSpeedMph(double timeSec){
    double v = 2.0 * this.horsePower * 746;
    v = v * timeSec * 32.174 / this.weightPounds;
    return Math.round(Math.sqrt(v) * 0.68);
  }
}
```

The test class does not change and produces the same results:

```
Passengers count=4
Car speed (10.0 sec) = 105.0 mph
Payload=3300 pounds
Truck speed (10.0 sec) = 86.0 mph
```

This last version--passing the maximum weight to the constructor of the base class--will now be the starting point for further demo code development.

Composition makes the design more extensible

In the preceding example, the speed model is implemented in the `getSpeedMph()` method of the `Vehicle` class. If we need to use a different speed model (which includes more input parameters and is more tuned to certain driving conditions, for example), we would need to change the `Vehicle` class or create a new subclass to override the method. In the case when we need to experiment with tens or even hundreds of different models, this approach becomes untenable.

Also, in real life, modeling based on machine learning and other advanced techniques becomes so involved and specialized, that it is quite common that the modeling of car acceleration is done by a different team, not the team that builds vehicles.

To avoid the proliferation of subclasses and code merge conflicts between vehicle builders and speed model developers, we can create a more extensible design using composition.



Composition is an OOD principle for implementing the necessary functionality using the behavior of classes that are not part of the inheritance hierarchy.

We can encapsulate the speed calculations inside the `SpeedModel` class in the `getSpeedMph()` method:

```
private Properties conditions;
  public SpeedModel(Properties drivingConditions){
    this.drivingConditions = drivingConditions;
  }
  public double getSpeedMph(double timeSec, int weightPounds,
                            int horsePower){
    String road = drivingConditions.getProperty("roadCondition","Dry");
    String tire = drivingConditions.getProperty("tireCondition","New");
    double v = 2.0 * horsePower * 746;
    v = v * timeSec * 32.174 / weightPounds;
    return Math.round(Math.sqrt(v) * 0.68)
```

```
                        - (road.equals("Dry") ? 2 : 5)
                        - (tire.equals("New") ? 0 : 5);
    }
}
```

An object of this class can be created and then set on the `Vehicle` class:

```
public class Vehicle {
   private SpeedModel speedModel;
   private int weightPounds, horsePower;
   public Vehicle(int weightPounds, int horsePower) {
      this.weightPounds = weightPounds;
      this.horsePower = horsePower;
   }
   public void setSpeedModel(SpeedModel speedModel){
      this.speedModel = speedModel;
   }
   public double getSpeedMph(double timeSec){
      return this.speedModel.getSpeedMph(timeSec,
                        this.weightPounds, this.horsePower);
   }
}
```

So, the test class may look like this:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int horsePower = 246;
  int vehicleWeight = 4000;
  Properties drivingConditions = new Properties();
  drivingConditions.put("roadCondition", "Wet");
  drivingConditions.put("tireCondition", "New");
  SpeedModel speedModel = new SpeedModel(drivingConditions);
  Car car = new Car(4, vehicleWeight, horsePower);
  car.setSpeedModel(speedModel);
  System.out.println("Car speed (" + timeSec + " sec) = " +
                      car.getSpeedMph(timeSec) + " mph");
}
```

The result will be as follows:

```
Car speed (10.0 sec) = 100.0 mph
```

We isolated the speed calculating functionality in a separate class and can now modify or extend it without changing any class of the `Vehicle` hierarchy. This is how the composition design principle allows you to change the behavior of the `Vehicle` class and its subclasses without changing their implementation.

In the next recipe, we will show how the OOD concept of Interface unlocks more power of composition and polymorphism, making the design simpler and even more expressive.

See also

Refer to the following recipes in this chapter:

- Coding to an interface
- Using enums to represent constant entities

Coding to an interface

In this recipe, you will learn about the last of the OOD concepts, called Interface, and further practice the usage of composition and polymorphism as well as inner classes and inheritance.

Getting ready

Interface, in this case, is a reference type that defines the signatures of the methods one can expect to see in the class that implements the interface. It is the public face of the functionality accessible to a client and is thus often called an **Application Program Interface** (**API**). It supports polymorphism and composition and thus facilitates even more flexible and extensible design.

An interface is implicitly abstract, which means it cannot be instantiated (no object can be created based on an interface only). It is used to contain abstract methods (without body) only. Now, since Java 8, it is possible to add default and private methods to an interface--the capability we are going to discuss in the following recipes.

Each interface can extend multiple other interfaces and, similar to class inheritance, inherit all the methods of the extended interfaces.

How to do it...

1. Create interfaces that will describe the API:

```
public interface SpeedModel {
  double getSpeedMph(double timeSec, int weightPounds,
                     int horsePower);
}
public interface Vehicle {
  void setSpeedModel(SpeedModel speedModel);
  double getSpeedMph(double timeSec);
}
public interface Car extends Vehicle {
  int getPassengersCount();
}
public interface Truck extends Vehicle {
  int getPayloadPounds();
}
```

2. Use factories, which are classes that generate objects that implement certain interfaces. A factory is an implementation of a pattern for creating objects without having to specify the exact class of the object that is created--specifying an interface only, rather than calling a constructor. It is especially helpful when an instance of object creation requires a complex process and/or significant code duplication.
   In our case, it makes sense to have the `FactoryVehicle` class that will create objects for the `Vehicle`, `Car`, and `Truck` interfaces and the `FactorySpeedModel` class that will generate objects for the `SpeedModel` interface. Such an API will allow you to write the following code:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int horsePower = 246;
  int vehicleWeight = 4000;
  Properties drivingConditions = new Properties();
  drivingConditions.put("roadCondition", "Wet");
  drivingConditions.put("tireCondition", "New");
   SpeedModel speedModel =  FactorySpeedModel
        .generateSpeedModel(drivingConditions);
   Car car = FactoryVehicle.buildCar(4, vehicleWeight,
                                     horsePower);
   car.setSpeedModel(speedModel);
   System.out.println("Car speed (" + timeSec + " sec) = " +
```

```
                                car.getSpeedMph(timeSec) + " mph");
        }
```

3. Observe that the code behavior is the same:

```
Car speed (10.0 sec) = 100.0 mph
```

However, the design is much more extensible.

How it works...

We have already seen one possible implementation of the `SpeedModel` class. Here is another way to do this inside the `FactorySpeedModel` class:

```
public class FactorySpeedModel {
  public static SpeedModel generateSpeedModel(
  Properties drivingConditions){
    //if drivingConditions includes "roadCondition"="Wet"
    return new SpeedModelWet(...);
    //if drivingConditions includes "roadCondition"="Dry"
    return new SpeedModelDry(...);
  }
  private class SpeedModelWet implements SpeedModel{
    public double getSpeedMph(double timeSec, int weightPounds,
                              int horsePower){...}
  }
  private class SpeedModelDry implements SpeedModel{
    public double getSpeedMph(double timeSec, int weightPounds,
                              int horsePower){...}
   }
}
```

We put comments (as pseudo code) and the … symbol instead of the code for brevity.

As you can see, the factory class may hide many different private and static nested classes, each containing a specialized model for particular driving conditions. Each model brings different results.

An implementation of the `FactoryVehicle` class may look like this:

```
public class FactoryVehicle {
  public static Car buildCar(int passengersCount, int weightPounds,
                             int horsePower){
    return new CarImpl(passengersCount, weightPounds, horsePower);
  }
  public static Truck buildTruck(int payloadPounds, int weightPounds,
                                 int horsePower){
    return new TruckImpl(payloadPounds, weightPounds, horsePower);
  }
  class CarImpl extends VehicleImpl implements Car {
    private int passengersCount;
    private CarImpl(int passengersCount, int weightPounds,
                    int horsePower){
      super(weightPounds + passengersCount * 250, horsePower);
      this.passengersCount = passengersCount;
```

```
    }
    public int getPassengersCount() {
      return this.passengersCount;
    }
  }
  class TruckImpl extends VehicleImpl implements Truck {
    private int payloadPounds;
    private TruckImpl(int payloadPounds, int weightPounds,
                      int horsePower){
      super(weightPounds+payloadPounds, horsePower);
      this.payloadPounds = payloadPounds;
    }
    public int getPayloadPounds(){ return payloadPounds; }
  }
  abstract class VehicleImpl implements Vehicle {
    private SpeedModel speedModel;
    private int weightPounds, horsePower;
    private VehicleImpl(int weightPounds, int horsePower){
      this.weightPounds = weightPounds;
      this.horsePower = horsePower;
    }
    public void setSpeedModel(SpeedModel speedModel){
      this.speedModel = speedModel;
    }
    public double getSpeedMph(double timeSec){
      return this.speedModel.getSpeedMph(timeSec, weightPounds,
                                         horsePower);
    }
  }
}
```

As you can see, an interface describes how to invoke object behavior; it also allows you to generate different implementations for different requests (and provided values) without changing the code of the main application.

There's more...

Let's try to model a crew cab--a truck with multiple passenger seats that combines the properties of a car and a truck. Java does not allow multiple inheritances. This is another case where interfaces come to the rescue.

The `CrewCab` class may look like this:

```
class CrewCab extends VehicleImpl implements Car, Truck {
  private int payloadPounds;
  private int passengersCount;
  private CrewCabImpl(int passengersCount, int payloadPounds,
                      int weightPounds, int horsePower) {
    super(weightPounds + payloadPounds
          + passengersCount * 250, horsePower);
    this.payloadPounds = payloadPounds;
    this. passengersCount = passengersCount;
  }
  public int getPayloadPounds(){ return payloadPounds; }
  public int getPassengersCount() {
    return this.passengersCount;
  }
}
```

This class implements both the interfaces--`Car` and `Truck`--and passes the combined weight of the vehicle, payload, and passengers with their luggage to the base class constructor.

We can also add the following method to `FactoryVehicle`:

```
public static Vehicle buildCrewCab(int passengersCount,
           int payload, int weightPounds, int horsePower){
  return new CrewCabImpl(passengersCount, payload,
                         weightPounds, horsePower);
}
```

The double nature of the `CrewCab` object can be demonstrated in the following test:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int horsePower = 246;
  int vehicleWeight = 4000;
  Properties drivingConditions = new Properties();
  drivingConditions.put("roadCondition", "Wet");
```

```
  drivingConditions.put("tireCondition", "New");
  SpeedModel speedModel = FactorySpeedModel
                    .generateSpeedModel(drivingConditions);
  Vehicle vehicle = FactoryVehicle
         .buildCrewCab(4, 3300, vehicleWeight, horsePower);
  vehicle.setSpeedModel(speedModel);
  System.out.println("Payload = " +
                     ((Truck)vehicle).getPayloadPounds()) + " pounds");
  System.out.println("Passengers count = " +
                     ((Car)vehicle).getPassengersCount());
  System.out.println("Crew cab speed (" + timeSec + " sec) = "  +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

As you can see, we can cast the object of the `CrewCub` class to each of the interfaces it implements. If we run this program, the results will be as follows:

```
Payload = 3300 pounds
Passengers count = 4
Crew cab speed (10.0 sec) = 76.0 mph
```

See also

Refer to the following recipes in this chapter:

- Creating interfaces with default and static methods
- Creating interfaces with private methods

Creating interfaces with default and static methods

In this recipe, you will learn about two new features that were first introduced in Java 8: default and static methods in an interface.

Getting ready

The default method allows you to add new functionality to an interface without changing the classes that have implemented this interface. The method is called *default* because it provides functionality in case a method is not implemented by the class. If, however, the class implements it, the interface's default implementation is ignored and overridden by the class implementation.

Having a static method in an interface can provide functionality the same way a static method in a class can. As with a class static method (which can be called without class instantiation), an interface static method can also be called by adding the name of the interface in front of it.

A static interface method cannot be overridden by any class, including the class that implements this interface, and it cannot hide any static method of any class, including the class that implements this interface.

So far, we have created an amazing piece of software that calculates the speed of a vehicle. If this program becomes popular (as it should), it can be used by readers who prefer a metric system of weight units. To address such a need later--after our speed-calculating software has become popular--we have decided to add more methods to the `Truck` interface; however, we do not want to break the existing implementation of `FactoryVehicle`, created by some other company.

The default interface methods were introduced for exactly such a situation. Using them, we can release a new version of an interface without the need to coordinate it with the development of `FactoryVehicle`.

How to do it...

1. Enhance the `Truck` interface by adding the `getPayloadKg()` method, which returns the truck payload in kilograms. You can do this without forcing a change in the `TruckImpl` class that implements the `Truck` interface inside `FactoryVehicle`--by adding a new default method to the `Truck` interface:

```
public interface Truck extends Vehicle {
  int getPayloadPounds();
  default int getPayloadKg(){
    return (int) Math.round(0.454 * getPayloadPounds());
  }
}
```

   Notice how the new method `getPayloadKg()` uses the existing `getPayloadPounds()` method as if the latter is implemented inside the interface too when, in fact, it is implemented by a class inside `FactoryVehicle`. The magic happens during runtime when this method becomes dynamically bound to the instance of the class that implements this interface.

   We could not make the `getPayloadKg()` method static because it would not be able to access the non-static `getPayloadPounds()` method, and we must use the `default` keyword because only the default or static method of an interface can have a body.

2. Write the demo code that uses the new method:

```
public static void main(String... arg) {
  Truck truck = FactoryVehicle.buildTruck(3300, 4000, 246);
  System.out.println("Payload in pounds: " +
                     truck.getPayloadPounds());
  System.out.println("Payload in kg: " + truck.getPayloadKg());
}
```

3. Run the preceding program and see the output:

```
Payload in pounds: 3300
Payload in kg: 1498
```

4. Notice that the new method works even without changing the class that implemented it.
5. Later, when you decide to improve the implementation of the `FactoryVehicle` class, you can do it by adding the corresponding method, for example:

```
class TruckImpl extends VehicleImpl implements Truck {
  private int payloadPounds;
  private TruckImpl(int payloadPounds, int weightPounds,
                    int horsePower) {
    super(weightPounds + payloadPounds, horsePower);
    this.payloadPounds = payloadPounds;
  }
  public int getPayloadPounds(){ return payloadPounds; }
  public int getPayloadKg(){ return -2; }
}
```

   We made the `return -2` implementation in order to make it obvious which implementation is used.

6. Run the same demo program. The results will be as follows:

```
Payload in pounds: 3300
Payload in kg: -2
```

   As you can see, the method in the `TruckImpl` class has overridden the default implementation in the `Truck` interface.

7. Enhance the `Truck` interface with the ability to enter the payload in kilograms without changing the implementation of `FactoryVehicle`. But we would also like the `Truck` implementation to remain immutable, and we do not want to add a setter method. With all these limitations, our only recourse is to add `convertKgToPounds()` to the `Truck` interface, and it has to be `static` since we are going to use it before the object that implements the `Truck` interface is constructed:

```
public interface Truck extends Vehicle {
  int getPayloadPounds();
  default int getPayloadKg(){
    return (int) Math.round(0.454 * getPayloadPounds());
  }
  static int convertKgToPounds(int kgs){
    return (int) Math.round(2.205 * kgs);
```

```
            }
        }
```

How it works...

Fans who love using the metric system of units can now take advantage of the new method:

```
public static void main(String... arg) {
  int horsePower = 246;
  int payload = Truck.convertKgToPounds(1500);
  int vehicleWeight = Truck.convertKgToPounds(1800);
  Truck truck = FactoryVehicle
           .buildTruck(payload, vehicleWeight, horsePower);
  System.out.println("Payload in pounds: " + truck.getPayloadPounds());
  int kg = truck.getPayloadKg();
  System.out.println("Payload converted to kg: " + kg);
  System.out.println("Payload converted back to pounds: " +
                     Truck.convertKgToPounds(kg));
}
```

The results will be as follows:

```
Payload in pounds: 3308
Payload converted to kg: 1502
Payload converted back to pounds: 3312
```

The value 1,502 is close to the original 1,500, while 3,308 is close to 3,312. The difference is caused by the error of approximation during the conversion.

See also

Refer to the following recipe in this chapter:

- Creating interfaces with private methods

Creating interfaces with private methods

In this recipe, you will learn about a new feature introduced in Java 9: private interface method, which is of two types, namely static and non-static.

Getting ready

Private methods in an interface are new in Java 9. They allow you to make interface methods (with a body) accessible only to other methods (with a body) in the same interface.

A private method in an interface cannot be overridden anywhere--not by a method of any interface, nor by a method in any class. Its only purpose is to contain functionality that is common between two or more methods with a body, either private or public, in the same interface. It can also be used by one method only in order to make the code easier to understand.

A private interface method must have an implementation. A private interface method not used by other methods of the same interface does not make sense.

A static private interface method can be accessed by non-static and static methods of the same interface. The non-static private interface method can be accessed only by static methods of the same interface.

How to do it...

1. Add the `getWeightKg()` implementation too. Since we do not have the `getWeightPounds()` method in the interface, the method signature should include an input parameter:

```
public interface Truck extends Vehicle {
  int getPayloadPounds();
  default int getPayloadKg(){
    return (int) Math.round(0.454 * getPayloadPounds());
  }
  static int convertKgToPounds(int kilograms){
    return (int) Math.round(2.205 * kilograms);
  }
  default int getWeightKg(int pounds){
    return (int) Math.round(0.454 * pounds);
  }
}
```

2. Remove the redundant code using the private interface method:

```
public interface Truck extends Vehicle {
  int getPayloadPounds();
  default int getPayloadKg(int pounds){
    return convertPoundsToKg(pounds);
  }
  static int convertKgToPounds(int kilograms){
    return (int) Math.round(2.205 * kilograms);
  }
  default int getWeightKg(int pounds){
    return convertPoundsToKg(pounds);
  }
  private int convertPoundsToKg(int pounds){
    return (int) Math.round(0.454 * pounds);
  }
}
```

How it works...

The following code demonstrates the new addition:

```
public static void main(String... arg) {
  int horsePower = 246;
  int payload = Truck.convertKgToPounds(1500);
  int vehicleWeight = Truck.convertKgToPounds(1800);
  Truck truck = FactoryVehicle
            .buildTruck(payload, vehicleWeight, horsePower);
  System.out.println("Weight in pounds: " + vehicleWeight);
  int kg = truck.getWeightKg(vehicleWeight);
  System.out.println("Weight converted to kg: " + kg);
  System.out.println("Weight converted back to pounds: " +
                     Truck.convertKgToPounds(kg));
}
```

The results of the test run do not change:

```
Weight in pounds: 3969
Weight converted to kg: 1802
Weight converted back to pounds: 3973
```

There's more...

Well, with the `getWeightKg(int pounds)` method accepting the input parameter, the method name can be misleading because (by contrast, with `getPayloadKg()`) the method does not control the source of the data. The input value can represent anything. After realizing it, we decided that the interface would be better without it, but only after making the `convertPoundsToKg()` method public. Since it does not require access to the object elements, it can be static too:

```
public interface Truck extends Vehicle {
  int getPayloadPounds();
  default int getPayloadKg(int pounds){
    return convertPoundsToKg(pounds);
  }
  static int convertKgToPounds(int kilograms){
    return (int) Math.round(2.205 * kilograms);
  }
  static int convertPoundsToKg(int pounds){
    return (int) Math.round(0.454 * pounds);
  }
}
```

Fans of the metric system will still be able to convert pounds into kilograms and back. Besides, since the converting methods are static, we do not need to create an instance of the class that implements the `Truck` interface:

```
public static void main(String... arg) {
  int payload = Truck.convertKgToPounds(1500);
  int vehicleWeight = Truck.convertKgToPounds(1800);
  System.out.println("Weight in pounds: " + vehicleWeight);
  int kg = Truck.convertPoundsToKg(vehicleWeight);
  System.out.println("Weight converted to kg: " + kg);
  System.out.println("Weight converted back to pounds: " +
                     Truck.convertKgToPounds(kg));
}
```

The results do not change:

```
Weight in pounds: 3969
Weight converted to kg: 1802
Weight converted back to pounds: 3973
```

See also

Refer to the following recipe of this chapter:

- Creating interfaces with default and static methods

Using enums to represent constant entities

In this recipe, you will learn about the special data type `enum` that allows you to create a custom type with predefined values.

Getting ready

Let's have a quick look at a couple of examples. Here is the simplest possible `enum` type:

```
public enum RoadCondition {
  DRY, WET, SNOW
}
```

Say we run the following loop:

```
for(RoadCondition v: RoadCondition.values()){
  System.out.println(v);
}
```

The results of this will be as follows:



The `enum` type implicitly extends `java.util.Enum` (so you cannot extend your custom type `RoadCondition`, for example) and automatically acquires its methods. In the preceding code, we have already seen one (the most useful) method, `values()`, that returns an array of the `enum` elements.

Another useful `enum` method is `valueOf(String)` that returns the constant of the specified `enum` type with the specified name (passed in as `String`).

The methods of each element include useful ones such as `equals()`, `name()`, and `ordinal()`. Now say we run the following loop:

```
for(RoadCondition v: RoadCondition.values()){
  System.out.println(v.ordinal());
  System.out.println(v.name());
  System.out.println(RoadCondition.SNOW.equals(v));
}
```

We will get the following result:



As you may have guessed, `ordinal()` returns the position of the value in the `enum` declaration, starting with zero.

Other `enum` features include the following:

- The `enum` class body can include a constructor, methods, variables, and constants
- The constructor is called automatically when `enum` is accessed the first time
- The static constant can be accessed via the class name
- The non-static constant can be accessed via any of the elements



Java comes with several predefined `enum` types, including `DayOfWeek` and `Month`.

To make it look more like a real-life `enum`, let's use traction coefficient as an element value. Such a coefficient reflects the degree of traction between the road and the tires. The higher the value, the better the traction. This value can be used for the improvement of a vehicle speed calculation. For example, `SpeedModel` should be able to receive `RoadCondition.WET` and extract the traction coefficient `0.2` from it.

How to do it...

1. Define the `RoadCondition` enumeration in the `SpeedModel` interface (because it is part of the `SpeedModel` API):

```
public interface SpeedModel {
  double getSpeedMph(double timeSec, int weightPounds,
                     int horsePower);

  enum RoadCondition {
    DRY(1.0), WET(0.2), SNOW(0.04);
    private double traction;
    RoadCondition(double traction){ this.traction = traction;}
    public double getTraction(){ return this.traction; }
  }
}
```

   There is no need to add the `public` keyword in front of `enum RoadCondition` because, in an interface, it defaults to `public` (if the `private` keyword is not used instead).

2. Run the following loop:

```
for(RoadCondition v: RoadCondition.values()){
  System.out.println(v + " => " + v.getTraction());
}
```

3. You'll get the following result:

```
DRY => 1.0
WET => 0.2
SNOW => 0.04
```

4. Add `enum TireCondition` to the `iSpeedModel` interface too:

```
enum TireCondition {
  NEW(1.0), WORN(0.2);
  private double traction;
  TireCondition(double traction){ this.traction = traction; }
  public double getTraction(){return this.traction;}
}
```

5. Add `enum` for `DrivingCondition`:

```
enum DrivingCondition {
  ROAD_CONDITION, TIRE_CONDITION
}
```

How it works...

With the three `enum` types in place, we can calculate the vehicle speed of all the possible combinations of the given driving conditions:

```
public static void main(String... arg) {
  double timeSec = 10.0;
  String[] roadConditions = { RoadCondition.WET.toString(),
                              RoadCondition.SNOW.toString() };
  String[] tireConditions = { TireCondition.NEW.toString(),
                              TireCondition.WORN.toString() };
  for(String rc: roadConditions){
    for(String tc: tireConditions){
      Properties drivingCond = new Properties();
      drivingCond.put(DrivingCondition
                      .ROAD_CONDITION.toString(), rc);
      drivingCond.put(DrivingCondition
                      .TIRE_CONDITION.toString(), tc);
      SpeedModel speedModel = FactorySpeedModel
                      .generateSpeedModel(drivingCond);
      Car car = FactoryVehicle.buildCar(4, 4000, 246);
      car.setSpeedModel(speedModel);
      System.out.println("Car speed (" + timeSec + " sec) = " +
                         car.getSpeedMph(timeSec) + " mph");
      }
   }
}
```

The results will be as follows:

```
Road condition is WET => traction=0.2
Tire condition is NEW => traction=1.0
Car speed (10.0 sec) = 52.0 mph
Road condition is WET => traction=0.2
Tire condition is WORN => traction=0.2
Car speed (10.0 sec) = 10.0 mph
Road condition is SNOW => traction=0.04
Tire condition is NEW => traction=1.0
Car speed (10.0 sec) = 10.0 mph
Road condition is SNOW => traction=0.04
Tire condition is WORN => traction=0.2
Car speed (10.0 sec) = 2.0 mph
```

That is how `enum` allows you to define an API input and make input data validation unnecessary.

There's more...

The higher the temperature, the quicker the road dries out, and the traction coefficient is getting higher. To account for this, we can add a `temperature` property to `enum RoadCondition` and override the `getTraction()` method for the `WET` element, for example:

```
enum RoadCondition {
  public int temperature;
  DRY(1.0),
  WET(0.2) {
    public double getTraction(){
      RoadCondition return temperature > 60 ? 0.4 : 0.2
    }
  },
  SNOW(0.04);
  private double traction;
  RoadCondition(double traction){this.traction = traction; }
  public double getTraction(){ return this.traction; }
}
```

Now we can set the `temperature` property on `RoadCondition` before the speed calculation and get a different value of speed for the wet road condition. Add this line before calling the speed calculations:

```
RoadCondition.temperature = 63;
```

If you do this, the results will be as follows:

```
Road condition is WET => traction=0.4
Tire condition is NEW => traction=1.0
Car speed (10.0 sec) = 103.0 mph
Road condition is WET => traction=0.4
Tire condition is WORN => traction=0.2
Car speed (10.0 sec) = 21.0 mph
Road condition is SNOW => traction=0.04
Tire condition is NEW => traction=1.0
Car speed (10.0 sec) = 10.0 mph
Road condition is SNOW => traction=0.04
Tire condition is WORN => traction=0.2
Car speed (10.0 sec) = 2.0 mph
```

Using the @Deprecated annotation to deprecate APIs

In this recipe, you will learn about the deprecation of API elements and the enhancements of the `@Deprecated` annotation in Java 9.

Getting ready

The `@Deprecated` annotation was first introduced in Java 5, while the Javadoc tag `@deprecated` was introduced in Java even earlier. The presence of the annotation forces the compiler to generate a warning that can be suppressed by the annotation:

```
@SuppressWarnings("deprecation")
```

Since Java 9, the annotation can have one or both the methods, namely `since()` and `forRemoval()`:

```
@Deprecated(since = "2.1", forRemoval = true)
```

The `since()` method allows the setting of the API version (as `String`). It depicts the API version from where the particular class or method was deprecated. If not specified, the default value of the `since()` method is `""` (empty `String`).

The `forRemoval()` method depicts the intent to remove the marked element (if `true`) or not (if `false`). If not set, the default value is `false`. If the `forRemoval()` method is present with the value `true` to suppress the warning, one needs to specify the following:

```
@SuppressWarnings("removal")
```

How to do it...

1. Look at the `Car` interface again:

```
public interface Truck extends Vehicle {
  int getPayloadPounds();
}
```

2. Instead of the `getPayloadKg()` method, introduce a more general method and `enum` for supporting the metric system of weight units:

```
int getPayload(WeigthUnit weightUnit);
enum WeigthUnit { Pound, Kilogram }
```

   Such an enhancement allows you to have more flexibility in future.

3. Deprecate the `getPayloadPounds()` method and add a Javadoc with explanations:

```
/**
* Returns the payload of the truck.
*
* @return the payload of the truck in the specified units
* of weight measurements
* @deprecated As of API 2.1, to avoid adding methods
* for each possible weight unit,
* use {@link #getPayload(WeigthUnit weightUnit)} instead.
*/
@Deprecated(since = "2.1", forRemoval = true)
int getPayloadPounds();
```

   Each of the methods in the `@Deprecated` annotation is optional.

How it works...

If we compile the preceding code, there will be a warning *this method has been deprecated and marked for removal* at every place of its implementation.

If the `forRemoval()` method is not present or set to `false`, the warning message would say *has been deprecated* only.

To avoid the warning, we need to either avoid using the deprecated method or suppress the warning as described earlier.

There's more...

In Java 9, both--the Javadoc tag `@deprecated` and `@Deprecated` annotation--are required. The presence of only one of them is considered an error.

Using HTML5 in Javadocs

In this recipe, you will learn about the usage of HTML5 tags in the Javadoc comments in Java 9.

Getting ready

There are plenty of sources on the Internet that describe HTML5 tags. Since Java 9, one can use any one of them in Javadoc comments.

HTML5 provides better browser compatibility. It is also more mobile-friendly than its predecessor, HTML4. But to take advantage of HTML5, one has to specify the `-html5` parameter during Javadoc generation. Otherwise, only HTM4-style comments will continue to be supported.

How to do it...

Here is an example of the HTML5 tags used for Javadoc comments:

```
/**
<h2>Returns the weight of the car.</h2>
<article>
  <h3>If life would be often that easy</h3>
  <p>
    Do you include unit of measurement into the method name or not?
  </p>
  <p>
    The new signature demonstrates extensible design of an interface.
  </p>
</article>
<aside>
  <p> A few other examples could be found
    <a href="http://www.nicksamoylov.com/cat/programming/">here</a>.
  </p>
</aside>
* @param weightUnit - an element of the enum Car.WeightUnit
* @return the weight of the car in the specified units of weight
*/
int getMaxWeight(WeigthUnit weightUnit);
```

If you are using IntelliJ IDEA, go to Tools | Generate JavaDoc... and set the `-html5` value in the Other command line arguments field and click on OK. Without an IDE, use the following command:

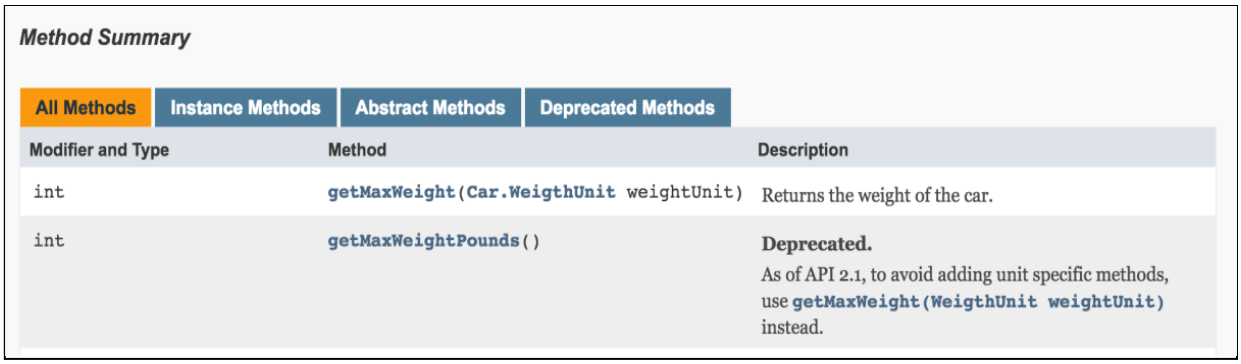
```
javadoc [options] [packagenames] [sourcefiles] [@files]
```

Consider the following example:

```
javadoc -html5 -sourcepath src -d api com.cookbook.oop
```

Here, `src` is the folder that contains the source code in the `com` subfolder, and `api` is a folder in the current directory where Javadoc is going to be stored. The `com.cookbook.oop` is the package for which you want to generate Javadoc.

The resulting Javadoc will look like this:

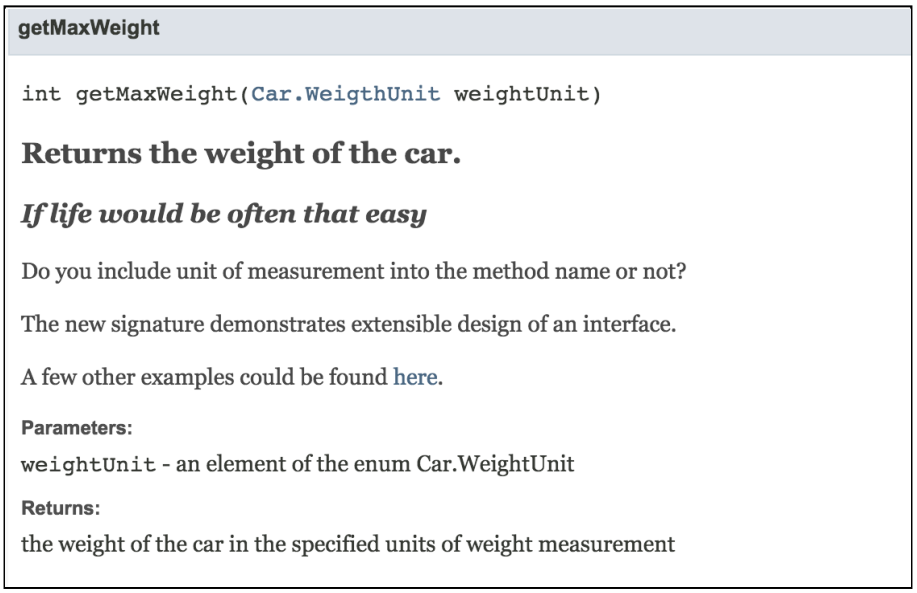


Method Summary

All Methods | Instance Methods | Abstract Methods | Deprecated Methods

Modifier and Type	Method	Description
int	`getMaxWeight(Car.WeigthUnit weightUnit)`	Returns the weight of the car.
int	`getMaxWeightPounds()`	**Deprecated.** As of API 2.1, to avoid adding unit specific methods, use `getMaxWeight(WeigthUnit weightUnit)` instead.

The preceding description is taken from the text inside tags `<h2>`, but its font is not different from other lines. Here's how the full description will look:



getMaxWeight

```
int getMaxWeight(Car.WeigthUnit weightUnit)
```

Returns the weight of the car.

If life would be often that easy

Do you include unit of measurement into the method name or not?

The new signature demonstrates extensible design of an interface.

A few other examples could be found here.

Parameters:

`weightUnit` - an element of the enum Car.WeightUnit

Returns:

the weight of the car in the specified units of weight measurement

You can see how prominently the text in tags `<h2>` and `<h3>` is presented and the `here` link is highlighted and can be clicked.

Modular Programming

In this chapter, we will cover the following recipes:

- Using jdeps to find dependencies in a Java application
- Creating a simple modular application
- Creating a modular JAR
- Using a module JAR with pre-JDK 9 applications
- Bottom-up migration
- Top-down migration
- Using services to create loose coupling between consumer and provider modules
- Creating a custom modular runtime image using jlink
- Compiling for older platform versions
- Creating multirelease JARs
- Using Maven to develop a modular application

Introduction

Modular programming enables one to organize code into independent, cohesive modules, which can be combined together to achieve the desired functionality. This allows in creating code that is:

- More cohesive because the modules are built with a specific purpose, so the code that resides there tends to cater to that specific purpose.
- Encapsulated because modules can interact with only those APIs that have been made available by the other modules.
- Reliable because the discoverability is based on the modules and not on the individual types. This means that if a module is not present, then the dependent module cannot be executed until it is discoverable by the dependent module. This helps in preventing runtime errors.
- Loosely coupled. If you use service interfaces, then the module interface and the service interface implementation can be loosely coupled.

So, the thought process in designing and organizing the code will now involve identifying the modules, code, and configuration files, which go into the module and the packages in which the code is organized within the module. After that, we have to decide the public APIs of the module, thereby making them available for use by dependent modules.

Coming to the development of the Java Platform Module System, it is being governed by **Java Specification Request** (**JSR**) 376 (https://www.jcp.org/en/jsr/detail?id=376). The JSR mentions the need for a module system is to address the following fundamental issues:

- **Reliable configuration**: Developers have long suffered from the brittle, error-prone classpath mechanism for configuring program components. The classpath cannot express relationships between components, so if a necessary component is missing, then that will not be discovered until an attempt is made to use it. The classpath also

allows classes in the same package to be loaded from different components, leading to unpredictable behavior and difficult-to-diagnose errors. The proposed specification will allow a component to declare that it depends upon other components as other components depend upon it.
- **Strong encapsulation**: The access-control mechanism of the Java programming language and the JVM provides no way for a component to prevent other components from accessing its internal packages. The proposed specification will allow a component to declare its packages that are accessible by other components and those that are not.

The JSR further goes to list the advantages that result from addressing the preceding issues, as follows:

- **A scalable platform**: The ever-increasing size of the Java SE platform has made it increasingly difficult to use in small devices, despite the fact that many such devices are capable of running an SE-class JVM. The compact profiles introduced in Java SE 8 (JSR 337) help in this regard, but they are not nearly flexible enough. The proposed specification will allow the Java SE platform and its implementations to be decomposed into a set of components that can be assembled by developers into custom configurations that contain only the functionality actually required by an application.
- **Greater platform integrity**: Casual use of APIs that are internal to Java SE platform implementations is both a security risk and a maintenance burden. The strong encapsulation provided by the proposed specification will allow components that implement the Java SE platform to prevent access to their internal APIs.
- **Improved performance**: Many ahead-of-time, whole-program optimization techniques can be more effective when it is known that a class can refer only to classes in a few other specific components rather than to any class loaded at runtime. Performance is especially enhanced when the components of an application can be optimized in conjunction with the components that implement the Java SE platform.

In this chapter, we will look at few important recipes that will help you get started with modular programming.

Using jdeps to find dependencies in a Java application

The first step in modularizing your application is to identify its dependencies. A static analysis tool called `jdeps` was introduced in JDK 8 to enable developers to find the dependencies of their applications. There are multiple options supported in the command, which enables developers to check for dependencies to the JDK internal APIs, show the dependencies at the package level, show the dependencies at the class level, and filter the dependencies, among other options.

In this recipe, we will look at how to make use of the `jdeps` tool by exploring its functionality and using the multiple command-line options it supports.

Getting ready

We need a sample application, which we can run against the `jdeps` command to find its dependencies. So, we thought of creating a very simple application that uses the Jackson API to consume JSON from the REST API: `http://jsonplaceholder.typicode.com/users`.

In the sample code, we also added a call to the deprecated JDK internal API, called `sun.reflect.Reflection.getCallerClass()`. This way, we can see how `jdeps` helps in finding dependencies to the JDK internal APIs.

The following steps will help you to set up the prerequisites for this recipe:

1. You can get the complete code for the sample from the location, `chp3/1_json-jackson-sample`. We have built this code against Java 9 and it compiles well. So, you need not install something other than Java 9 to compile it.
2. Once you have the code, compile it by using the following:

```
#On Linux
javac -cp 'lib/*' -d classes -sourcepath src $(find src -name *.java)

#On Windows
javac -cp lib\*;classes -d classes src\com\packt\model\*.java
src\com\packt\*.java
```

   You will see a warning for the use of an internal API, which you can safely ignore. We added this with a purpose to demonstrate the capability of `jdeps`. Now, you should have your compiled class files in the classes directory.

3. You can run the sample program by using the following:

```
#On Linux:
java -cp lib/*:classes/ com.packt.Sample
#On Windows:
java -cp lib\*;classes com.packt.Sample
```

4. We have provided the `run.bat` and `run.sh` scripts at `chp3/1_json-jackson-sample`. You can compile and run using these scripts as well.
5. Let's also create a JAR file for this sample so that we can run `jdeps` on the JAR file as well:

```
jar cvfm sample.jar manifest.mf -C classes .
```

A `sample.jar` file gets created in the current directory. You can also run the JAR by issuing this command: `java -jar sample.jar`.

How to do it...

1. The simplest way to use `jdeps` is as follows:

```
jdeps -cp classes/:lib/* classes/com/packt/Sample.class
```

The preceding command is equivalent to the following:

```
jdeps -verbose:package -cp classes/:lib/*
classes/com/packt/Sample.class
```

The output for the preceding code is as follows:

```
Sample.class -> classes
Sample.class -> lib/jackson-core-2.8.4.jar
Sample.class -> lib/jackson-databind-2.8.4.jar
Sample.class -> java.base
   com.packt (Sample.class)
      -> com.fasterxml.jackson.core.type                 jackson-core-2.8.4.jar
      -> com.fasterxml.jackson.databind                  jackson-databind-2.8.4.jar
      -> com.packt.model                                 classes
      -> java.io
      -> java.lang
      -> java.lang.invoke
      -> java.net
      -> java.util
      -> sun.reflect                                     JDK internal API (java.base)
```

In the preceding command, we use `jdeps` to list the dependencies for the class file, `Sample.class`, at the package level. We have to provide `jdeps` with the path to search for the dependencies of the code being analyzed. This can be done by setting the `-classpath`, `-cp`, or `--class-path` option of the `jdeps` command.



The `-verbose:package` option lists the dependencies at the package level.

2. Let's list the dependencies at the class level:

```
jdeps -verbose:class -cp classes/:lib/*
classes/com/packt/Sample.class
```

The output of the preceding command is as follows:

```
Sample.class -> classes
Sample.class -> lib/jackson-core-2.8.4.jar
Sample.class -> lib/jackson-databind-2.8.4.jar
Sample.class -> java.base
   com.packt.Sample (Sample.class)
      -> com.fasterxml.jackson.core.type.TypeReference      jackson-core-2.8.4.jar
      -> com.fasterxml.jackson.databind.ObjectMapper        jackson-databind-2.8.4.jar
      -> com.packt.model.Company                            classes
      -> com.packt.model.User                               classes
      -> java.io.PrintStream
      -> java.lang.Class
      -> java.lang.Exception
      -> java.lang.Object
      -> java.lang.String
      -> java.lang.System
      -> java.lang.invoke.CallSite
      -> java.lang.invoke.MethodHandles
      -> java.lang.invoke.MethodHandles$Lookup
      -> java.lang.invoke.MethodType
      -> java.lang.invoke.StringConcatFactory
      -> java.net.URL
      -> java.util.Iterator
      -> java.util.List
      -> sun.reflect.Reflection                             JDK internal API (java.base)
```

In this case, we make use of the `-verbose:class` option to list the dependencies at the class level, which is why you can see that the `com.packt.Sample` class depends on `com.packt.model.Company`, `java.lang.Exception`, `com.fasterxml.jackson.core.type.TypeReference`, and so on.

3. Let's get the summary of the dependencies:

```
jdeps -summary -cp classes/:lib/* classes/com/packt/Sample.class
```

The output is as follows:

```
Sample.class -> classes
Sample.class -> lib/jackson-core-2.8.4.jar
Sample.class -> lib/jackson-databind-2.8.4.jar
Sample.class -> java.base
```

4. Let's check for the dependency on the JDK internal API:

```
jdeps -jdkinternals -cp classes/:lib/*
classes/com/packt/Sample.class
```

The following is the output of the preceding command:

```
Sample.class -> jdk.unsupported
   com.packt.Sample                          -> sun.reflect.Reflection                      JDK internal API (jdk.unsupported)
classes -> jdk.unsupported
   com.packt.Sample                          -> sun.reflect.Reflection                      JDK internal API (jdk.unsupported)

Warning: JDK internal APIs are unsupported and private to JDK implementation that are
subject to be removed or changed incompatibly and could break your application.
Please modify your code to eliminate dependence on any JDK internal APIs.
For the most recent update on JDK internal API replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

JDK Internal API                        Suggested Replacement
----------------                        ---------------------
sun.reflect.Reflection                  Use java.lang.StackWalker @since 9
```



The StackWalker API is the new API for traversing the call stack, introduced in Java 9. This is the replacement for the `sun.reflect.Reflection.getCallerClass()` *method. We will discuss this API in* `Chapter 12`*, Memory Management and Debugging.*

5. Let's run the `jdeps` on the JAR file, `sample.jar`:

```
jdeps -s -cp lib/* sample.jar
```

The output we get is the following:

```
jackson-core-2.8.4.jar -> java.base
jackson-databind-2.8.4.jar -> lib/jackson-annotations-2.8.4.jar
jackson-databind-2.8.4.jar -> lib/jackson-core-2.8.4.jar
jackson-databind-2.8.4.jar -> java.base
jackson-databind-2.8.4.jar -> java.desktop
jackson-databind-2.8.4.jar -> java.logging
jackson-databind-2.8.4.jar -> java.sql
jackson-databind-2.8.4.jar -> java.xml
sample.jar -> lib/jackson-core-2.8.4.jar
sample.jar -> lib/jackson-databind-2.8.4.jar
sample.jar -> java.base
sample.jar -> jdk.unsupported
```

The preceding information obtained after investigating the `sample.jar` using `jdeps` is quite useful. It clearly states the dependencies of our JAR files and is very useful when we try to migrate this application to a modular application.

6. Let's find if there are any dependencies to a given package name:

```
jdeps -p java.util sample.jar
```

The output is as follows:

```
sample.jar -> java.base
   com.packt                                -> java.util                                  java.base
```

The `-p` option is used to find dependencies on the given package name. So, we get to know that our code depends on the `java.util` package. Let's try with some other package name:

```
jdeps -p java.util.concurrent sample.jar
```

There is no output, which means that our code doesn't depend on the `java.util.concurrent` package.

7. We would want to run the dependency check only for our code. Yes, this is possible. Suppose we run `jdeps -cp lib/* sample.jar`, you will see even the library JARs being analyzed. We wouldn't want that, right? Let's just include the classes of the `com.packt` package:

```
jdeps -include 'com.packt.*' -cp lib/* sample.jar
```

The output is as follows:

```
sample.jar -> lib/jackson-core-2.8.4.jar
sample.jar -> lib/jackson-databind-2.8.4.jar
sample.jar -> java.base
sample.jar -> jdk.unsupported
   com.packt                                -> com.fasterxml.jackson.core.type            jackson-core-2.8.4.jar
   com.packt                                -> com.fasterxml.jackson.databind             jackson-databind-2.8.4.jar
   com.packt                                -> com.packt.model                            sample.jar
   com.packt                                -> java.io                                    java.base
   com.packt                                -> java.lang                                  java.base
   com.packt                                -> java.lang.invoke                           java.base
   com.packt                                -> java.net                                   java.base
   com.packt                                -> java.util                                  java.base
   com.packt                                -> sun.reflect                                JDK internal API (jdk.unsupported)
   com.packt.model                          -> java.lang                                  java.base
```

8. Let's check whether our code is dependent on a specific package:

```
jdeps -p 'com.packt.model' sample.jar
```

The output is as follows:

```
sample.jar -> sample.jar
   com.packt                                -> com.packt.model                            sample.jar
```

9. We can use `jdeps` on analyzing the JDK modules. Let's pick the `java.httpclient` module for analysis:

```
jdeps -m java.httpclient
```

Here is the output:

```
java.httpclient
 [jrt:/java.httpclient]
   requires java.base
java.httpclient -> java.base
   java.net.http                           -> java.io                                java.base
   java.net.http                           -> java.lang                              java.base
   java.net.http                           -> java.lang.invoke                       java.base
   java.net.http                           -> java.net                               java.base
   java.net.http                           -> java.nio                               java.base
   java.net.http                           -> java.nio.channels                      java.base
   java.net.http                           -> java.nio.charset                       java.base
   java.net.http                           -> java.nio.file                          java.base
   java.net.http                           -> java.security                          java.base
   java.net.http                           -> java.time                              java.base
   java.net.http                           -> java.time.temporal                     java.base
   java.net.http                           -> java.util                              java.base
   java.net.http                           -> java.util.concurrent                   java.base
   java.net.http                           -> java.util.concurrent.atomic            java.base
   java.net.http                           -> java.util.concurrent.locks             java.base
   java.net.http                           -> java.util.function                     java.base
   java.net.http                           -> java.util.stream                       java.base
   java.net.http                           -> javax.net.ssl                          java.base
   java.net.http                           -> sun.net                                java.base (qualified)
   java.net.http                           -> sun.net.httpclient.hpack               java.httpclient
   sun.net.httpclient.hpack                -> java.io                                java.base
   sun.net.httpclient.hpack                -> java.lang                              java.base
   sun.net.httpclient.hpack                -> java.lang.invoke                       java.base
   sun.net.httpclient.hpack                -> java.net                               java.base
   sun.net.httpclient.hpack                -> java.nio                               java.base
   sun.net.httpclient.hpack                -> java.util                              java.base
   sun.net.httpclient.hpack                -> java.util.function                     java.base
```

We can also find whether a given module is dependent on another module by using the `--require` option, as follows:

```
jdeps --require java.logging -m java.sql
```

Here is the output:

```
java.sql
 [jrt:/java.sql]
   requires mandated java.base
   requires public java.logging
   requires public java.xml
java.sql -> java.logging
   java.sql                                -> java.util.logging                      java.logging
   javax.sql                               -> java.util.logging                      java.logging
```

In the preceding command, we tried to find out whether the `java.sql` module is dependent on the `java.logging` module. The output we get is the dependency summary of the `java.sql` module and the packages in the `java.sql` module, which make use of the code from the `java.logging` module.

How it works...

The `jdeps` command is a static class dependency analyzer and is used to analyze the static dependencies of the application and its libraries.
The `jdeps` command by default shows the package-level dependencies of the input files, which can be `.class` files, a directory, or a JAR file. This is configurable and can be changed to show class-level dependencies. There are multiple options available to filter out the dependencies and to specify the class files to be analyzed. We have seen a regular use of the `-cp` command-line option. This option is used to provide the locations to search for the analyzed code's dependencies.

We have analyzed the class file, JAR files, and the JDK modules, and we also tried out different options of the `jdeps` command. There are a few options, such as `-e`, `-regex`, `--regex`, `-f`, `--filter`, and `-include`, which accept a regular expression (regex). It's important to understand the output of the `jdeps` command. There are two parts of information for every class/JAR file being analyzed:

1. The summary of the dependency for the analyzed file (JAR or class file). This consists of the name of the class or the JAR file on the left and the name of the dependent entity on the right. The dependent entity can be a directory, a JAR file, or a JDK module, as follows:

```
Sample.class -> classes
Sample.class -> lib/jackson-core-2.8.4.jar
Sample.class -> lib/jackson-databind-2.8.4.jar
Sample.class -> java.base
Sample.class -> jdk.unsupported
```

2. A more verbose dependency information of the contents of the analyzed file at the package or class level (depending on the command-line options). This consists of three columns: column 1 contains the name of the package/class, column 2 contains the name of the dependent package, and column 3 contains the name of the

module/JAR where the dependency is found. A sample output looks like the following:

```
com.packt  -> com.fasterxml.jackson.core.type  jackson-core-2.8.4.jar
com.packt  -> com.fasterxml.jackson.databind   jackson-databind-
                                               2.8.4.jar
com.packt  -> com.packt.model                  sample.jar
```

There's more...

We have seen quite a few options of the `jdeps` command. There are a few more related to filtering the dependencies and filtering the classes to be analyzed. Apart from that, there are a few options that deal with module paths.

The following are the options that can be tried out:

- `-e`, `-regex`, `--regex`: These find dependencies matching the given pattern.
- `-f`, `-filter`: These exclude dependencies matching the given pattern.
- `-filter:none`: This allows no filtering applied via `filter:package` or `filter:archive`.
- `-filter:package`: This excludes dependencies within the same package. This is the default option. For example, if we added `-filter:none` to `jdeps sample.jar`, it would print the dependency of the package to itself.
- `-filter:archive`: This excludes dependencies within the same archive.
- `-filter:module`: This excludes dependencies in the same module.
- `-P`, `-profile`: This is used to show the profile of the package, whether it is in compact1, compact2, compact3, or Full JRE.
- `-R`, `-recursive`: These recursively traverse all the runtime dependencies; they are equivalent to the `-filter:none` option.

Creating a simple modular application

You should be wondering what this modularity is all about and how to create a modular application in Java. In this recipe, we will try to clear the mystery around creating modular applications in Java by walking you through a simple example. Our goal is to show you how to create a modular application; hence, we picked a simple example so as to focus on our goal.

Our example is a simple advanced calculator, which checks whether a number is prime, calculates the sum of prime numbers, checks whether a number is even, and calculates the sum of even and odd numbers.

Getting ready

We will divide our application into two modules:

- The `math.util` module, which contains the APIs for performing the mathematical calculations
- The `calculator` module, which launches an advanced calculator

How to do it...

1. Let's implement the APIs in the `com.packt.math.MathUtil` class, starting with the `isPrime(Integer number)` API:

```
public static Boolean isPrime(Integer number){
  if ( number == 1 ) { return false; }
  return IntStream.range(2,num).noneMatch(i -> num % i == 0 );
}
```

2. The next step is to implement the `sumOfFirstNPrimes(Integer count)` API:

```
public static Integer sumOfFirstNPrimes(Integer count){
  return IntStream.iterate(1,i -> i+1)
                  .filter(j -> isPrime(j))
                  .limit(count).sum();
}
```

3. Let's write a function to check whether the number is even:

```
public static Boolean isEven(Integer number){
  return number % 2 == 0;
}
```

4. The negation of `isEven` tells us whether the number is odd. We can have functions to find the sum of the first *N* even numbers and the first *N* odd numbers, as shown here:

```
public static Integer sumOfFirstNEvens(Integer count){
  return IntStream.iterate(1,i -> i+1)
                  .filter(j -> isEven(j))
                  .limit(count).sum();
}

public static Integer sumOfFirstNOdds(Integer count){
  return IntStream.iterate(1,i -> i+1)
                  .filter(j -> !isEven(j))
                  .limit(count).sum();
}
```

We can see in the preceding APIs that the following operations are repeated:

- An infinite sequence of numbers starting from 1
- Filtering the numbers based on some condition

- Limiting the stream of numbers to a given count
- Finding the sum of numbers thus obtained

Based on our observation, we can refactor the preceding APIs and extract these operations into a method, as follows:

```
private static Integer computeFirstNSum(Integer count, IntPredicate filter){
  return IntStream.iterate(1,i -> i+1)
                  .filter(filter)
                  .limit(count).sum();
 }
```

Here, `count` is the limit of numbers we need to find the sum of and `filter` is the condition for picking the numbers for summing.

Let's rewrite the APIs based on the refactoring we just did:

```
public static Integer sumOfFirstNPrimes(Integer count){
  return computeFirstNSum(count, (i -> isPrime(i)));
}

public static Integer sumOfFirstNEvens(Integer count){
  return computeFirstNSum(count, (i -> isEven(i)));
}

public static Integer sumOfFirstNOdds(Integer count){
  return computeFirstNSum(count, (i -> !isEven(i)));
}
```

You must be wondering about the following:

- The `IntStream` class and the related chaining of the methods
- The use of `->` in the code base
- The use of the `IntPredicate` class

If you are indeed wondering, then you need not worry, as we will cover these things in `Chapter` 4, *Going Functional* and `Chapter` 5, *Stream Operations and Pipelines*.

So far, we have seen a few APIs around mathematical computations. These APIs are part of our `com.packt.math.MathUtil` class. The complete code for this class can be found at the location, `chp3/2_simple-modular-math-util/math.util/com/packt/math`, in the code base downloaded for this book.

Let's make this small utility class part of a module named `math.util`. The following are some conventions we use to create a module:

1. Place all the code related to the module under a directory named `math.util` and treat this as our module root directory.
2. In the root folder, place a file by the name `module-info.java`.
3. We then place the packages and the code files under the root directory.

What does `module-info.java` contain?

- The name of the module
- The packages it exports, that is, makes available for other modules to use
- The modules it depends on
- The services it uses
- The service for which it provides implementation

As mentioned in `Chapter 1`, *Installation and a Sneak Peek into Java 9*, the JDK comes bundled with a lot of modules, that is, the existing Java SDK has been modularized! One of those modules is a module named `java.base`. All the user-defined modules implicitly depend (or require) the `java.base` module (think of every class implicitly extending the `Object` class).

Our `math.util` module doesn't depend on any other module (except, of course, on the `java.base` module). However, it makes its API available for other modules (if not, then this module's existence is questionable). Let's go ahead and put this statement into code:

```
module math.util{
  exports com.packt.math;
}
```

We are telling the Java compiler and runtime that our `math.util` module is *exporting* the code in the `com.packt.math` package to any module that depends on `math.util`.



The code for this module can be found at the location, *`chp3/2_simple-modular-math-util/math.util`*.

Now, let's create another module calculator that uses the `math.util` module. This module has a `Calculator` class whose work is to accept the user's choice for which mathematical operation to execute and then the inputs required to execute the operation. The user can choose from five available mathematical operations:

1. Prime number check
2. Even number check
3. Sum of *N* primes
4. Sum of *N* evens
5. Sum of *N* odds

Let's see this in code:

```
private static Integer acceptChoice(Scanner reader){
  System.out.println("************Advanced Calculator************");
  System.out.println("1. Prime Number check");
  System.out.println("2. Even Number check");
  System.out.println("3. Sum of N Primes");
  System.out.println("4. Sum of N Evens");
  System.out.println("5. Sum of N Odds");
  System.out.println("6. Exit");
  System.out.println("Enter the number to choose operation");
  return reader.nextInt();
}
```

Then, for each of the choice, we accept the required inputs and invoke the corresponding `MathUtil` API, as follows:

```
switch(choice){
  case 1:
    System.out.println("Enter the number");
    Integer number = reader.nextInt();
    if (MathUtil.isPrime(number)){
      System.out.println("The number " + number +" is prime");
    }else{
      System.out.println("The number " + number +" is not prime");
    }
  break;
  case 2:
    System.out.println("Enter the number");
    Integer number = reader.nextInt();
    if (MathUtil.isEven(number)){
      System.out.println("The number " + number +" is even");
    }
  break;
  case 3:
```

```
    System.out.println("How many primes?");
    Integer count = reader.nextInt();
    System.out.println(String.format("Sum of %d primes is %d",
          count, MathUtil.sumOfFirstNPrimes(count)));
  break;
  case 4:
    System.out.println("How many evens?");
    Integer count = reader.nextInt();
    System.out.println(String.format("Sum of %d evens is %d",
          count, MathUtil.sumOfFirstNEvens(count)));
  break;
  case 5:
    System.out.println("How many odds?");
    Integer count = reader.nextInt();
    System.out.println(String.format("Sum of %d odds is %d",
          count, MathUtil.sumOfFirstNOdds(count)));
  break;
}
```

The complete code for the `Calculator` class can be found at `chp3/2_simple-modular-math-util/calculator/com/packt/calculator/Calculator.java`.

Let's create the module definition for our `calculator` module in the same way we created for the `math.util` module:

```
module calculator{
  requires math.util;
}
```

In the preceding module definition, we mention that the `calculator` module depends on the `math.util` module by using the keyword, `required`.



The code for this module can be found at `chp3/2_simple-modular-math-util/calculator`.

Let's now compile the code:

```
javac -d mods --module-source-path . $(find . -name "*.java")
```

The preceding command has to be executed from `chp3/2_simple-modular-math-util`.

Also, you should have the compiled code from across both the modules, `math.util` and `calculator` in the `mods` directory. Was it not quite simple? Just a single command and everything including the dependency

between the modules is taken care of by the compiler. We didn't require build tools such as `ant` to manage the compilation of modules.

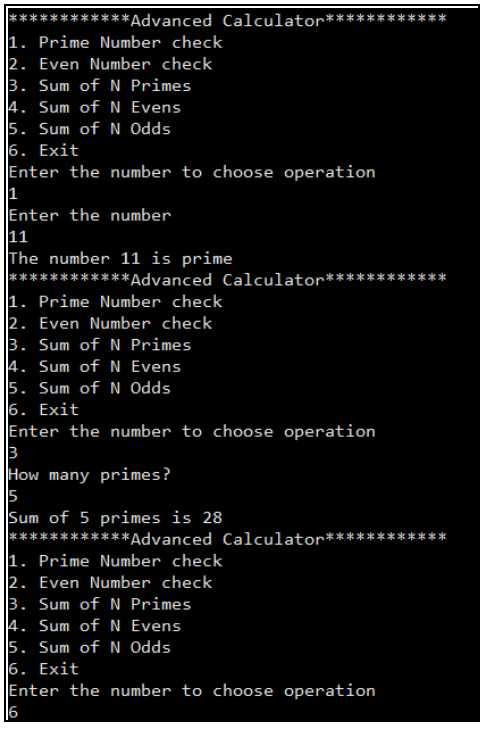
The `--module-source-path` command is the new command-line option to `javac`, specifying the location of our module source code.

Let's now execute the preceding code:

```
java --module-path mods -m calculator/com.packt.calculator.Calculator
```

The `--module-path` command, similar to `--classpath`, is the new command-line option to `java`, specifying the location of the compiled modules.

After running the preceding command, you will see the calculator in action:



Congratulations! With this, we have a simple modular application up and running.

We have provided scripts to test out the code on both Windows and Linux platforms. Please use `run.bat` for Windows and `run.sh` for Linux.

How it works...

Now that you have been through the example, we will look at how to generalize it so that we can apply the same pattern in all our modules. We followed a particular convention to create the modules:

```
|application_root_directory
|--module1_root
|----module-info.java
|----com
|------packt
|--------sample
|----------MyClass.java
|--module2_root
|----module-info.java
|----com
|------packt
|--------test
|----------MyAnotherClass.java
```

We place the module-specific code within its folders with a corresponding `module-info.java` at the root of the folder. This way, the code is organized well.

Let's look into what `module-info.java` can contain. From the Java language specification (http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html), a module declaration is of the following form:

```
{Annotation} [open] module ModuleName { {ModuleStatement} }
```

Here's the syntax, explained:

- `{Annotation}`: This is any annotation of the form `@Annotation(2)`.
- `open`: This keyword is optional. An open module makes all its components accessible at runtime via reflection. However, at compile time and runtime, only those components that are explicitly exported are accessible.
- `module`: This is keyword used to declare a module.
- `ModuleName`: This is the name of the module that is a valid Java identifier with a permissible dot (`.`) between the identifier names--similar to `math.util`.

- `{ModuleStatement}`: This is a collection of the permissible statements within a module definition. Let's expand this next.

A module statement is of the following form:

```
ModuleStatement:
  requires {RequiresModifier} ModuleName ;
  exports PackageName [to ModuleName {, ModuleName}] ;
  opens PackageName [to ModuleName {, ModuleName}] ;
  uses TypeName ;
  provides TypeName with TypeName {, TypeName} ;
```

The module statement is decoded here:

- `requires`: This is used to declare a dependency on a module. `{RequiresModifier}` can be **transitive**, **static**, or both. Transitive means that any module that depends on the given module also implicitly depends on the module that is required by the given module transitively. Static means that the module dependence is mandatory at compile time, but optional at runtime. Some examples are `requires math.util`, `requires transitive math.util`, and `requires static math.util`.
- `exports`: This is used to make the given packages accessible to the dependent modules. Optionally, we can force the package accessibility to specific modules by specifying the module name, such as `exports com.package.math to claculator`.
- `opens`: This is used to open a specific package. We saw earlier that we can open a module by specifying the `open` keyword with the module declaration. But this can be less restrictive. So, to make it more restrictive, we can open a specific package for reflective access at runtime by using the `opens` keyword: `opens com.packt.math`.
- `uses`: This is used to declare a dependency on a service interface that is accessible via `java.util.ServiceLoader`. The service interface can be in the current module or in any module that the current module depends on.
- `provides`: This is used to declare a service interface and provide it with at least one implementation. The service interface can be declared in the current module or in any other dependent module. However, the service implementation must be provided in the same module; otherwise, a compile-time error would occur.

We will look at the `uses` and `provides` clauses in more detail in our recipe, *Using services to create loose coupling between consumer and provider modules*.

The module source of all modules can be compiled at once using the `--module-source-path` command-line option. This way, all the modules will be compiled and placed in their corresponding directories under the directory provided by the `-d` option. For example, `javac -d mods --module-source-path . $(find . -name "*.java")` compiles the code in the current directory into a `mods` directory.

Running the code is equally simple. We specify the path where all our modules are compiled into, using the command-line option, `--module-path`. Then, we mention the module name along with the fully qualified main class name using the command-line option, `-m`, for example, `java --module-path mods -m calculator/com.packt.calculator.Calculator`.

See also

Look at the recipe, *Compiling and running a Java application* from `Chapter 1`, *Installation and a Sneak Peek into Java 9*, where we try out another modular application.

Creating a modular JAR

Compiling modules into a class is good, but it is not suitable for sharing binaries and deployment. JARs are better formats for sharing and deployment. We can package the compiled module into JARs, and the JARs that contain `module-info.class` at its top level are called **modular JARs**. In this recipe, we will look at how to create modular JARs, and we'll also look at how to execute the application, which is composed of multiple modular JARs

Getting ready

We have seen and created a simple modular application in the recipe, *Creating a simpler modular application*. In order to build a modular JAR, we will make use of the sample code available at `chp3/3_modular_jar`. This sample code contains two modules: `math.util` and `calculator`. We will create modular JARs for both the modules.

How to do it...

1. Compile the code and place the compiled classes in a directory, say `mods`:

```
javac -d mods --module-source-path . $(find . -name *.java)
```

2. Build a modular JAR for the `math.util` module:

```
jar --create --file=mlib/math.util@1.0.jar --module-version 1.0
-C mods/math.util .
```



Do not forget the dot (.) in the end in the preceding code.

3. Build a modular JAR for the `calculator` module, specifying the main class to make the JAR executable:

```
jar --create --file=mlib/calculator@1.0.jar --module-version 1.0
--main-class com.packt.calculator.Calculator -C mods/calculator .
```

   The critical piece in the preceding command is the `--main-class` option. This enables us to execute the JAR without providing the main class information during execution.

4. Now we have two JARs in the `mlib` directory: `math.util@1.0.jar` and `calculator@1.0.jar`. These JARs are called modular JARs. If you want to run the example, you can use the following command:

```
java -p mlib -m calculator
```

A new command-line option for the JAR command has been introduced in Java 9, called `-d` or `--describe-module`. This prints the information about the module that the modular JAR contains:

```
jar -d --file=mlib/calculator@1.0.jar
```

The output of `jar -d` for `calculator@1.0.jar` is as follows:

```
calculator@1.0
  requires mandated java.base
  requires math.util
  conceals com.packt.calculator
  main-class com.packt.calculator.Calculator

jar -d --file=mlib/math.util@1.0.jar
```

The output of `jar -d` for `math.util@1.0.jar` is as follows:

```
math.util@1.0
  requires mandated java.base
  exports com.packt.math
```

We have provided the following scripts to try out the recipe code on Windows:

- `compile-math.bat`
- `compile-calculator.bat`
- `jar-math.bat`
- `jar-calculator.bat`
- `run.bat`

We have provided the following scripts to try out the recipe code on Linux:

- `compile.sh`
- `jar-math.sh`
- `jar-calculator.sh`
- `run.sh`

You have to run the scripts in the order they have been listed.

Using a module JAR with pre-JDK 9 applications

It would be amazing if our modular JARs could be run with pre-JDK 9 applications. This way, we will not be concerned with writing another version of our API for pre-JDK 9 applications. The good news is that we can use our modular JARs just as if they were ordinary JARs, that is, JARs without `module-info.class` at its root. We will see how to do so in this recipe.

Getting ready

For this recipe, we will need a modular jar and a non-modular application. Our modular code can be found at `chp3/4_modular_jar_with_pre_java9/math.util` (this is the same `math.util` module that we created in our recipe, *Creating a simple modular application*). Let's compile this modular code and create a modular JAR by using the following commands:

```
javac -d classes --module-source-path . $(find math.util -name *.java)
mkdir mlib
jar --create --file mlib/math.util.jar -C classes/math.util .
```

We have also provided a `jar-math.bat` script at `chp3/4_modular_jar_with_pre_java9`, which can be used to create modular JARs on Windows. We have our modular JAR. Let's verify by using the `-d` option of the `jar` command:

```
jar -d --file mlib/math.util@1.0.jar
math.util@1.0
  requires mandated java.base
  exports com.packt.math
```

How to do it...

Let's now create a simple application, which is non-modular. Our application will consist of one class named `NonModularCalculator`, which borrows its code from the class, `Calculator`, in the recipe *Creating a simple modular application*.

You can find the `NonModularCalculator` class definition in the `com.packt.calculator` package under the directory, `chp3/4_modular_jar_with_pre_java9/calculator`. As it is non-modular, it doesn't need a `module-info.java` file. This application makes use of our modular JAR `math.util.jar` to execute some mathematical calculations.

At this point, you should have the following:

- A modular JAR named `math.util@1.0.jar`
- A non-modular application consisting of the `NonModularCalculator` package

Now we need to compile our `NonModularCalculator` class:

```
javac -d classes/ --source-path calculator $(find calculator -name *.java)
```

After running the previous command, you will see a list of errors saying that the `com.packt.math` package doesn't exist, the `MathUtil` symbol cannot be found, and so on. You guessed it right; we missed providing the location of our modular JAR for the compiler. Let's add the modular jar location using the `--class-path` option:

```
javac --class-path mlib/* -d classes/ --source-path calculator $(find calculator
-name *.java)
```

Now, we have successfully compiled our non-modular code, which was dependent on the modular JAR. Let's run the compiled code:

```
java -cp classes:mlib/* com.packt.calculator.NonModularCalculator
```

Congratulations! You have successfully used your modular JAR with a non-modular application. Amazing, right!

We have provided the following scripts at `chp3/4_modular_jar_with_pre_java9` to run the code on the Windows platform:

- `compile-calculator.bat`
- `run.bat`

See also

We recommend you to try out the following recipes:

- Creating a simple modular application
- Creating a modular JAR

Bottom-up migration

Now that Java 9 is out of the door, the much-awaited modularity feature is now available to be adopted by developers. At some point or the other, you will be involved in migrating your application to Java 9 and, hence, trying to modularize it. A change of such magnitude, which involves third-party libraries and rethinking of the code structure, would require proper planning and implementation. The Java team has suggested two migration approaches:

- Bottom-up migration
- Top-down migration

Before going into learning about bottom-up migration, it's important to understand what unnamed module and automatic module are. Suppose you are accessing a type not available in any of the modules; in such a case, the module system will search for the type on the classpath, and if found, the type becomes part of an unnamed module. This is similar to the classes we write that do not belong to any package, but Java adds them to an unnamed package so as to simplify the creation of new classes.

So, an unnamed module is a catch-all module without a name, which contains all those types that are not part of any modules, but are found in the classpath. An unnamed module can access all the exported types of all the named modules (user-defined modules) and built-in modules (Java platform modules). On the other hand, a named module (user-defined module) will not be able to access the types in the unnamed module. In other words, a named module cannot declare dependency on an unnamed module. If at all you want to declare dependency, how would you do that? An unnamed module doesn't have a name!



With the concept of unnamed modules, you can take your Java 8 application as is and run it on Java 9 (except for any



deprecated internal APIs, which might not be available for user code in Java 9).

You may have seen this if you have tried out the *Using jdeps to find dependencies in a Java application* recipe, where we had a non-modular application and were able to run it on Java 9. However, running as is on Java 9 would defeat the purpose of introducing the modular system.



If a package is defined in both named and unnamed modules, the one in the named module would be given preference over the one in the unnamed module. This helps in preventing conflict of packages when they come from both named and unnamed modules.

Automatic modules are those that are automatically created by the JVM. These modules are created when we introduce the classes packaged in JARs in the module path instead of the classpath. The name of this module will be derived from the name of the JAR without the `.jar` extension and, hence, is different from unnamed modules. Alternatively, one can provide the name for these automatic modules by providing the module name against `Automatic-Module-Name` in the JAR manifest file. These automatic modules export all the packages present in it and also depend on all the automatic and named (user/JDK) modules.

Based on this explanation, modules can be classified into the following:

- **Unnamed modules**: The code available on classpath and not available on the module path is placed in an unnamed module
- **Named modules**: all those modules which have a name associated with it - this can be user defined modules and JDK modules.
- **Automatic modules**: all those modules which are implicitly created by JVM based on the jar files present in the module path
- **Implicit modules**: modules which are implicitly created. They are same as automatic modules
- **Explicit modules**: all modules which are created explicitly by user or JDK.

But the unnamed module and automatic module are a good first step to start your migration. So let's get started!

Getting ready

We need a non-modular application which we will eventually modularize. We have already created a simple application whose source code is available at the location `chp3/6_bottom_up_migration_before`. This simple application has 3 parts to it:

1. A math utility library which contains our favorite mathematical APIs namely: prime checker, even number checker, sum of primes, sum of evens and sum of odds. The code for this is available at the location `chp3/6_bottom_up_migration_before/math_util`.
2. A banking utility library which contains API to compute simple interest and compound interest. The code for this is available at the location `chp3/6_bottom_up_migration_before/banking_util`.
3. Our calculator app which helps us with our mathematical and banking calculations. To make this more interesting we will output the results in JSON and for this, we will make use of Jackson JSON API. The code for this is available at the location `chp3/6_bottom_up_migration_before/calculator`.

After you have copied or downloaded the code, we will compile and build respective jars. So use the following commands to compile and build jars:

```
#Compiling math util

javac -d math_util/out/classes/ -sourcepath math_util/src $(find math_util/src -
name *.java)
jar --create --file=math_util/out/math.util.jar
-C math_util/out/classes/ .

#Compiling banking util

javac -d banking_util/out/classes/ -sourcepath banking_util/src $(find
banking_util/src -name *.java)
jar --create --file=banking_util/out/banking.util.jar
-C banking_util/out/classes/ .

#Compiling calculator

javac -cp
calculator/lib/*:math_util/out/math.util.jar:banking_util/out/banking.util.jar -d
```

```
calculator/out/classes/ -sourcepath calculator/src $(find calculator/src -name
*.java)
```

Let's also create a JAR for this (we make use of the JAR to build the dependency graph but not for running the app)

```
jar --create --file=calculator/out/calculator.jar -C calculator/out/classes/ .
```

Please note that our Jackson JARs are in the calculator/lib so you need not worry about downloading them. Let's run our calculator using the command:

```
java -cp
calculator/out/classes:calculator/lib/*:math_util/out/math.util.jar:banking_util/
out/banking.util.jar com.packt.calculator.Calculator
```

You will see a menu asking for the choice of operation and then you can play around with different operations. Let's now modularize this application!

We have provided `package-*.bat` and run.bat to package and run the application on Windows. And `package-*.sh` and `run.sh` to package and run the application on Linux.

How to do it...

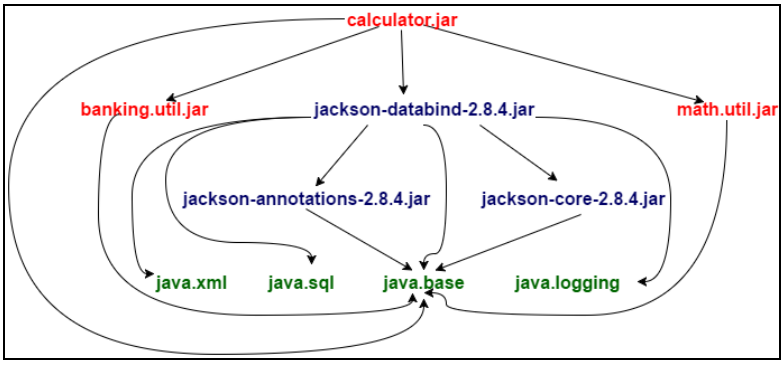
The first step in modularizing your application is to understand its dependency graph. Let's create a dependency graph for our application. And for that, we make use of `jdeps` tool. If you are wondering what `jdeps` tool is, stop right away and read the recipe: *Using jdeps to find dependencies in a Java application*. OK so let's run the `jdeps` tool:

```
jdeps -summary -R -cp calculator/lib/*:math_util/out/*:banking_util/out/*
calculator/out/calculator.jar
```

We are asking `jdeps` to give us a summary of the dependencies of our `calculator.jar` and then do this recursively for each dependency of `calculator.jar`. And the output we get is:

```
banking.util.jar -> java.base
calculator.jar -> banking_util/out/banking.util.jar
calculator.jar -> calculator/lib/jackson-databind-2.8.4.jar
calculator.jar -> java.base
calculator.jar -> math_util/out/math.util.jar
jackson-annotations-2.8.4.jar -> java.base
jackson-core-2.8.4.jar -> java.base
jackson-databind-2.8.4.jar -> calculator/lib/jackson-annotations-2.8.4.jar
jackson-databind-2.8.4.jar -> calculator/lib/jackson-core-2.8.4.jar
jackson-databind-2.8.4.jar -> java.base
jackson-databind-2.8.4.jar -> java.logging
jackson-databind-2.8.4.jar -> java.sql
jackson-databind-2.8.4.jar -> java.xml
math.util.jar -> java.base
```

The preceding output is not clear, hence we have put the same diagrammatically as shown which is as follows:



In bottom-up migration, we start with modularizing the leaf nodes. In our graph the leaf nodes namely `java.xml`, `java.sql`, `java.base` and `java.logging` are already modularized. Let's pick to modularize `banking.util.jar`.



All the code for this recipe is available at the location *`chp3/6_bottom_up_migration_after`*.

Modularizing banking.util.jar

1. Copy `BankUtil.java` from `chp3/6_bottom_up_migration_before/banking_util/src/com/packt/banking` to the location `chp3/6_bottom_up_migration_after/src/banking.util/com/packt/banking`. Two things to take a note of:
    - We have renamed the folder from `banking_util` to `banking.util`. This is to follow the convention of placing module related code under the folder bearing module name.
    - We have placed the package directly under the `banking.util` folder and not under `src`, again this is to follow the convention. And we would be placing all our modules under the `src` folder.
2. Create the module definition file `module-info.java` under `chp3/6_bottom_up_migration_after/src/banking.util` with the following definition:

```
module banking.util{
  exports com.packt.banking;
}
```

3. From within the folder `6_bottom_up_migration_after`, compile the java code of the modules by running the command:

```
javac -d mods --module-source-path src $(find src -name *.java)
```

4. You will see that the java code in the module `banking.util` is compiled into the mods directory.
5. Let's create a modular JAR for this module:

```
jar --create --file=mlib/banking.util.jar -C mods/banking.util .
```

If you are wondering what a modular JAR is, feel free to read through the recipe, *Creating a modular JAR* in this chapter.

Now that we have modularized `banking.util.jar`, let's use this modular jar in place of the non-modular JAR used in *Getting ready* section earlier. You

should execute the following from the `6_bottom_up_migration_before` folder because we haven't yet completely modularized the app.

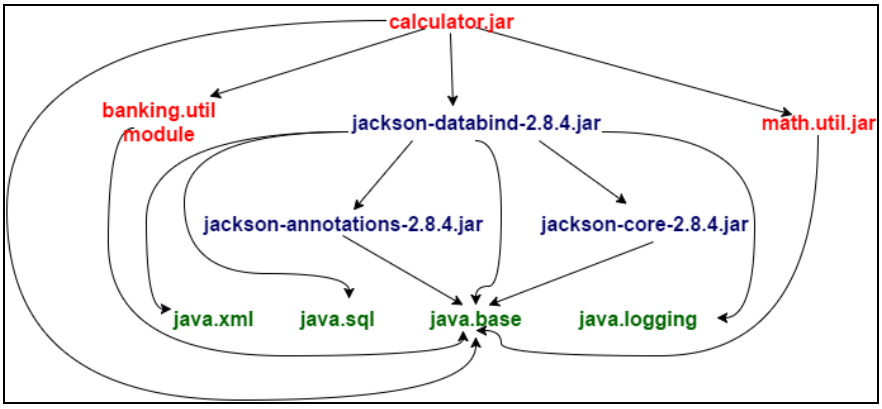
```
java --add-modules ALL-MODULE-PATH --module-path
../6_bottom_up_migration_after/mods/banking.util -cp
calculator/out/classes:calculator/lib/*:math_util/out/math.util.jar
com.packt.calculator.Calculator
```



`--add-modules` option tells the Java runtime to include the modules either by module name or by predefined constants namely: `ALL-MODULE-PATH`, `ALL-DEFAULT`, `ALL-SYSTEM`. We made use of `ALL-MODULE-PATH` to add module which is available on our module path

`--module-path` option tells the Java runtime the location of our modules.

You will see that our calculator is running as usual. Try out simple interest calculation, compound interest calculation to check if the `BankUtil` class is found. So our dependency graph now looks like:



Modularizing math.util.jar

1. Copy `MathUtil.java` from `chp3/6_bottom_up_migration_before/math_util/src/com/packt/math` to the location `chp3/6_bottom_up_migration_after/src/math.util/com/packt/math`.
2. Create the module definition file `module-info.java` under `chp3/6_bottom_up_migration_after/src/math.util` with the following definition:

```
module math.util{
  exports com.packt.math;
}
```

3. From within the folder `6_bottom_up_migration_after`, compile the Java code of the modules by running the command:

```
javac -d mods --module-source-path src $(find src -name *.java)
```

4. You will see that the Java code in the module `math.util` and `banking.util` is compiled into the `mods` directory.
5. Let's create a modular JAR for this module:

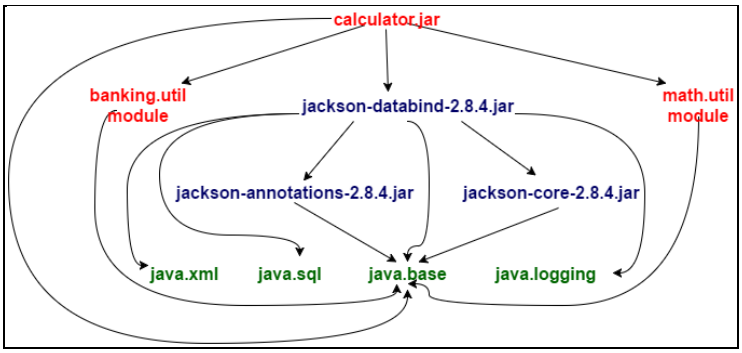
```
jar --create --file=mlib/math.util.jar -C mods/math.util .
```

   If you are wondering what a modular jar is, feel free to read through the recipe, *Creating a modular JAR* in this chapter.

6. Now that we have modularized `math.util.jar`, let's use this modular jar in place of the non-modular jar used in *Getting ready* section earlier. You should execute the below from the `6_bottom_up_migration_before` folder because we haven't yet completely modularized the app:

```
java --add-modules ALL-MODULE-PATH --module-path
../6_bottom_up_migration_after/mods/banking.util:
../6_bottom_up_migration_after/mods/math.util
-cp calculator/out/classes:calculator/lib/*
com.packt.calculator.Calculator
```

This time as well our app is running fine. And the dependency graph looks like:



We cannot modularize calculator.jar because it depends on one another non-modular code jackson-databind. And we cannot modularize `jackson-databind` as it is not maintained by us. So we cannot achieve 100% modularity for our application. We introduced you to Unnamed modules at the beginning of this recipe. All our non-modular code in the classpath are grouped in unnamed modules which means all jackson related code can still remain in the unnamed module and we can try to modularize `calculator.jar`. But we cannot do so because `calculator.jar` cannot declare dependency on `jackson-databind-2.8.4.jar` (because it is an unnamed module and named modules cannot declare dependency on unnamed modules).

A way to get around this is to make the jackson related code as automatic modules. We can do this by moving the jars related to jackson namely:

- `jackson-databind-2.8.4.jar`
- `jackson-annotations-2.8.4.jar`
- `jackson-core-2.8.4.jar`

To our mods folder under the folder `6_bottom_up_migration_after` using the following commands:

```
$ pwd
/root/java9-samples/chp3/6_bottom_up_migration_after
$ cp ../6_bottom_up_migration_before/calculator/lib/*.jar mlib/
$ mv mlib/jackson-annotations-2.8.4.jar mods/jackson.annotations.jar
$ mv mlib/jackson-core-2.8.4.jar mods/jackson.core.jar
$ mv mlib/jackson-databind-2.8.4.jar mods/jackson.databind.jar
```

The reason for renaming the jars is that the name of the module has to be a valid identifier (should not be only numeric, should not contain - and other rules) separated with . and as the names are derived from the name of the JAR files, we had to rename the JAR files to conform to Java identifier rules.

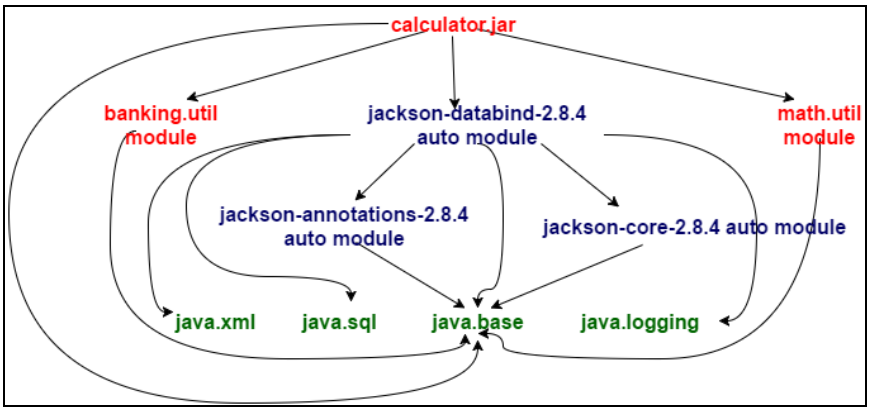


Create a new `mlib` directory if it is not present under `6_bottom_up_migration_after`.

Let's now run our calculator program again using the command:

```
java --add-modules ALL-MODULE-PATH --module-path
../6_bottom_up_migration_after/mods:../6_bottom_up_migration_after/mlib -cp
calculator/out/classes com.packt.calculator.Calculator
```

The application will run as usual. You will notice that our `-cp` option value is getting smaller as all the dependent libraries have been moved as modules in the module path. The dependency graph now looks like:



Modularizing calculator.jar

The last step in the migration is to modularize `calculator.jar`. Follow the below steps to modularize it:

1. Copy the folder `com` from `chp3/6_bottom_up_migration_before/calculator/src` to the location `chp3/6_bottom_up_migration_after/src/calculator`.
2. Create the module definition file `module-info.java` under `chp3/6_bottom_up_migration_after/src/calculator` with the following definition:

```
module calculator{
  requires math.util;
  requires banking.util;
  requires jackson.databind;
  requires jackson.core;
  requires jackson.annotations;
}
```

3. From within the folder `6_bottom_up_migration_after`, compile the Java code of the modules by running the command:

```
javac -d mods --module-path mlib:mods --module-source-path src $(find src -
name *.java)
```

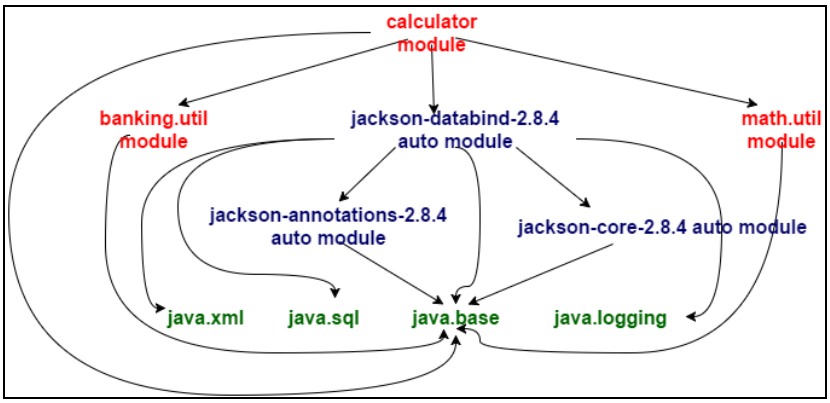
4. You will see that the Java code in all our modules is compiled into the mods directory. Please note that you should have the automatic modules (that is, jackson related JARs) already placed in `mlib` directory.
5. Let's create a modular JAR for this module and also mention which is the `main` class:

```
jar --create --file=mlib/calculator.jar --main-
class=com.packt.calculator.Calculator -C mods/calculator .
```

6. Now we have a modular JAR for our calculator module which is our main module as it contains the `main` class. With this, we have also modularized our complete application. Let's run the following command from the folder: `6_bottom_up_migration_after`:

```
java -p mlib:mods -m calculator
```

So we have seen how we modularized a non-modular application using a bottom-up migration approach. The final dependency graph looks something like:



The final code for this modular application can be found in the location: `chp3/6_bottom_up_migration_after`.



We could have done modification in line that is, modularize the code in the same directory `6_bottom_up_migration_before`*. But we prefer to do it separately in a different directory* `6_bottom_up_migration_after` *so as to keep it clean and not disturb the existing code base.*

How it works...

The concept of unnamed modules helped us to run our non-modular application on Java 9. The use of both module path and classpath helped us to run the partly modular application while we were doing the migration. We started with modularizing those code base which were not dependent on any non-modular code. And any code base which we couldn't modularize, we converted them into automatic modules. And thereby enabling us to modularize the code which was dependent on such code base. Eventually, we ended up with a completely modular application.

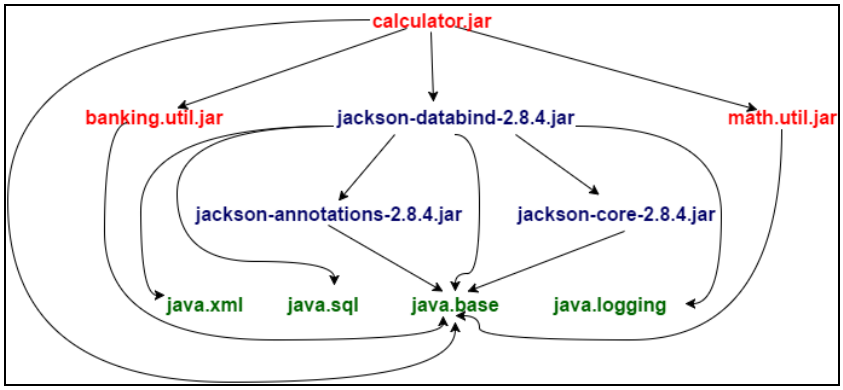
Top-down migration

The other technique for migration is the top-down migration. In this approach, we start with the root JAR in the dependency graph of the JARs.



JARs indicate a code base. We have assumed that the code base is available in the form of JARs and hence the dependency graph which we get has nodes which are JARs.

Modularizing the root of the dependency graph would mean that all other JARs on which this root depends have to be modular. Otherwise, this modular root cannot declare a dependency on unnamed modules. Let's consider the example non-modular application we introduced in our previous recipe. The dependency graph looks something like:



We extensively make use of automatic modules in top-down migration. Automatic modules are those modules which are implicitly created by the JVM. These are created based on the non-modular JARs available in the module path.

Getting ready

We will make use of the calculator example which we introduced in the previous recipe *Bottom-up migration*. Go ahead and copy the non-modular code from the location: `chp3/7_top_down_migration_before`. Use the following commands if you wish to run it and see if it's working:

```
$ javac -d math_util/out/classes/ -sourcepath math_util/src $(find math_util/src
-name *.java)

$ jar --create --file=math_util/out/math.util.jar
-C math_util/out/classes/ .

$ javac -d banking_util/out/classes/ -sourcepath banking_util/src $(find
banking_util/src -name *.java)

$ jar --create --file=banking_util/out/banking.util.jar
-C banking_util/out/classes/ .

$ javac -cp
calculator/lib/*:math_util/out/math.util.jar:banking_util/out/banking.util.jar -d
calculator/out/classes/ -sourcepath calculator/src $(find calculator/src -name
*.java)

$ java -cp
calculator/out/classes:calculator/lib/*:math_util/out/math.util.jar:banking_util/
out/banking.util.jar com.packt.calculator.Calculator
```



We have provided `package-*.bat` and `run.bat` to package and run the code on Windows. And `package-*.sh` and `run.sh` to package and run the code on Linux.

How to do it...

We will be modularizing the application under the directory `chp3/7_top_down_migration_after`. Create two directories `src` and `mlib` under `chp3/7_top_down_migration_after`.

Modularizing the calculator

1. We cannot modularize the calculator until we have modularized all its dependencies. But modularizing its dependencies might be easier at times and not so at other times especially in cases where the dependency is from a third party. In such scenarios, we make use of automatic modules. We copy the non-modular JARs under the folder `mlib` and ensuring the name of the JAR is of the form `<identifier>(.<identifier>)*` where `<identifier>` is a valid Java identifier:

```
$ cp ../7_top_down_migration_before/calculator/lib/jackson-
annotations-
2.8.4.jar mlib/jackson.annotations.jar

$ cp ../7_top_down_migration_before/calculator/lib/jackson-core-
2.8.4.jar
mlib/jackson.core.jar

$ cp ../7_top_down_migration_before/calculator/lib/jackson-databind-
2.8.4.jar mlib/jackson.databind.jar

$ cp ../7_top_down_migration_before/banking_util/out/banking.util.jar
mlib/

$ cp ../7_top_down_migration_before/math_util/out/math.util.jar mlib/
```



We have provided script `copy-non-mod-jar.bat` and `copy-non-mod-jar.sh` to copy the jars easily.

Let's see what all we copied into `mlib`:

```
$ ls mlib
banking.util.jar  jackson.annotations.jar  jackson.core.jar
jackson.databind.jar  math.util.jar
```



The `banking.util.jar` and `math.util.jar` will exist only if you have compiled and JAR'd the code in the `chp3/7_top_down_migration_before/banking_util` and `chp3/7_top_down_migration_before/math_util` directories. We did this in the Getting ready section earlier.

2. Create a new folder `calculator` under `src`. This will contain the code for the `calculator` module.
3. Create `module-info.java` under the `chp3/7_top_down_migration_after/src/calculator` directory that contains the following**:**

```
module calculator{
  requires math.util;
  requires banking.util;
  requires jackson.databind;
  requires jackson.core;
  requires jackson.annotations;
}
```

4. Copy the directory `chp3/7_top_down_migration_before/calculator/src/com` and all the code under it to `chp3/7_top_down_migration_after/src/calculator`.

5. Compile the calculator module:

```
#On Linux
javac -d mods --module-path mlib --module-source-path src $(find
src -name *.java)

#On Windows
javac -d mods --module-path mlib --module-source-path src
src\calculator\module-info.java
src\calculator\com\packt\calculator\Calculator.java
src\calculator\com\packt\calculator\commands\*.java
```

6. Create the modular JAR for `calculator` module:

```
jar --create --file=mlib/calculator.jar --main-
class=com.packt.calculator.Calculator -C mods/calculator/ .
```

7. Run the `calculator` module :

```
java --module-path mlib -m calculator
```

We will see that our calculator is executing correctly. You can try out different operations to verify if all of them are executing correctly.

Modularizing banking.util

As this doesn't depend on other non-module code we can directly convert this into a module by following the steps:

1. Create a new folder `banking.util` under `src`. This will contain the code for `banking.util` module.
2. Create `module-info.java` under the directory `chp3/7_top_down_migration_after/src/banking.util` which contains the following**:**

```
module banking.util{
  exports com.packt.banking;
}
```

3. Copy the directory `chp3/7_top_down_migration_before/banking_util/src/com` and all the code under it to `chp3/7_top_down_migration_after/src/banking.util`.

4. Compile the modules:

```
#On Linux
javac -d mods --module-path mlib --module-source-path src $(find
src -name *.java)

#On Windows
javac -d mods --module-path mlib --module-source-path src
src\banking.util\module-info.java
src\banking.util\com\packt\banking\BankUtil.java
```

5. Create modular JAR for `banking.util` module. This will replace the non-modular `banking.util.jar` already present in `mlib`:

```
jar --create --file=mlib/banking.util.jar -C mods/banking.util/ .
```

6. Run the `calculator` module to test if the `banking.util` modular JAR has been created successfully:

```
java --module-path mlib -m calculator
```

7. You should see the calculator getting executed. Play around with different operations to ensure there is no class not found issues.

Modularizing math.util

1. Create a new folder `math.util` under `src`. This will contain the code for `math.util` module.
2. Create `module-info.java` under the directory `chp3/7_top_down_migration_after/src/math.util` which contains the following**:**

```
module math.util{
  exports com.packt.math;
}
```

3. Copy the directory `chp3/7_top_down_migration_before/math_util/src/com` and all the code under it to `chp3/7_top_down_migration_after/src/math.util`.

4. Compile the modules:

```
#On Linux
javac -d mods --module-path mlib --module-source-path src $(find
src -name *.java)

#On Windows
javac -d mods --module-path mlib --module-source-path src
src\math.util\module-info.java
src\math.util\com\packt\math\MathUtil.java
```

5. Create modular JAR for `banking.util` module. This will replace the non-modular `banking.util.jar` already present in `mlib`:

```
jar --create --file=mlib/math.util.jar -C mods/math.util/ .
```

6. Run the `calculator` module to test if the `math.util` modular JAR has been created successfully.

```
java --module-path mlib -m calculator
```

7. You should see the calculator getting executed. Play around with different operations to ensure there is no class not found issues.

With this, we have completely modularized the application baring the Jackson libraries which we have converted to automatic modules.

We would prefer the top-down approach for migration. This is because we don't have to deal with classpath and module-path at the same time. We can make everything into automatic modules and then use the module-path as we keep migrating the non-modular JARs into modular JARs.

Using services to create loose coupling between consumer and provider modules

Generally, in our applications, we have some interfaces and multiple implementations of those interfaces. Then at runtime depending on some condition, we make use of some specific implementation. This principle is called **Dependency Inversion**. This principle is used by the dependency injection frameworks like Spring to create objects of concrete implementations and assign (or inject) into the references of type abstract interface.

Java has for long (since Java 6) supported service-provider loading facility via the `java.util.ServiceLoader` class. Using Service Loader you can have a **service provider interface** (**SPI**) and multiple implementations of the SPI simply called service provider. These service providers are located in the classpath and loaded at run time. When these service providers are located within modules and as we no longer depend on the classpath scanning to load the service provider, we need a mechanism to tell our modules about the service provider and the SPI for which it is providing. In this recipe, we will look at that mechanism using a simple example.

Getting ready

There is nothing specific we need to set up for this recipe. In this recipe, we will take a simple example. We have one `BookService` abstract class which supports CRUD operations. Now, these CRUD operations can work on a SQL DB or on MongoDB or on file system and so on. This flexibility can be provided by using service provider interface and `ServiceLoader` class to load the required service provider implementation.

How to do it...

We have four modules in this recipe:

1. `book.service`: This is the module which contains our service provider interface that is, the service
2. `mongodb.book.service`: This is one of the service provider module
3. `sqldb.book.service`: This is the other service provider module
4. `book.manage`: This is the service consumer module

The following steps demonstrate how to make use of `ServiceLoader` to achieve loose coupling:

1. Create a folder `book.service` under the directory `chp3/8_services/src`. All our code for `book.service` module will be under this folder.
2. Create a new package `com.packt.model` and a new class `Book` under the new package. This is our model class which contains the following properties:

```
public String id;
public String title;
public String author;
```

3. Create a new package `com.packt.service` and a new class `BookService` under the new package. This is our main service interface and the service providers will provide an implementation for this service. Apart from the abstract methods for CRUD operations, one method worth mentioning is the `getInstance()`. This method uses the `ServiceLoader` class to load any one service provider (the last one to be specific) and then use that service provider to get an implementation of the `BookService`. Let's see the following code:

```
public static BookService getInstance(){
  ServiceLoader<BookServiceProvider> sl =
         ServiceLoader.load(BookServiceProvider.class);
  Iterator<BookServiceProvider> iter = sl.iterator();
  if (!iter.hasNext())
    throw new RuntimeException("No service providers found!");
```

```
        BookServiceProvider provider = null;
        while(iter.hasNext()){
          provider = iter.next();
          System.out.println(provider.getClass());
        }
        return provider.getBookService();
      }
```

The first `while` loop is just for the demonstration that the `ServiceLoader` loads all the service providers and we pick one of the service providers. You can conditionally return the service provider as well, but that all depends on the requirements.

4. The other important part is the actual service provider interface. The responsibility of this is to return an appropriate instance of the service implementation. In our recipe, `BookServiceProvider` in the package `com.packt.spi` is a service provider interface.

```
public interface BookServiceProvider{
  public BookService getBookService();
}
```

5. Next is the main part which is module definition. We create `module-info.java` under the directory `chp3/8_services/src/book.service` which contains:

```
module book.service{
  exports com.packt.model;
  exports com.packt.service;
  exports com.packt.spi;
  uses com.packt.spi.BookServiceProvider;
}
```

The `uses` statement in the preceding module definition specifies the service interface which the module discovers using the `ServiceLoader`.

6. Let's now create a service provider module called `mongodb.book.service`. This will provide an implementation for our `BookService` and `BookServiceProvider` interface in `book.service` module. Our idea is that this service provider will implement the CRUD operations using MongoDB datastore.

7. Create a folder `mongodb.book.service` under the directory `chp3/8_services/src`.
8. Create a class `MongoDbBookService` in the package `com.packt.mongodb.service` which extends `BookService` abstract class and provides an implementation for our abstract CRUD operation methods.

```
public void create(Book book){
  System.out.println("Mongodb Create book ... " + book.title);
}

public Book read(String id){
  System.out.println("Mongodb Reading book ... " + id);
  return new Book(id, "Title", "Author");
}

public void update(Book book){
  System.out.println("Mongodb Updating book ... " + book.title);
}

public void delete(String id){
  System.out.println("Mongodb Deleting ... " + id);
}
```

9. Create a class `MongoDbBookServiceProvider` in the package `com.packt.mongodb` which implements the `BookServiceProvider` interface. This is our service discovery, class. Basically, it returns a relevant instance of `BookService` implementation. It overrides the method in `BookServiceProvider` interface as follows:

```
@Override
public BookService getBookService(){
  return new MongoDbBookService();
}
```

10. The module definition is quite interesting. We have to declare in the module definition that this module is a service provider for the interface `BookServiceProvider` and that can be done as follows:

```
module mongodb.book.service{
  requires book.service;
  provides com.packt.spi.BookServiceProvider
           with com.packt.mongodb.MongoDbBookServiceProvider;
}
```

`provides .. with ..` statement is used to specify the service interface and one of the service provider.

11. Let's now create a service consumer module called `book.manage`.
12. Create a new folder `book.manage` under `chp3/8_services/src` which will contain the code for the module.
13. Create a new class called `BookManager` in package `com.packt.manage`. The main aim of this class is to get an instance of `BookService` and then execute its CRUD operations. The instance returned is decided by the service providers loaded by the `ServiceLoader`. The `BookManager` class looks something like:

```
public class BookManager{
  public static void main(String[] args){
    BookService service = BookService.getInstance();
    System.out.println(service.getClass());
    Book book = new Book("1", "Title", "Author");
    service.create(book);
    service.read("1");
    service.update(book);
    service.delete("1");
  }
}
```

14. Let's now compile and run our main module by using the following commands:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java --module-path mods -m book.manage/com.packt.manage.BookManager
class com.packt.mongodb.MongoDbBookServiceProvider
class com.packt.mongodb.service.MongoDbBookService
Mongodb Create book ... Title
Mongodb Reading book ... 1
Mongodb Updating book ... Title
Mongodb Deleting ... 1
```

    In the preceding output, the first line says the service providers available and the second line says which `BookService` implementation we are using.

15. With one service provider, it looks simple. Let's go ahead and add another module `sqldb.book.service` whose module definition would be:

```
module sqldb.book.service{
  requires book.service;
  provides com.packt.spi.BookServiceProvider
           with com.packt.sqldb.SqlDbBookServiceProvider;
}
```

16. The `SqlDbBookServiceProvider` class in the `com.packt.sqldb` package is an implementation of the interface `BookServiceProvider` as follows:

```
@Override
public BookService getBookService(){
  return new SqlDbBookService();
}
```

17. The implementation of CRUD operations is done by the class `SqlDbBookService` in the package `com.packt.sqldb.service`.
18. Let's compile and run the main module, this time with two service providers:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java --module-path mods -m book.manage/com.packt.manage.BookManager
class com.packt.sqldb.SqlDbBookServiceProvider
class com.packt.mongodb.MongoDbBookServiceProvider
class com.packt.mongodb.service.MongoDbBookService
Mongodb Create book ... Title
Mongodb Reading book ... 1
Mongodb Updating book ... Title
Mongodb Deleting ... 1
```

The first two lines print the class names of the available service providers and the third line prints which `BookService` implementation we are using.

Creating a custom modular runtime image using jlink

Java comes in two flavors:

- Java runtime only also called as JRE - this supports execution of Java applications
- Java development kit with Java run time also called as JDK - this supports development and execution of Java applications.

Apart from this, there were 3 compact profiles introduced in Java 8 with the aim of providing runtimes with a smaller footprint in order to run on embedded and smaller devices.

Full SE API	Beans	JNI	JAX-WS
	Preferences	Accessibility	IDL
	RMI-IIOP	CORBA	Print Service
	Sound	Swing	Java 2D
	AWT	Drag and Drop	Input Methods
	Image I/O		
compact3	Security[1]	JMX	
	XML JAXP[2]	Management	Instrumentation
compact2	JDBC	RMI	XML JAXP
compact1	Core (`java.lang.*`)	Security	Serialization
	Networking	Ref Objects	Regular Expressions
	Date and Time	Input/Output	Collections
	Logging	Concurrency	Reflection
	JAR	ZIP	Versioning
	Internationalization	JNDI	Override Mechanism
	Extension Mechanism	Scripting	

1. Adds kerberos, acl, and sasl to compact1 Security.
2. Adds crypto to compact2 XML JAXP.

The preceding image shows the different profiles and the features supported by them.

A new tool called `jLink` is introduced in Java 9 which enables the creation of modular run time images. These run time images are nothing but a

collection of a set of modules and their dependencies. There is a Java enhancement proposal, JEP 220, governing the structure of this run time image.

In this recipe, we will use `jLink` to create a run time image consisting of our modules `math.util`, `banking.util`, and `calculator` along with the Jackson automatic modules.

Getting ready

In the recipe *Creating a simple modular application* we created a simple modular application consisting of the following modules:

- `math.util`
- `calculator` - consists of the main class

We will reuse the same set of modules and code to demonstrate the use of jLink tool. For the convenience of our readers the code can be found at the location: `chp3/9_jlink_modular_run_time_image`.

How to do it...

1. Let's compile the modules:

```
$ javac -d mods --module-path mlib --module-source-path src $(find
src - name *.java)
```

2. Let's create the modular JAR for all the modules:

```
$ jar --create --file mlib/math.util.jar -C mods/math.util .

$ jar --create --file=mlib/calculator.jar --main-
class=com.packt.calculator.Calculator -C mods/calculator/ .
```

3. Let's use `jlink` to create a run time image consisting of the modules: `calculator`, `math.util` and its dependencies:

```
$ jlink --module-path mlib:$JAVA_HOME/jmods --add-modules
calculator,math.util --output image --launcher
launch=calculator/com.packt.calculator.Calculator
```

   The run time image gets created at the location specified with `--output` command line option.

4. The run time image created under the directory image contains `bin` directory among other directories. This `bin` directory consists of a shell script by name `calculator`. This can be used to launch our application:

```
$ ./image/bin/launch

************Advanced Calculator************
1. Prime Number check
2. Even Number check
3. Sum of N Primes
4. Sum of N Evens
5. Sum of N Odds
6. Exit
Enter the number to choose operation
```



We cannot create run time image of modules which contain automatic modules. jLink gives an error if the JAR files are not modular or if there is no `module-info.class`.

Compiling for older platform versions

We have at some point used options `-source` and `-target` to create a java build. The `-source` option is used to indicate the version of java language accepted by the compiler and the `-target` option is used to indicate the version supported by the class files. Often we forget to use `-source` option and by default, `javac` compiles against the latest available Java version and due to this there are chances of newer APIs being used and as a result, the resultant build doesn't run as expected on the target version.

So as to overcome the confusion of providing two different command line options, a new command line option `--release` is introduced in Java 9. This acts as a substitute to `-source`, `-target` and `-bootclasspath` options. The `-bootclasspath` is used to provide the location of the bootstrap class files for a given version *N*.

Getting ready

We have created a simple module called demo which contains a very simple class called `CollectionsDemo` which just puts a few values in the map and iterate over them as follows:

```
public class CollectionsDemo{
  public static void main(String[] args){
    Map<String, String> map = new HashMap<>();
    map.put("key1", "value1");
    map.put("key2", "value3");
    map.put("key3", "value3");
    map.forEach((k,v) -> System.out.println(k + ", " + v));
  }
}
```

Let's compile and run it to see its output:

```
$ javac -d mods --module-source-path src src\demo\module-info.java
src\demo\com\packt\CollectionsDemo.java
$ java --module-path mods -m demo/com.packt.CollectionsDemo
```

The output we get is:

```
key1, value1
key2, value3
key3, value3
```

Let's now compile this to run on Java 8 and then run it on Java 8.

How to do it...

1. As the older version of Java that is, Java 8 and before don't support modules, we would have to get rid of the `module-info.java` while compiling on an older version. So what we did was to not include `module-info.java` during our compilation. So we compiled using the following code:

```
$ javac --release 8 -d mods src\demo\com\packt\CollectionsDemo.java
```

   You can see that we are using the `--release` option targeting Java 8 and not compiling the `module-info.java`.

2. Let's create a JAR file because it becomes easier to transport the java build instead of copying all the class files:

```
$jar --create --file mlib/demo.jar --main-class
com.packt.CollectionsDemo -C mods/ .
```

3. Let's run the preceding JAR in Java 9:

```
$ java -version
java version "9"
Java(TM) SE Runtime Environment (build 9+179)
Java HotSpot(TM) 64-Bit Server VM (build 9+179, mixed mode)

$ java -jar mlib/demo.jar
key1, value1
key2, value3
key3, value3
```

4. Let's run the JAR in Java 8 :

```
$ "%JAVA8_HOME%"\bin\java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)

$ "%JAVA8_HOME%"\bin\java -jar mlib\demo.jar
key1, value1
key2, value3
key3, value3
```

What if we did not use the `-release` option while building on Java 9? Let's try that as well.

1. Compile without using `--release` option and create a JAR out of the resulting class files:

```
$ javac -d mods src\demo\com\packt\CollectionsDemo.java
$ jar --create --file mlib/demo.jar --main-class
com.packt.CollectionsDemo -C mods/ .
```

2. Let's run the JAR on Java 9:

```
$ java -jar mlib/demo.jar
key1, value1
key2, value3
key3, value3
```

   Works as expected

3. Let's run the JAR on Java 8:

```
$ "%JAVA8_HOME%"\bin\java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
```

The output is:

```
$ java -jar mlib\demo.jar

Exception in thread "main" java.lang.UnsupportedClassVersionError:

com/packt/CollectionsDemo has been compiled by a more recent version of the Java
Runtime (class file version 53.0), this version of the Java Runtime only
recognizes class file versions up to 52.0
```

It is clearly stating that there is a mismatch in the version of the class file. As it was compiled for Java 9 (version 53.0), it doesn't run on Java 8 (version 52.0)

How it works...

The data required for compiling to a target older version is stored in the `$JDK_ROOT/lib/ct.sym` file. This information is used by the `--release` option to locate the `bootclasspath`. The `ct.sym` file is a ZIP file containing stripped-down class files corresponding to class files from the target platform versions (taken verbatim from http://openjdk.java.net/jeps/247).

Creating multirelease JARs

Prior to Java 9, it was hard for the developers of a library to adopt the new features introduced in the language without releasing a new library version. But in Java 9 multirelease JARs provide such a functionality where you can bundle certain class files to run when a higher version of Java is being used.

In this recipe, we will show you how to create such a multirelease JAR.

How to do it...

1. Create the required Java code for Java 8 platform. We will add two classes `CollectionUtil.java` and `FactoryDemo.java` in the directory `src\8\com\packt`:

```
public class CollectionUtil{
  public static List<String> list(String ... args){
    System.out.println("Using Arrays.asList");
    return Arrays.asList(args);
  }

  public static Set<String> set(String ... args){
    System.out.println("Using Arrays.asList and set.addAll");
    Set<String> set = new HashSet<>();
    set.addAll(list(args));
    return set;
  }
}

public class FactoryDemo{
  public static void main(String[] args){
    System.out.println(CollectionUtil.list("element1",
                "element2", "element3"));
    System.out.println(CollectionUtil.set("element1",
                "element2", "element3"));
  }
}
```

2. We wish to make use of the `Collection` factory methods introduced in Java 9. So what we can do is create another sub directory under `src` to place our Java 9 related code: `src\9\com\packt` where we will add another `CollectionUtil` class:

```
public class CollectionUtil{
  public static List<String> list(String ... args){
    System.out.println("Using factory methods");
    return List.of(args);
  }
  public static Set<String> set(String ... args){
    System.out.println("Using factory methods");
    return Set.of(args);
  }
}
```

3. The preceding code uses the Java 9 collection factory methods. Compile the source code using the following commands:

```
javac -d mods --release 8 src\8\com\packt\*.java
javac -d mods9 --release 9 src\9\com\packt\*.java
```

Make a note of the `--release` option used to compile the code for different java versions.

4. Let's now create the multirelease JAR:

```
jar --create --file mr.jar --main-class=com.packt.FactoryDemo
-C mods . --release 9 -C mods9 .
```

While creating the JAR we have also mentioned that when running on Java 9 make use of the Java 9 specific code.

5. We will run the `mr.jar` on Java 9:

```
java -jar mr.jar
[element1, element2, element3]
Using factory methods
[element2, element3, element1]
```

6. We will run the `mr.jar` on Java 8:

```
#Linux
$ /usr/lib/jdk1.8.0_144/bin/java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
$ /usr/lib/jdk1.8.0_144/bin/java -jar mr.jar
Using Arrays.asList
[element1, element2, element3]
Using Arrays.asList and set.addAll
Using Arrays.asList
[element1, element2, element3]

#Windows
$ "%JAVA8_HOME%"\bin\java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
$ "%JAVA8_HOME%"\bin\java -jar mr.jar
Using Arrays.asList
[element1, element2, element3]
Using Arrays.asList and set.addAll
Using Arrays.asList
[element1, element2, element3]
```

How it works...

Let's look at the layout of the content in `mr.jar`:

```
jar -tvf mr.jar
```

The contents of the JAR is as shown in the following:

```
  0 Sat Jul 29 00:24:08 UTC 2017 META-INF/
117 Sat Jul 29 00:24:08 UTC 2017 META-INF/MANIFEST.MF
  0 Sat Jul 29 00:19:40 UTC 2017 com/
  0 Sat Jul 29 00:19:40 UTC 2017 com/packt/
966 Sat Jul 29 00:23:54 UTC 2017 com/packt/CollectionUtil.class
678 Sat Jul 29 00:23:54 UTC 2017 com/packt/FactoryDemo.class
  0 Sat Jul 29 00:19:46 UTC 2017 META-INF/versions/9/
  0 Sat Jul 29 00:19:46 UTC 2017 META-INF/versions/9/com/
  0 Sat Jul 29 00:19:46 UTC 2017 META-INF/versions/9/com/packt/
862 Sat Jul 29 00:24:02 UTC 2017 META-INF/versions/9/com/packt/CollectionUtil.class
```

The preceding layout we have `META-INF/versions/9` which contains the Java 9 specific code. Another important thing to note is the contents of the `META-INF/MANIFEST.MF` file. Let's extract the JAR and view its contents:

```
jar -xvf mr.jar

$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: 9 (Oracle Corporation)
Main-Class: com.packt.FactoryDemo
Multi-Release: true
```

The new manifest attribute `Multi-Release` is used to indicate if the JAR is a multirelease JAR or not.

Using Maven to develop a modular application

In this recipe, we will look at using Maven, most popular build tool in Java ecosystem, to develop a simple modular application. We will reuse the idea we had introduced in the Services recipe in this chapter.

Getting ready

We have the following modules in our example:

- `book.manage`: This is the main module which interacts with the data source
- `book.service`: This is the module which contains the service provider interface
- `mongodb.book.service`: This is the module which provides an implementation to the service provider interface
- `sqldb.book.service`: This is the module which provides another implementation to the service provider interface

In the course of this recipe, we will create a maven project and include the preceding JDK modules as maven modules. So let's get started.

How to do it...

1. Create a folder to contain all the modules. We have called it `12_services_using_maven` with the following folder structure:

```
12_services_using_maven
      |---book-manage
      |---book-service
      |---mongodb-book-service
      |---sqldb-book-service
      |---pom.xml
```

2. The `pom.xml` for the parent is:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt</groupId>
  <artifactId>services_using_maven</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>book-service</module>
    <module>mongodb-book-service</module>
    <module>sqldb-book-service</module>
    <module>book-manage</module>
  </modules>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.6.1</version>
        <configuration>
          <source>9</source>
          <target>9</target>
          <showWarnings>true</showWarnings>
          <showDeprecation>true</showDeprecation>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

3. Let's create the structure for the `book-service` Maven module:

```
book-service
 |---pom.xml
 |---src
   |---main
     |---book.service
       |---module-info.java
       |---com
         |---packt
           |---model
             |---Book.java
           |---service
             |---BookService.java
           |---spi
             |---BookServiceProvider.java
```

4. The content of `pom.xml` for `book-service` Maven module is:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.packt</groupId>
    <artifactId>services_using_maven</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>book-service</artifactId>
  <version>1.0</version>
  <build>
    <sourceDirectory>src/main/book.service</sourceDirectory>
  </build>
</project>
```

5. The `module-info.java` is:

```
module book.service{
  exports com.packt.model;
  exports com.packt.service;
  exports com.packt.spi;
  uses com.packt.spi.BookServiceProvider;
}
```

6. The `Book.java` is:

```
public class Book{
  public Book(String id, String title, String author){
    this.id = id;
    this.title = title
    this.author = author;
  }
  public String id;
  public String title;
```

```
    public String author;
  }
```

7. The `BookService.java` is:

```
  public abstract class BookService{
    public abstract void create(Book book);
    public abstract Book read(String id);
    public abstract void update(Book book);
    public abstract void delete(String id);
    public static BookService getInstance(){
      ServiceLoader<BookServiceProvider> sl
           ServiceLoader.load(BookServiceProvider.class);
      Iterator<BookServiceProvider> iter = sl.iterator();
      if (!iter.hasNext())
         throw new RuntimeException("No service providers found!");
      BookServiceProvider provider = null;
      while(iter.hasNext()){
        provider = iter.next();
        System.out.println(provider.getClass());
      }
      return provider.getBookService();
    }
  }
```

8. The `BookServiceProvider.java` is:

```
  public interface BookServiceProvider{
    public BookService getBookService();
  }
```

On similar lines, we define other three Maven modules namely `mongodb-book-service`, `sqldb-book-service` and `book-manager`. The code for this can be found at the location `chp3/12_services_using_maven`.

We can compile the classes and build the required JAR files using the command:

```
mvn clean install
```

We have provided `run-with-mongo.*` to use the `mongodb-book-service` as the service provider implementation and `run-with-sqldb.*` to use the `sqldb-book-service` as the service provider implementation.

The complete code for this recipe can be found at `chp3/12_services_using_maven`.

Going Functional

This chapter introduces a programming paradigm called functional programming and its applicability in Java 9. We will cover the following recipes:

- Understanding and creating a functional interface
- Understanding lambda expressions
- Using method references
- Creating and invoking lambda-friendly APIs
- Leveraging lambda expressions in your programs

Introduction

Functional programming--the ability to treat a certain piece of functionality as an object and to pass it as a parameter or the return value of a method--is a feature present in many programming languages. It avoids the changing of an object state and mutable data. The result of a function depends only on the input data, no matter how many times it is called. This style makes the outcome more predictable, which is the most attractive aspect of functional programming.

Its introduction to Java also allows you to improve parallel programming capabilities in Java 8 by shifting the responsibility of parallelism from the client code to the library. Before this, in order to process elements of Java collections, the client code had to acquire an iterator from the collection and organize the processing of the collection.

In Java 8, new (default) methods were added that accept a function (implementation of a functional interface) as a parameter and then apply it to each element of the collection. So, it is the library's responsibility to organize parallel processing. One example is the `forEach(Consumer)` method that is available in every `Iterable` interface, where `Consumer` is a functional interface. Another example is the `removeIf(Predicate)` method that is available for every `Collection` interface, where `Predicate` is a functional interface too. Then we have the `sort(Comparator)` and `replaceAll(UnaryOperator)` methods that are available for `List` and several other methods, such as `compute()` for `Map`.

Forty-three functional interfaces are provided in the `java.util.function` package. Each of them contains only one abstract method. Lambda expressions take advantage of the one-abstract-method limitation and significantly simplifies the implementation of such an interface.

Without functional programming, the only way to pass some functionality as a parameter in Java would be through writing a class that implements an interface, creating its object, and then passing it as a parameter. But even

the least involved style--using an anonymous class--requires writing too much of code. Using functional interfaces and lambda expressions makes the code shorter, clearer, and more expressive.

Throughout the chapter, we will define and explain these new Java features--functional interfaces and lambda expressions--and demonstrate their applicability in code examples. Bringing these new features into Java makes functions first-class citizens of the language. But taking advantage of their power requires, for those not exposed to functional programming yet, a new way of thinking and organizing the code.

Demonstrating these features and sharing the best practices of using them is the purpose of this chapter.

Understanding and creating a functional interface

In this recipe, you will learn about functional interfaces that are supported since Java 8.

Getting ready

Any interface that has one and only one abstract method is called a functional interface. To help avoid a runtime error, the `@FunctionalInterface` annotation was introduced in Java 8 that tells the compiler about the intent. In our demo code in the previous chapters, we've already had an example of a functional interface:

```
public interface SpeedModel {
  double getSpeedMph(double timeSec,
  int weightPounds, int horsePower);
  enum DrivingCondition {
    ROAD_CONDITION,
    TIRE_CONDITION
  }
  enum RoadCondition {
    //...
  }
  enum TireCondition {
    //...
  }
}
```

The presence of `enum` types or any implemented (default or static) methods does not make it a non-functional interface. Only abstract (not implemented) methods count. So, this is an example of a functional interface too:

```
public interface Vehicle {
  void setSpeedModel(SpeedModel speedModel);
  default double getSpeedMph(double timeSec){ return -1; };
  default int getWeightPounds(){ return -1; }
  default int getWeightKg(){
    return convertPoundsToKg(getWeightPounds());
  }
  private int convertPoundsToKg(int pounds){
    return (int) Math.round(0.454 * pounds);
  }
  static int convertKgToPounds(int kilograms){
    return (int) Math.round(2.205 * kilograms);
  }
}
```

To recap what you have already learned about interfaces in one of the previous chapters, the implementation of the `getWeightPounds()` method will

return `-1` when called by `getWeightKg()`. However, this is true only if the `getWeightPounds()` method is not implemented in a class. Otherwise, the class implementation will be used at runtime.

In addition to default and static interface methods, a functional interface can include any and all abstract methods of the base `java.lang.Object`. In Java, every object is provided with the default implementation of `java.lang.Object` methods, so the compiler and Java runtime ignore such abstract methods.

For example, this is a functional interface too:

```
public interface SpeedModel {
  double getSpeedMph(double timeSec,
  int weightPounds, int horsePower);
  boolean equals(Object obj);
  String toString();
}
```

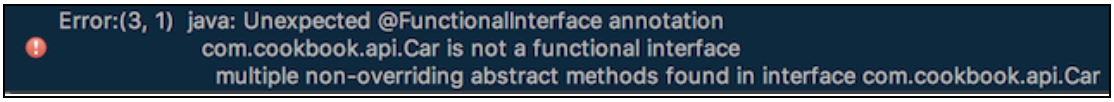
The following is not a functional interface, though:

```
public interface Car extends Vehicle {
  int getPassengersCount();
}
```

This is because the `Car` interface has two abstract methods: its own `getPassengersCount()` method and the `setSpeedModel()` method inherited from the `Vehicle` interface. Say, we add the `@FunctionalInterface` annotation to the `Car` interface:

```
@FunctionalInterface
public interface Car extends Vehicle {
  int getPassengersCount();
}
```

If we do this, the compiler will generate the following error:



Using the `@FunctionalInterface` annotation helps to not only catch errors at compile time, but it also secures reliable communication of the design intent. It helps you or other programmers remember that this interface

cannot have more than one abstract method, which is especially important in case code exists already that relies on such an assumption.

For the same reason, the `Runnable` and `Callable` interfaces (they existed in Java since its earlier versions) in Java 8 were annotated as `@FunctionalInterface` to make this distinction explicit and remind the users about it or those attempting to add another abstract method to them:

```
@FunctionalInterface
interface Runnable { void run(); }

@FunctionalInterface
interface Callable<V> { V call() throws Exception; }
```

Before you create your own functional interface that you plan to use as a parameter for a method, consider avoiding it using one of the forty-three functional interfaces provided in the `java.util.function` package first. Most of them are specializations of the following four interfaces: `Function`, `Consumer`, `Supplier`, and `Predicate`.

How to do it...

The following are the steps you can follow to get familiar with functional interfaces:

1. Look at the functional interface `Function`:

```
@FunctionalInterface
public interface Function<T,R>
```

   Here is its Javadoc, *Accepts one argument of type T and produces result of type R. The functional method is apply(Object).* You can create an implementation of this interface using an anonymous class:

```
Function<Integer, Double> ourFunc = new
                  Function<Integer, Double>(){
  public Double apply(Integer i){
    return i * 10.0;
  }
};
```

   Its only method, namely `apply()`, accepts the value of the type `Integer` (or the primitive `int`, which is going to be autoboxed) as a parameter, then multiplies it by `10`, and returns the value of the type `Double` (or unboxed to the primitive `double`) so we can write the following:

```
System.out.println(ourFunc.apply(1));
```

   The result will be as follows:

```
10.0
```

   In the next recipe, we will introduce a lambda expression and show you how its usage makes the implementation much shorter. But for now, we will continue using an anonymous class.

2. Look at the functional interface `Consumer` (the name helps to remember that the method of this interface accepts a value but does not return anything--it only consumes):

```
public interface Consumer<T>
Accepts a single input argument of type T and returns no result.
The functional method is accept(Object).
```

The implementation of this interface can look like the following:

```
Consumer<String> ourConsumer = new Consumer<String>() {
  public void accept(String s) {
    System.out.println("The " + s + " is consumed.");
  }
};
```

The `accepts()` method receives the parameter value of the type `String` and prints it. Say, we write the following:

```
ourConsumer.accept("Hello!");
```

The result of this will be as follows:

```
The Hello! is consumed.
```

3. Look at the functional interface `Supplier` (the name helps to remember that the method of this interface does not accept any value but does return something--only supplies):

```
public interface Supplier<T>
Represents a supplier of results of type T.
The functional method is get().
```

It means that the only method of this interface is `get()`, which has no input parameter and returns the value of the type `T`. Based on this, we can create a function:

```
Supplier<String> ourSupplier = new Supplier<String>() {
  public String get() {
    String res = "Success";
    //Do something and return result - Success or Error.
    return res;
  }
};
```

The `(get()` method does something and then returns the value of the type `String`, so we can write the following:

```
System.out.println(ourSupplier.get());
```

The result of this will be as follows:

```
Success
```

4. Look at the functional interface `Predicate` (the name helps to remember that the method of this interface returns a boolean--predicates something):

```
@FunctionalInterface
public interface Predicate<T>
```

Its JavaDoc states: *Represents a predicate (boolean-valued function) of one argument of type T. The functional method is test(Object)*. It means that the only method of this interface is `test(Object)` that accepts an input parameter of the type `T` and returns the value of the type `boolean`. Let's create a function:

```
Predicate<Double> ourPredicate = new Predicate<Double>() {
  public boolean test(Double num) {
    System.out.println("Test if " + num +
                       " is smaller than 20");
    return num < 20;
  }
};
```

Its `test()` method accepts a value of the type `Double` as a parameter and returns the value of the type `boolean` so we can write the following:

```
System.out.println(ourPredicate.test(10.0) ?
                   "10 is smaller" : "10 is bigger");
```

The result of this will be as follows:

```
Test if 10.0 is smaller than 20
10 is smaller
```

5. Look at the other 39 functional interfaces in the `java.util.function` package. Notice that they are variations of the four interfaces we have discussed already. These variations are created for the following reasons:

    - For better performance by avoiding autoboxing and unboxing via the explicit usage of the `int`, `double`, or `long` primitives
    - For accepting two input parameters
    - For a shorter notation

    The following functional interfaces are just a few examples from the list of 39 interfaces.

    The functional interface `IntFunction<R>` (its only abstract method is `apply(int)`), accepts an `int` primitive and returns the value of the type `R`. It provides a shorter notation (without generics for the parameter type) and avoids autoboxing (by defining the `int` primitive as the parameter). Here's an example of this:

```
IntFunction<String> function = new IntFunction<String>() {
  public String apply(int i) {
    return String.valueOf(i * 10);
  }
};
```

    The functional interface `BiFunction<T,U,R>` (the `apply(T,U)` method) accepts two parameters of the types `T` and `U` and returns the value of the type `R`. Here's an example of this:

```
BiFunction<String, Integer, Double> function =
      new BiFunction<String, Integer, Double >() {
  public Double apply(String s, Integer i) {
    return (s.length() * 10d)/i;
  }
};
```

The functional interface `BinaryOperator<T>` (the `apply(T,T)` method) accepts two parameters of the type `T` and returns the value of the type `T`. It provides a shorter notation by avoiding repeating the same type three times. Here's an example of this:

```
BinaryOperator<Integer> function =
                  new BinaryOperator<Integer>(){
  public Integer apply(Integer i, Integer j) {
    return i >= j ? i : j;
  }
};
```

The functional interface `IntBinaryOperator` (the `applyAsInt(int,int)` method) accepts two parameters of the type `int` and returns the value of the type `int` too. Here's an example of this:

```
IntBinaryOperator function = new IntBinaryOperator(){
  public int apply(int i, int j) {
    return i >= j ? i : j;
  }
};
```

We will see examples of the usage of such specializations in the following recipes.

How it works...

You can create and pass around an implementation of any functional interface as you do with any object. For example, let's write a method that creates `Function<Integer, Double>`:

```
Function<Integer, Double> createMultiplyBy10(){
  Function<Integer, Double> ourFunc = new Function<Integer, Double>(){
    public Double apply(Integer i){ return i * 10.0; }
  };
  return ourFunc;
}
```

Better yet, we can create a generic method that uses the following:

```
Function<Integer, Double> createMultiplyBy(double num){
  Function<Integer, Double> ourFunc = new Function<Integer, Double>(){
    public Double apply(Integer i){ return i * num; }
  };
  return ourFunc;
}
```

Now we can write this:

```
Function<Integer, Double> multiplyBy10 = createMultiplyBy(10d);
System.out.println(multiplyBy10.apply(1));

Function<Integer, Double> multiplyBy30 = createMultiplyBy(30d);
System.out.println(multiplyBy30.apply(1));
```

The results will be as follows:



Similarly, we can have factory methods that create the `Function<Double, Double>` function, the `Consumer<String>` function, and the `Supplier<String>` function:

```
Function<Double, Double> createSubtract(double num){
  Function<Double, Double> ourFunc = new Function<Double, Double>(){
    public Double apply(Double dbl){ return dbl - num; }
  };
  return ourFunc;
```

```
}
public static Consumer<String> createTalker(String value){
  Consumer<String> consumer = new Consumer<String>() {
    public void accept(String s) {
      System.out.println(s + value);
    }
  };
  return consumer;
}
public static Supplier<String> createResultSupplier(){
  Supplier<String> supplier = new Supplier<String>() {
    public String get() {
      String res = "Success";
      //Do something and return Success or Error.
      return res;
    }
  };
  return supplier;
}
```

Let's use the preceding functions:

```
Function<Double,Double> subtract7 = createSubtract(7.0);
System.out.println(subtract7.apply(10.0));

Consumer<String> sayHappyToSee = createTalker("Happy to see
                                              you again!");
sayHappyToSee.accept("Hello!");

Supplier<String> successOrFailure = createResultSupplier();
System.out.println(successOrFailure.get());
```

The results will be as follows:

```
3.0
Hello!Happy to see you again!
Success
```

We can also have factory methods that create different versions of the `Predicate<Double>` function:

```
Predicate<Double> createIsSmallerThan(double limit){
  Predicate<Double> pred = new Predicate<Double>() {
    public boolean test(Double num) {
      System.out.println("Test if " + num + " is smaller than "
                         + limit);
      return num < limit;
    }
  };
  return pred;
}
Predicate<Double> createIsBiggerThan(double limit){
  Predicate<Double> pred = new Predicate<Double>() {
```

```
    public boolean test(Double num) {
      System.out.println("Test if " + num + " is bigger than "
                         + limit);
      return num > limit;
    }
  };
  return pred;
}
```

Let's use the preceding methods as following:

```
Predicate<Double> isSmallerThan20 = createIsSmallerThan(20d);
System.out.println(isSmallerThan20.test(10d));

Predicate<Double> isBiggerThan18 = createIsBiggerThan(18d);
System.out.println(isBiggerThan18.test(10d));
```

When we use them, we get the following results:

```
Test if 10.0 is smaller than 20.0
true
Test if 10.0 is bigger than 18.0
false
```

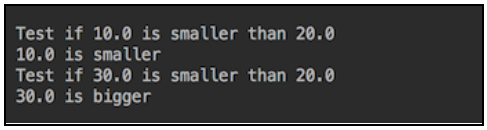
If needed, a function with more complex logic can be composed of several already existing functions:

```
Supplier<String> applyCompareAndSay(int i,
                      Function<Integer, Double> func,
                      Predicate<Double> isSmaller){
  Supplier<String> supplier = new Supplier<String>() {
    public String get() {
      double v = func.apply(i);
      return isSmaller.test(v)? v + " is smaller" : v + " is bigger";
    }
  };
  return supplier;
}
```

We can create it by passing the `multiplyBy10`, `multiplyBy30`, and `isSmallerThan20` functions to the factory method, which we have created before:

```
Supplier<String> compare1By10And20 =
         applyCompareAndSay(1, multiplyBy10, isSmallerThan20);
System.out.println(compare1By10And20.get());
Supplier<String> compare1By30And20 =
         applyCompareAndSay(1, multiplyBy30, isSmallerThan20);
System.out.println(compare1By30And20.get());
```

If we run the preceding code, you'll get the following results:



The first and the third lines come from the `isSmallerThan20` function, while the second and fourth line come from the `compare1By10And20` and `compare1By30And20` functions, correspondingly.

As you see, the introduction of functional interfaces enhances Java by allowing passing functions as parameters. An application developer can now concentrate on the implementation of the function (business process) and not worry about the plumbing of applying it to each element of a collection.

There's more...

Many of the functional interfaces in the `java.util.function` package have default methods that not only enhance their functionality, but also allow you to chain the functions and pass the result of one as an input parameter to another. For example, we can use the default method `andThen(Function after)` of the `Function` interface:

```
Supplier<String> compare1By30Less7To20 =
         applyCompareAndSay(1, multiplyBy30.andThen(subtract7),
                            isSmallerThan20);
System.out.println(compare1By30Less7To20.get());
Supplier<String> compare1By30Less7TwiceTo20 =
         applyCompareAndSay(1, multiplyBy30.andThen(subtract7)
                              .andThen(subtract7),isSmallerThan20);
System.out.println(compare1By30Less7TwiceTo20.get());
```

The `after` function is applied to the result of this function, so naturally, the input type of the `after` function has to be the same or a base type of the result of this function. The result of this code is as follows:

```
Test if 23.0 is smaller than 20.0
23.0 is bigger
Test if 16.0 is smaller than 20.0
16.0 is smaller
```

We could achieve the same result using another default method of the `Function` interface called `compose(Function before)`, which applies the `before` function first before applying this function. Naturally, in this case, we would need to switch positions of the `multiplyBy30` function and the `subtract7`: function:

```
Supplier<String> compare1By30Less7To20 =
         applyCompareAndSay(1, subtract7.compose(multiplyBy30),
                            isSmallerThan20);
System.out.println(compare1By30Less7To20.get());
Supplier<String> compare1By30Less7TwiceTo20 =
         applyCompareAndSay(1, subtract7.compose(multiplyBy30)
                              .andThen(subtract7), isSmallerThan20);
 System.out.println(compare1By30Less7TwiceTo20.get());
```

The result is as follows:

```
Test if 23.0 is smaller than 20.0
23.0 is bigger
Test if 16.0 is smaller than 20.0
16.0 is smaller
```

The `Consumer` interface has the `andThen(Consumer after)` method too, so we can create a dialog using the `sayHappyToSee` function we have created before:

```
Consumer<String> askHowAreYou = createTalker("How are you?");
sayHappyToSee.andThen(askHowAreYou).accept("Hello!");
```

The result will be as follows:

```
Hello!Happy to see you again!
Hello!How are you?
```

The `Supplier` interface does not have default methods, while the `Predicate` interface has one `isEqual(Object targetRef)` static method and three default methods: `and(Predicate other)`, `negate()`, and `or(Predicate other)`. We will demonstrate the usage of the `and(Predicate other)` method.

We can create a predicate, using the already created functions `isSmallerThan20` and `isBiggerThan18`, that checks whether the input value falls between the two values. But before this, we need to overload the `applyCompareAndSay()` factory method by adding another parameter to the signature named `message` that matches the new predicate:

```
Supplier<String> applyCompareAndSay(int i,
                    Function<Integer, Double> func,
                    Predicate<Double> compare, String message) {
  Supplier<String> supplier = new Supplier<String>() {
    public String get() {
      double v = func.apply(i);
      return (compare.test(v)? v + " is " : v + " is not ") + message;
    }
  };
  return supplier;
}
```

Now we can write the following:

```
Supplier<String> compare1By30Less7TwiceTo18And20 =
         applyCompareAndSay(1, multiplyBy30.andThen(subtract7)
                               .andThen(subtract7),
                            isSmallerThan20.and(isBiggerThan18),
```

```
                                "between 18 and 20");
System.out.println(compare1By30Less7TwiceTo18And20.get());
```

We get the following results:

```
Test if 16.0 is smaller than 20.0
Test if 16.0 is bigger than 18.0
16.0 is not between 18 and 20
```

If this coding looks a bit over-engineered and convoluted, it is true. We did it for demo purposes. Good news is that lambda expressions (presented in the next recipe) allow you to achieve the same results in a much more straightforward and clear way.

Before we end this recipe, we would like to mention that functional interfaces of the `java.util.function` package have other helpful default methods. The one that stands out is the `identity()` method, which returns a function that always returns its input argument:

```
Function<Integer, Integer> id = Function.identity();
System.out.println("Function.identity.apply(4) => " + id.apply(4));
```

We will get the following output:

```
Function.identity.apply(4) => 4
```

The `identity()` method is very helpful when some procedure requires you to provide a certain function, but you do not want this function to modify the result.

Other default methods are mostly related to conversion and boxing and unboxing and extracting min and max of two parameters. We encourage you to walk through the API of all the functional interfaces of the `java.util.function` package and get a feeling of the possibilities.

See also

Refer to the following recipes in this chapter:

- Understanding lambda expressions
- Using method references

Understanding lambda expressions

In this recipe, you will learn about lambda expressions that are supported since Java 8.

Getting ready

The examples in the previous recipe (that used anonymous classes for functional interfaces' implementation) looked bulky and felt excessively verbose. For one, there was no need to repeat the interface name because we had declared it already as the type for the object reference. Second, in the case of a functional interface (that had only one abstract method), there was no need to specify the method name to be implemented. The compiler and Java runtime can figure it out anyway. All we needed was to provide the new functionality. This is where a lambda expression comes to the rescue.

How to do it...

The following steps will help you to understand lambda expressions:

1. Consider the following code, for example:

```
Function<Integer, Double> ourFunc =
                  new Function<Integer, Double>(){
  public Double apply(Integer i){ return i * 10.0; }
};
System.out.println(ourFunc.apply(1));

Consumer<String> consumer = new Consumer<String>() {
  public void accept(String s) {
    System.out.println("The " + s + " is consumed.");
  }
};
consumer.accept("Hello!");

Supplier<String> supplier = new Supplier<String>() {
  public String get() {
    String res = "Success";
    //Do something and return result - Success or Error.
    return res;
  }
};
System.out.println(supplier.get());

Predicate<Double> pred = new Predicate<Double>() {
  public boolean test(Double num) {
    System.out.println("Test if " + num +
                       " is smaller than 20");
    return num < 20;
  }
};
System.out.println(pred.test(10.0) ?
                   "10 is smaller" : "10 is bigger");
```

2. Rewrite it using lambda expressions:

```
Function<Integer, Double> ourFunc = i -> i * 10.0;
System.out.println(ourFunc.apply(1));

Consumer<String> consumer =
   s -> System.out.println("The " + s + " is consumed.");
consumer.accept("Hello!");

Supplier<String> supplier = () -> {
  String res = "Success";
  //Do something and return result - Success or Error.
  return res;
```

```
        };
        System.out.println(supplier.get());

        Predicate<Double> pred = num -> {
          System.out.println("Test if " + num + " is smaller than 20");
          return num < 20;
        };
        System.out.println(pred.test(10.0) ?
                           "10 is smaller" : "10 is bigger");
```

3. Run it and you'll get the same result:

```
10.0
The Hello! is consumed.
Success
Test if 10.0 is smaller than 20
10 is smaller
```

How it works...

The syntax of a lambda expression includes the list of parameters, an arrow token (`->`), and a body. The list of parameters can be empty (`()`), without brackets (if there is only one parameter, as in our examples), or a comma-separated list of parameters surrounded by brackets. The body can be a single expression (as in our preceding code) or a statement block. Here is another example:

```
BiFunction<Integer, String, Double> demo =
                   (x,y) -> x * 10d + Double.parseDouble(y);
System.out.println(demo.apply(1, "100"));

//The above is the equivalent to the statement block:
demo = (x,y) -> {
  //You can add here any code you need
  double v = 10d;
  return x * v + Double.parseDouble(y);
};
System.out.println(demo.apply(1, "100"));
```

The result of this example is as follows:

```
110.0
110.0
```

Braces are required only in the case of a statement block. They are optional in a one-line lambda expression, whether the function returns a value or not.

Let's rewrite the code we wrote before using lambda expressions:

```
Function<Integer, Double> multiplyBy10 = i -> i * 10.0;
System.out.println("1 * 10.0 => "+multiplyBy10.apply(1));

Function<Integer, Double> multiplyBy30 = i -> i * 30.0;
System.out.println("1 * 30.0 => "+multiplyBy30.apply(1));

Function<Double,Double> subtract7 = x -> x - 7.0;
System.out.println("10.0 - 7.0 =>"+subtract7.apply(10.0));

Consumer<String> sayHappyToSee =
   s -> System.out.println(s + " Happy to see you again!");
```

```
sayHappyToSee.accept("Hello!");

Predicate<Double> isSmallerThan20 = x -> x < 20d;
System.out.println("10.0 is smaller than 20.0 => " +
                   isSmallerThan20.test(10d));

Predicate<Double> isBiggerThan18 = x -> x > 18d;
System.out.println("10.0 is smaller than 18.0 => " +
                   isBiggerThan18.test(10d));
```

If we run this, we get the following:

```
1 * 10.0 => 10.0
1 * 30.0 => 30.0
10.0 - 7.0 => 3.0
Hello! Happy to see you again!
10.0 is smaller than 20.0 => true
10.0 is smaller than 18.0 => false
```

As you see, the results are exactly the same, but the code is much simpler and captures only the essence.

The factory method can be rewritten and simplified using lambda expressions too:

```
Supplier<String> applyCompareAndSay(int i,
                      Function<Integer, Double> func,
                      Predicate<Double> compare, String message){
  return () -> {
    double v = func.apply(i);
    return (compare.test(v) ? v + " is " : v + " is not ") + message;
  };
}
```

We don't repeat the name of the implemented `Supplier<String>` interface anymore because it is specified as the return type in the method signature. And we do not specify the name of the implemented `test()` method either because it is the only method of the `Supplier` interface that has to be implemented. Writing such a compact and efficient code became possible because of the combination of a lambda expression and functional interface.

As in an anonymous class, the variable created outside and used inside a lambda expression becomes effectively final and cannot be modified. You can write the following code:

```
double v = 10d;
multiplyBy10 = i -> i * v;
```

However, you cannot change the value of the variable `v` outside the lambda expression:

```
double v = 10d;
v = 30d; //Causes compiler error
multiplyBy10 = i -> i * v;
```

You cannot change it inside the expression as well:

```
double v = 10d;
multiplyBy10 = i -> {
  v = 30d; //Causes compiler error
  return i * v;
};
```

The reason for this restriction is that a function can be passed and executed for different arguments in different contexts (different threads, for example), and the attempt to synchronize these contexts would frustrate the original idea of the distributed evaluation of functions.

One principal difference between an anonymous class and lambda expression is the interpretation of the `this` keyword. Inside an anonymous class, it refers to the instance of the anonymous class. Inside the lambda expression, `this` refers to the instance of the class that surrounds the expression. Here is the demo code:

```
public static void main(String arg[]) {
  Demo d = new Demo();
  d.method();
}

public static class Demo{
  private String prop = "DemoProperty";
  public void method(){
    Consumer<String> consumer = s -> {
    System.out.println("Lambda accept(" + s + "): this.prop="
                       + this.prop);
  };
  consumer.accept(this.prop);

  consumer = new Consumer<String>() {
    private String prop = "ConsumerProperty";
    public void accept(String s) {
      System.out.println("Anonymous accept(" + s + "): this.prop="
                         + this.prop);
    }
```

```
    };
    consumer.accept(this.prop);
}
```

The output of this code is as follows:

```
Lambda accept(DemoProperty): this.prop=DemoProperty
Anonymous accept(DemoProperty): this.prop=ConsumerProperty
```

The lambda expression is not an inner class and cannot be referred to by `this`. According to the Java specification, such an approach allows you to have more flexibility of implementation by treating reference `this` as coming from the surrounding context.

There's more...

Look how simpler and less convoluted the demo code becomes. We can create functions on the fly while passing them as parameters:

```
Supplier<String> compare1By30Less7TwiceTo18And20 =
        applyCompareAndSay(1, x -> x * 30.0 - 7.0 - 7.0,
                           x -> x < 20 && x > 18, "betwen 18 and 20");
System.out.println("Compare (1 * 30 - 7 - 7) and the range"
                   + " 18 to 20 => "
                   + compare1By30Less7TwiceTo18And20Lambda.get());
```

However, the result does not change:

```
Compare (1 * 30 - 7 - 7) and the range 18 to 20 => 16.0 is not betwen 18 and 20
```

This is the power and beauty of lambda expressions in combination with functional interfaces.

See also

Refer to the following recipes in this chapter:

- Understanding and creating a functional interface
- Using method references

Also, refer to `Chapter 5`, *Stream Operations and Pipelines*

Using method references

In this recipe, you will learn how to use a method reference, the constructor reference being one of the cases.

Getting ready

In the cases, when a one-line lambda expression consists of a reference to an existing method only (implemented somewhere else), it is possible to further simplify the notation using the method reference. The reference method can be static or non-static (the latter can be bound to a particular object or not) or can be a constructor with or without parameters.

The syntax of the method reference is `Location::methodName`, where `Location` indicates where (in which object or class) the `methodName` method can be found. The two colons (`::`) serve as a separator between the location and the method name. If there are several methods with the same name at the specified location (because of the method overload), the reference method is identified by the signature of the abstract method of the functional interface implemented by the lambda expression.

How to do it...

Using a method reference is straightforward and can be easily illustrated by a few examples for each case:

1. First, we have the static method reference. If a `Food` class has a static method, named `String getFavorite()`, then the lambda expression may look like this:

```
Supplier<String> supplier = Food::getFavorite;
```

   If the `Foo` class has another method, named `String getFavorite(int num)`, the lambda expression that uses it may look exactly the same:

```
Function<Integer, String> func = Food::getFavorite;
```

   The difference is in the interface that this lambda expression implements. It allows the compiler and Java runtime to identify the method to be used. Let's look at the code. Here is the `Food` class:

```
public class Food{
  public static String getFavorite(){ return "Donut!"; }
  public static String getFavorite(int num){
    return num > 1 ? String.valueOf(num)
                     + " donuts!" : "Donut!";
  }
}
```

   We can use its static methods as the implementation of the functional interfaces:

```
Supplier<String> supplier = Food::getFavorite;
System.out.println("supplier.get() => " + supplier.get());

Function<Integer, String> func = Food::getFavorite;
System.out.println("func.getFavorite(1) => "
                   + func.apply(1));
System.out.println("func.getFavorite(2) => "
                   + func.apply(2));
```

   The result is going to be as follows:

```
supplier.get() => Donut!
func.getFavorite(1) => Donut!
func.getFavorite(2) => 2 donuts!
```

2. Second, we have the method reference to a constructor.

Let's assume that the `Food` class does not have an explicit constructor or has one without parameters. The closest to such a signature is a functional interface called `Supplier<Food>` because it does not take any parameter either. Let's add the following to our `Food` class:

```
private String name;
public Food(){ this.name = "Donut"; };
public String sayFavorite(){
  return this.name + (this.name.toLowerCase()
              .contains("donut") ? "? Yes!" : "? D'oh!");
}
```

Then we can write the following:

```
Supplier<Food> constrFood = Food::new;
Food food = constrFood.get();
System.out.println("new Food().sayFavorite() => "
                   + food.sayFavorite());
```

If we write this, we get the following output:

```
new Food().sayFavorite() => Donut? Yes!
```

The preceding non-static method reference was bound to a particular instance of the `Food` class. We will come back to it and also discuss an unbound non-static method reference later. But, for now, we will add another constructor with one parameter to the `Food` class:

```
public Food(String name){
  this.name = name;
}
```

Once we do this, we will express it via the method reference:

```
Function<String, Food> constrFood1 = Food::new;
food = constrFood1.apply("Donuts");
System.out.println("new Food(Donuts).sayFavorite() => "
                   + food.sayFavorite());
```

```
food = constrFood1.apply("Carrot");
System.out.println("new Food(Carrot).sayFavorite() => "
                   + food.sayFavorite());
```

This results in the following code:

```
new Food(Donuts).sayFavorite() => Donuts? Yes!
new Food(Carrot).sayFavorite() => Carrot? D'oh!
```

In the same manner, we can add a constructor with two parameters:

```
public Food(String name, String anotherName) {
  this.name = name + " and " + anotherName;
}
```

Once we do this, we can express it via `BiFunction<String, String>`:

```
BiFunction<String, String, Food> constrFood2 = Food::new;
food = constrFood2.apply("Donuts", "Carrots");
System.out.println("new Food(Donuts,Carrot).sayFavorite() => "
                   + food.sayFavorite());
food = constrFood2.apply("Carrot", "Broccoli");
System.out.println("new Food(Carrot,Broccoli)
                   .sayFavorite() => " + food.sayFavorite());
```

This results in the following:

```
new Food(Donuts,Carrot).sayFavorite() => Donuts and Carrots? Yes!
new Food(Carrot,Broccoli).sayFavorite() => Carrot and Broccoli? D'oh!
```

To express a constructor that accepts more than two parameters, we can create a custom functional interface with any number of parameters. For example, consider the following:

```
@FunctionalInterface
interface Func<T1,T2,T3,R>{ R apply(T1 t1, T2 t2, T3 t3);}
```

We can use it for different types:

```
Func<Integer, Double, String, Food> constr3 = Food::new;
Food food = constr3.apply(1, 2d, "Food");
```

The name of this custom interface and the name of its only method can be anything we like:

```
@FunctionalInterface
interface FourParamFunction<T1,T2,T3,R>{
  R construct(T1 t1, T2 t2, T3 t3);
}
Func<Integer, Double, String, Food> constr3 = Food::new;
Food food = constr3.construct(1, 2d, "Food");
```

3. Third, we have the bound and unbound non-static methods.

The method reference we used for the `sayFavorite()` method requires (bound to) the class instance. This means that we cannot change the instance of the class used in the function after the function is created. To demonstrate this, let's create three instances of the `Food` class and three instances of the `Supplier<String>` interface that capture the functionality of the `sayFavorite()` method:

```
Food food1 = new Food();
Food food2 = new Food("Carrot");
Food food3 = new Food("Carrot", "Broccoli");
Supplier<String> supplier1 = food1::sayFavorite;
Supplier<String> supplier2 = food2::sayFavorite;
Supplier<String> supplier3 = food3::sayFavorite;
System.out.println("new Food()=>supplier1.get() => " +
                   supplier1.get());
System.out.println("new Food(Carrot)=>supplier2.get() => "
                   + supplier2.get());
System.out.println("new Food(Carrot,Broccoli)" +
                   "=>supplier3.get() => " + supplier3.get());
```

As you can see, after the supplier is created, we can only call the `get()` method on it and cannot change the instance (of the `Food` class) to which it was bound (the `get()` method refers to the method of the objects `food1`, `food2`, or `food3`). The results are as follows:

```
new Food()=>supplier1.get() => Donut? Yes!
new Food(Carrot)=>supplier2.get() => Carrot? D'oh!
new Food(Carrot,Broccoli)=>supplier3.get() => Carrot and Broccoli? D'oh!
```

By contrast, we can create an unbound master reference with the instance of the `Function<Food, String>` interface (notice that the method location is specified as a class name called `Food`):

```
Function<Food, String> func = Food::sayFavorite;
```

This means we can use a different instance of the `Food` class for every call of this function:

```
System.out.println("new Food().sayFavorite() => "
                   + func.apply(food1));
System.out.println("new Food(Carrot).sayFavorite() => "
                   + func.apply(food2));
System.out.println("new Food(Carrot,Broccoli).sayFavorite()=> "
                   + func.apply(food3));
```

This is why this method reference is called **unbound**.

Finally, we can overload the `sayFavorite()` method (in the same way we did it for the static method `getFavorite()`) by adding the `sayFavorite(String name)` method:

```
public String sayFavorite(String name){
  this.name = this.name + " and " + name;
  return sayFavorite();
}
```

With this, we can show that the compiler and Java runtime can still understand (using the signature of the specified functional interface) our intent and invoke the correct method:

```
Function<String, String> func1 = food1::sayFavorite;
Function<String, String> func2 = food2::sayFavorite;
Function<String, String> func3 = food3::sayFavorite;
System.out.println("new Food().sayFavorite(Carrot) => "
                   + func1.apply("Carrot"));
System.out.println("new Food(Carrot).sayFavorite(Broccoli) => "
                   + func2.apply("Broccoli"));
System.out.println("new Food(Carrot,Broccoli)" +
                   ".sayFavorite(Donuts) => " +
                   func3.apply("Donuts"));
```

The results are as follows:

```
new Food().sayFavorite(Carrot) => Donut and Carrot? Yes!
new Food(Carrot).sayFavorite(Broccoli) => Carrot and Broccoli? D'oh!
new Food(Carrot,Broccoli).sayFavorite(Donuts) => Carrot and Broccoli and Donuts? Yes!
```

There's more...

There are several simple but useful lambda expressions and method references often used in practice:

```
Function<String, Integer> strLength = String::length;
System.out.println(strLength.apply("3"));

Function<String, Integer> parseInt = Integer::parseInt;
System.out.println(parseInt.apply("3"));

Consumer<String> consumer = System.out::println;
consumer.accept("Hello!");
```

If we run them, the results would be as follows:

```
1
3
Hello!
```

Here are a few useful methods for working with arrays and lists:

```
Function<Integer, String[]> createArray = String[]::new;
String[] arr = createArray.apply(3);
System.out.println("Array length=" + arr.length);
int i = 0;
for(String s: arr){ arr[i++] = String.valueOf(i); }

Function<String[], List<String>> toList = Arrays::<String>asList;
List<String> l = toList.apply(arr);
System.out.println("List size=" + l.size());
for(String s: l){ System.out.println(s); }
```

Here are the results of the preceding code:

```
Array length=3
List size=3
1
2
3
```

We leave it up to you to analyze how they were created and used.

See also

Refer to the following recipes in this chapter:

- Understanding and creating a functional interface
- Understanding lambda expressions

Also, refer to `Chapter 5`, *Stream Operations and Pipelines*

Creating and invoking lambda-friendly APIs

In this recipe, you will learn how to create lambda-friendly APIs and the best practices of doing it.

Getting ready

When an idea of a new API first comes up, it usually looks clean and well focused. Even the first implemented version often preserves the same qualities. But then "one-offs" and other small deviations from the main use cases become pressing, and the API starts to grow (and becomes increasingly complex and more difficult to use) as the variety of use cases increases. Life does not always comply with our vision of it. That's why any API designer at some point faces the question of how generic and flexible the API should be. A too-generic API makes it difficult to understand in terms of the specific business domain, while a very flexible API makes the implementation more complex and difficult to test, maintain, and use.

Using interfaces as parameters facilitate flexibility but require writing new code that implements them. Functional interfaces and lambda expressions allow the decreasing of the scope of such code to a minimum by capturing the functionality with almost no plumbing. The granularity of such an implementation can be as fine or as coarse as needed without the need to create new classes, their object factories, and other traditional infrastructure.

However, it is possible to push the flexibility too far and overuse the power of the new features to the point when it defeats the purpose by making the API difficult to understand and next to impossible to use. To warn about certain pitfalls and share the best practices of a lambda-friendly API design is the purpose of this recipe.

The best practices of a lambda-friendly API design include the following:

- Prefer the interfaces of the `java.util.function` package
- Avoid overloading methods by the type of the functional interface
- Use the `@FunctionalInterface` annotation for custom functional interfaces
- Consider the functional interface as a parameter instead of creating several methods that are different only as per some step of the

functionality

How it works...

We talked about the first two of the listed practices in the *Understanding and creating a functional interface* recipe. The advantage of using the interfaces of the `java.util.function` package is based on two facts. First, any standardization facilitates better understanding and ease of API usage. Second, it facilitates minimal code writing.

To illustrate these considerations, let's try to create an API and call it `GrandApi`--an interface that is going to be implemented in the `GrandApiImpl` class.

Let's add a method to our new API that allows the client to pass a function that calculates something. Such an arrangement allows a client to customize the behavior of the API as needed. This design is called a delegation pattern. It helps with an object composition, which we have discussed in `Chapter 2`, *Fast Track to OOP - Classes and Interfaces*.

First, we demonstrate the traditional object-oriented approach and introduce a `Calculator` interface that has a method that calculates something:

```
public interface Calculator {
  double calculateSomething();
}
public class CalcImpl implements Calculator{
  private int par1;
  private double par2;
  public CalcImpl(int par1, double par2){
    this.par1 = par1;
    this.par2 = par2;
  }
  public double calculateSomething(){
    return par1 * par2;
  }
}
```

So, the first method of our new interface can have the following method:

```
public interface GrandApi{
  double doSomething(Calculator calc, String str, int i);
}
```

The implementation of this method may look like this:

```
double doSomething(Calculator calc, String str, int i){
  return calc.calculateSomething() * i + str.length();
}
```

We can now create a `CalucaltorImpl` class that implements the `Calculator` interface and pass the object of `CalculatorImpl` to the `doSomething()` method:

```
GrandApi api = new GrandImpl();
Calculator calc = new CalcImpl(20, 10d);
double res = api.doSomething(calc, "abc", 2);
System.out.println(res);
```

The result will be as follows:

```
403.0
```

If the client would want to use another `Calculator` implementation, this approach would require you to either create a new class or use an anonymous class:

```
GrandApi api = new GrandImpl();
double res = api.doSomething(new Calculator() {
  public double calculateSomething() {
    return 20 * 10d;
  }
}, "abc", 2);
System.out.println(res);
```

These were the two options available for a client before Java 8. And if the client is forced to use a certain `Calculator` implementation (developed by a third party, for example), the anonymous class could not be used; therefore, only one option would remain.

With Java 8, the client can take advantage of the `Calculator` interface that has one abstract method only (so, it is being a functional interface). Say, a client is forced to use a third-party implementation:

```
public static class AnyImpl{
  public double doIt(){ return 1d; }
  public double doSomethingElse(){ return 100d; }
}
```

The client could write the following:

```
GrandApi api = new GrandImpl();
AnyImpl anyImpl = new AnyImpl();
double res = api.doSomething(anyImpl::doIt, "abc", 2);
System.out.println(res);
```

If they do this, they'll get the following result:

```
5.0
```

The compiler and Java runtime match the functional interface primarily by the number of input parameters and the presence or absence of the return value.

If the usage of a third-party implementation is not mandatory, the client can use a lambda expression, for example, the following:

```
double res = api.doSomething(() -> 20 * 10d, "abc", 2);
System.out.println(res);
```

They can alternatively use this:

```
int i = 20;
double d = 10.0;
double res = api.doSomething(() -> i * d, "abc", 2);
System.out.println(res);
```

In both these cases, they will get this:

```
403.0
```

However, all of these is possible only as long as the `Calculator` interface has only one abstract method. So, we better have the `@FunctionalInterface` annotation added to it, if we can. This is because the client code with lambda will break as soon as `Calculator` gains another abstract method.

However, we can avoid creating a custom interface if we use one of the standard functional interfaces from the `java.util.function` package. The matching one, in this case, would be `Supplier<Double>`, and we can change our first API method to this:

```
public interface GrandApi{
  double doSomething(Supplier<Double> supp, String str, int i);
}
```

Now we are sure that the client code with lambda will never break, and the client code will be much shorter:

```
GrandApi api = new GrandImpl();
Supplier<Double> supp = () -> 20 * 10d;
double res = api.doSomething(supp, "abc", 2);
System.out.println(res);
```

It could also be this:

```
GrandApi api = new GrandImpl();
double res = api.doSomething(() -> 20 * 10d, "abc", 2);
System.out.println(res);
```

In either case, the result will be the same:

```
403.0
```

If the client is forced to use the existing implementation, the code can be as follows:

```
GrandApi api = new GrandImpl();
AnyImpl anyImpl = new AnyImpl();
double res = api.doSomething(anyImpl::doIt, "abc", 2);
System.out.println(res);
```

It will still yield the same result for the `AnyImpl` class:

```
5.0
```

This is why using standard functional interfaces is highly advisable because it allows more flexibility and less client code writing. That said, one should be careful not to get into an overloading of the methods by the type of functional interface only. The problem is that the algorithm that identifies the method by its functional interface parameter often does not have much to work with, especially if the method invocation includes an inline lambda expression only. The algorithm checks the number of input parameters (arity) and the presence or absence (`void`) of the return value. But even this may be

not enough, and the API user may have a serious debugging problem. Let's look at an example and add these two methods to our API:

```
double doSomething2(Function<Integer,Integer> function, double num);
void doSomething2(Consumer<String> consumer, double num);
```

For a human eye, these methods have very different signatures. They are resolved correctly as long as the passed-in instance of a functional interface is explicitly specified:

```
GrandApi api = new GrandImpl();
Consumer<String> consumer = System.out::println;
api.doSomething2(consumer, 2d);
```

The type of the functional interface can be also specified using typecasting:

```
GrandApi api = new GrandImpl();
api.doSomething2((Consumer<String>)System.out::println,2d);
```

Now consider the code that does not specify the type of the passed-in functional interface:

```
GrandApi api = new GrandImpl();
api.doSomething2(System.out::println, 2d);
```

It does not even compile, and gives the following error message:



The reason for the error is that both the interfaces (`Function<Integer, Integer>` and `Consumer<String>`) have the same number of input parameters, while the different return type (`Integer` and `void`) apparently is not enough to resolve the method overload in this case. So, instead of overloading the method using different functional interfaces, use different method names. For example, check out the following:

```
double doSomethingWithFunction(Function<Integer, Integer> function,
                               double num);
void doSomethingWIthConsumer(Consumer<String> consumer, double num);
```

There's more...

The following are the best practices for constructing a lambda expression:

- Keep it stateless
- Avoid a block statement
- Use a method reference
- Rely on effectively final
- Do not work around effectively final
- Avoid specifying the parameter type
- Avoid brackets for a single parameter
- Avoid braces and the return statement

An API designer should keep these guidelines in mind too because the API clients will probably use them.

Keeping lambda expressions stateless means that the result of the function evaluation must depend only on the input parameters, no matter how often the expression is evaluated or which parameters were used for the previous call. For example, this would be a bug-prone code:

```
GrandApi api = new GrandImpl();
int[] arr = new int[1];
arr[0] = 1;
double res = api.doSomething(() -> 20 * 10d + arr[0]++, "abc", 2);
System.out.println(res);
res = api.doSomething(() -> 20 * 10d + arr[0]++, "abc", 2);
System.out.println(res);
res = api.doSomething(() -> 20 * 10d + arr[0]++, "abc", 2);
System.out.println(res);
```

This is because it yields different results every time:

```
405.0
407.0
409.0
```

This example also covers recommendations to rely on effectively final and not to work around it. By specifying `final int[] arr` in the preceding example, one gets an illusion that the code is bulletproof while, in fact, it hides the defect.

Other best practices of a lambda help to keep the code clear to better express its main logic, which otherwise might be lost in long-winded code and a multitude of notations. Just compare the following lines. Here's the first line:

```
double res = api.doSomething2((Integer i) -> { return i * 10; }, 2d);
```

Here's the second line:

```
double res = api.doSomething2(i -> i * 10, 2d);
```

The second line is much cleaner and clearer, especially in the case of several parameters and complex logic in the block statement. We will see examples of block statements and how to avoid them in the next recipe. Any modern editor helps to remove unnecessary notation.

See also

Refer to the following recipe in this chapter:

- Leveraging lambda expressions in your programs

Also, refer to `Chapter 5`, *Stream Operations and Pipelines*

Leveraging lambda expressions in your programs

In this recipe, you will learn how to apply a lambda expression to your code. We will get back to the demo application and modify it by introducing a lambda where it makes sense.

Getting ready

Equipped with functional interfaces, lambda expressions, and the best practices of a lambda-friendly API design, we can substantially improve our speed-calculating application by making its design more flexible and user friendly. Let's set up some background, the motivation, and the infrastructure as close to a real-life problem as possible without making it too complex.

The driverless cars and related problems are on the front pages today, and there is a good reason to believe it is going be this way for quite some time. One of the tasks in this domain is the analysis and modeling of the traffic flow in an urban area based on real data. A lot of such data already exists and will continue to be collected in future. Let's assume that we have access to such a database by date, time, and geographical location. Let's also assume that the traffic data from this database comes in units, each capturing details about one vehicle and its driving conditions:

```
public interface TrafficUnit {
  VehicleType getVehicleType();
  int getHorsePower();
  int getWeightPounds();
  int getPayloadPounds();
  int getPassengersCount();
  double getSpeedLimitMph();
  double getTraction();
  RoadCondition getRoadCondition();
  TireCondition getTireCondition();
  int getTemperature();
}
```

This is where `VehicleType`, `RoadCondition`, and `TireCondition` are `enum` types we have already constructed in the previous chapter:

```
enum VehicleType {
  CAR("Car"), TRUCK("Truck"), CAB_CREW("CabCrew");
  private String type;
  VehicleType(String type){ this.type = type; }
  public String getType(){ return this.type;}
}
enum RoadCondition {
  DRY(1.0),
```

```
  WET(0.2) { public double getTraction() {
    return temperature > 60 ? 0.4 : 0.2; } },
  SNOW(0.04);
  public static int temperature;
  private double traction;
  RoadCondition(double traction){ this.traction = traction; }
  public double getTraction(){return this.traction;}
}
enum TireCondition {
  NEW(1.0), WORN(0.2);
  private double traction;
  TireCondition(double traction){ this.traction = traction; }
  public double getTraction(){ return this.traction;}
}
```

The interface of accessing traffic data may look like this:

```
TrafficUnit getOneUnit(Month month, DayOfWeek dayOfWeek,
                       int hour, String country, String city,
                       String trafficLight);
List<TrafficUnit> generateTraffic(int trafficUnitsNumber,
                  Month month, DayOfWeek dayOfWeek, int hour,
                  String country, String city, String trafficLight);
```

So the possible call could be done as follows:

```
TrafficUnit trafficUnit = FactoryTraffic.getOneUnit(Month.APRIL,
               DayOfWeek.FRIDAY, 17, "USA", "Denver", "Main103S");
```

This is where `17` refers to an hour of the day (5 p.m.), and `Main1035` is a traffic light identification, or the call can request multiple results as follows:

```
List<TrafficUnit> trafficUnits =
    FactoryTrafficModel.generateTraffic(20, Month.APRIL, DayOfWeek.FRIDAY,
                                        17, "USA", "Denver", "Main103S");
```

Where `20` is the number of the requested units of traffic.

As you can see, such a traffic factory provides data about traffic in a particular location at a particular time (between 5 p.m. and 6 p.m. in our example). Each call to the factory yields a different result, while the list of traffic units describes statistically correct data (including the most probable weather conditions) in the specified location.

We will also change the interfaces of `FactoryVehicle` and `FactorySpeedModel` so they could build `Vehicle` and `SpeedModel` based on the `TrafficUnit` interface. The

resulting demo code is as follows:

```
double timeSec = 10.0;
TrafficUnit trafficUnit = FactoryTraffic.getOneUnit(Month.APRIL,
            DayOfWeek.FRIDAY, 17, "USA", "Denver", "Main103S");
Vehicle vehicle = FactoryVehicle.build(trafficUnit);
SpeedModel speedModel =
          FactorySpeedModel.generateSpeedModel(trafficUnit);
vehicle.setSpeedModel(speedModel);
printResult(trafficUnit, timeSec, vehicle.getSpeedMph(timeSec));
```

Where method `printResult()` has the following code:

```
void printResult(TrafficUnit tu, double timeSec,
                                          double speedMph){
   System.out.println("Road " + tu.getRoadCondition() + ", tires "
                      + tu.getTireCondition() + ": "
                      + tu.getVehicleType().getType()
                      + " speedMph (" + timeSec + " sec)="
                      + speedMph + " mph");
}
```

The output of this code may look like this:

```
Road WET, tires NEW: Truck speedMph (10.0 sec)=22.0 mph
```

Since we use the "real" data now, every run of this program produces a different result, based on the statistical properties of the data. In a certain location, a car or dry weather would appear more often at that date and time, while in another location, a truck or snow would be more typical.

In this run, the traffic unit brought a wet road, new tires, and `Truck` with such an engine power and load that in 10 seconds it was able to reach the speed of 22 mph. The formula which we used to calculate the speed (inside an object of `SpeedModel`) is familiar to you:

```
double weightPower = 2.0 * horsePower * 746 * 32.174 / weightPounds;
double speed = Math.round(Math.sqrt(timeSec * weightPower)
                          * 0.68 * traction);
```

Here, the `traction` value comes from `TrafficUnit` (see its interface we just discussed). In the class that implements the `TrafficUnit` interface, the method `getTraction()` looks like the following:

```
public double getTraction() {
  double rt = getRoadCondition().getTraction();
  double tt = getTireCondition().getTraction();
  return rt * tt;
}
```

The methods `getRoadCondition()` and `getTireCondition()` return the elements of the corresponding `enum` types we just described.

Now we are ready to improve our speed-calculating application using the new features of Java we discussed in the previous recipes.

How to do it...

Follow these steps to learn how to use lambda expressions:

1. Let's start building an API. We will call it `Traffic`. Without using functional interfaces, it might look like this:

```
public interface Traffic {
  void speedAfterStart(double timeSec, int trafficUnitsNumber);
}
```

   Its implementation may be as follows:

```
public class TrafficImpl implements Traffic {
  private int hour;
  private Month month;
  private DayOfWeek dayOfWeek;
  private String country, city, trafficLight;
  public TrafficImpl(Month month, DayOfWeek dayOfWeek,
                     int hour, String country, String city,
                     String trafficLight){
    this.month = month;
    this.dayOfWeek = dayOfWeek;
    this.hour = hour;
    this.country = country;
    this.city = city;
    this.trafficLight = trafficLight;
  }
  public void speedAfterStart(double timeSec,
                              int trafficUnitsNumber) {
    List<TrafficUnit> trafficUnits =
        FactoryTraffic.generateTraffic(trafficUnitsNumber,
                       month, dayOfWeek, hour, country,
                       city, trafficLight);
    for(TrafficUnit tu: trafficUnits){
      Vehicle vehicle = FactoryVehicle.build(tu);
      SpeedModel speedModel =
          FactorySpeedModel.generateSpeedModel(tu);
      vehicle.setSpeedModel(speedModel);
      double speed = vehicle.getSpeedMph(timeSec);
      printResult(tu, timeSec, speed);
    }
  }
}
```

2. Now let's write sample code that uses this interface:

```
Traffic api = new TrafficImpl(Month.APRIL, DayOfWeek.FRIDAY,
                              17, "USA", "Denver", "Main103S");
```

```
double timeSec = 10.0;
int trafficUnitsNumber = 10;
api.speedAfterStart(timeSec, trafficUnitsNumber);
```

We get results similar to the following:

```
Road WET, tires WORN: Truck speedMph (10.0 sec)=5.0 mph
Road DRY, tires WORN: Car speedMph (10.0 sec)=33.0 mph
Road DRY, tires WORN: Car speedMph (10.0 sec)=21.0 mph
Road WET, tires NEW: Truck speedMph (10.0 sec)=29.0 mph
Road DRY, tires NEW: Truck speedMph (10.0 sec)=66.0 mph
Road DRY, tires WORN: Car speedMph (10.0 sec)=17.0 mph
Road WET, tires WORN: CabCrew speedMph (10.0 sec)=3.0 mph
Road DRY, tires NEW: CabCrew speedMph (10.0 sec)=83.0 mph
Road DRY, tires NEW: Car speedMph (10.0 sec)=71.0 mph
Road WET, tires WORN: Car speedMph (10.0 sec)=12.0 mph
```

As mentioned before, since we are using real data, the same code does not produce exactly the same result every time. One should not expect to see the speed values as in the preceding screenshot but something that looks very similar instead.

3. Let's use a lambda expression. The preceding API is quite limited. For example, it does not allow you to test different speed calculation formulas without changing `FactorySpeedModel`. Meanwhile, the `SpeedModel` interface has only one abstract method called `getSpeedMph()`:

```
public interface SpeedModel {
  double getSpeedMph(double timeSec, int weightPounds,
                     int horsePower);
}
```

This makes it a functional interface, and we can take advantage of this fact and add another method to our API that is able to accept the `SpeedModel` implementation as a lambda expression:

```
public interface Traffic {
  void speedAfterStart(double timeSec, int trafficUnitsNumber,
                       SpeedModel speedModel);
}
```

The problem though is that the `traction` value does not come as a parameter to the `getSpeedMph()` method, so we cannot implement it as a function before passing it to our API method. Look closer at the speed calculation:

```
double weightPower =
       2.0 * horsePower * 746 * 32.174/weightPounds;
double speed = Math.round(Math.sqrt(timeSec * weightPower)
                          * 0.68 * traction);
```

When you do this, you notice that `traction` acts as a simple multiplier to the value of `speed`, so we can apply it after the speed calculation (and avoid calling `FactorySpeedModel`):

```
public void speedAfterStart(double timeSec,
       int trafficUnitsNumber, SpeedModel speedModel) {
  List<TrafficUnit> trafficUnits =
       FactoryTraffic.generateTraffic(trafficUnitsNumber,
                           month, dayOfWeek, hour, country,
                           city, trafficLight);
  for(TrafficUnit tu: trafficUnits){
    Vehicle vehicle = FactoryVehicle.build(tu);
    vehicle.setSpeedModel(speedModel);
    double speed = vehicle.getSpeedMph(timeSec);
    speed = Math.round(speed * tu.getTraction());
    printResult(tu, timeSec, speed);
  }
}
```

This change allows the API users to pass `SpeedModel` as a function:

```
Traffic api = new TrafficImpl(Month.APRIL, DayOfWeek.FRIDAY,
                              17, "USA", "Denver", "Main103S");
double timeSec = 10.0;
int trafficUnitsNumber = 10;
SpeedModel speedModel = (t, wp, hp) -> {
  double weightPower = 2.0 * hp * 746 * 32.174 / wp;
  return Math.round(Math.sqrt(t * weightPower) * 0.68);
};
api.speedAfterStart(timeSec, trafficUnitsNumber, speedModel);
```

4. The result of this code is the same as `SpeedModel` generated by `FactorySpeedModel`. But now the API users can come up with their own speed-calculating function. For example, they can write the following:

```
Vehicle vehicle = FactoryVehicle.build(trafficUnit);
SpeedModel speedModel = (t, wp, hp) -> {
  return -1.0;
};
vehicle.setSpeedModel(speedModel);
printResult(trafficUnit, timeSec, vehicle.getSpeedMph(timeSec));
```

The result will be as follows:

```
Road DRY, tires NEW: Truck speedMph (10.0 sec)=-1.0 mph
Road DRY, tires NEW: Car speedMph (10.0 sec)=-1.0 mph
Road WET, tires WORN: Truck speedMph (10.0 sec)=0.0 mph
Road WET, tires NEW: CabCrew speedMph (10.0 sec)=0.0 mph
Road WET, tires WORN: Truck speedMph (10.0 sec)=0.0 mph
Road DRY, tires NEW: Truck speedMph (10.0 sec)=-1.0 mph
Road DRY, tires NEW: Car speedMph (10.0 sec)=-1.0 mph
Road WET, tires NEW: Truck speedMph (10.0 sec)=0.0 mph
Road DRY, tires WORN: Truck speedMph (10.0 sec)=0.0 mph
Road DRY, tires NEW: CabCrew speedMph (10.0 sec)=-1.0 mph
```

5. Annotate the `SpeedModel` interface as `@FunctionalInterface`, so everybody who tries to add another method to it would be dutifully warned and would not be able to add another abstract method without removing this annotation and being aware of the risk of breaking the code of the existing clients that have implemented this functional interface already.
6. Improve the API by adding various criteria that slice all of the possible traffic into segments.

   For example, API users might want to analyze only cars, trucks, cars that have an engine bigger than 300 horsepower, trucks with an engine bigger than 400 horsepower, and so on. The traditional way to accomplish this would be by creating methods such as these:

```
void speedAfterStartCarEngine(double timeSec,
                    int trafficUnitsNumber, int horsePower);
void speedAfterStartCarTruckOnly(double timeSec,
                         int trafficUnitsNumber);
void speedAfterStartCarTruckEngine(double timeSec,
                    int trafficUnitsNumber, int carHorsePower,
                    int truckHorsePower);
```

   Instead, we can just add standard functional interfaces to the existing method and let the API user decide which slice of traffic to extract:

```
void speedAfterStart(double timeSec, int trafficUnitsNumber,
                     SpeedModel speedModel,
                     Predicate<TrafficUnit> limitTraffic);
```

   The implementation would look like as follows:

```
public void speedAfterStart(double timeSec,
            int trafficUnitsNumber, SpeedModel speedModel,
```

```
            Predicate<TrafficUnit> limitTraffic) {
  List<TrafficUnit> trafficUnits =
       FactoryTraffic.generateTraffic(trafficUnitsNumber,
                                 month, dayOfWeek, hour, country,
                                 city, trafficLight);
  for(TrafficUnit tu: trafficUnits){
    if(limitTraffic.test(tu){
      Vehicle vehicle = FactoryVehicle.build(tu);
      vehicle.setSpeedModel(speedModel);
      double speed = vehicle.getSpeedMph(timeSec);
      speed = Math.round(speed * tu.getTraction());
      printResult(tu, timeSec, speed);
    }
  }
}
```

The API users can call it, for example, as follows:

```
Predicate<TrafficUnit> limitTraffic = tu ->
          (tu.getHorsePower() < 250 && tu.getVehicleType()
           == VehicleType.CAR) || (tu.getHorsePower() < 400
           && tu.getVehicleType()==VehicleType.TRUCK);

api.speedAfterStart(timeSec, trafficUnitsNumber,
                    speedModel, limitTraffic);
```

The results are now limited to the cars with an engine smaller than 250 hp and trucks with an engine smaller than 400 hp:

```
Road WET, tires WORN: Truck speedMph (10.0 sec)=3.0 mph
Road DRY, tires WORN: Car speedMph (10.0 sec)=25.0 mph
Road DRY, tires NEW: Truck speedMph (10.0 sec)=107.0 mph
Road DRY, tires NEW: Truck speedMph (10.0 sec)=72.0 mph
Road DRY, tires NEW: Car speedMph (10.0 sec)=86.0 mph
```

In fact, an API user can now apply any criteria for limiting the traffic as long as they are applicable to the values in the `TrafficUnit` object. A user can write, for example, the following:

```
Predicate<TrafficUnit> limitTraffic2 =
    tu -> tu.getTemperature() > 65
    && tu.getTireCondition() == TireCondition.NEW
    && tu.getRoadCondition() == RoadCondition.WET;
```

Alternatively, they can write any other combination of limits on the values that come from `TrafficUnit`. If a user decides to remove the limit and analyze all of the traffic, this code will do it too:

```
api.speedAfterStart(timeSec, trafficUnitsNumber,
                    speedModel, tu -> true);
```

7. Allow including the value of the calculated speed in the list of the criteria. One way to do this is to change the implementation this way:

```
public void speedAfterStart(double timeSec,
            int trafficUnitsNumber, SpeedModel speedModel,
            BiPredicate<TrafficUnit, Double> limitSpeed){
  List<TrafficUnit> trafficUnits =
       FactoryTraffic.generateTraffic(trafficUnitsNumber,
                              month, dayOfWeek, hour, country,
                              city, trafficLight);
  for(TrafficUnit tu: trafficUnits){
    Vehicle vehicle = FactoryVehicle.build(tu);
    vehicle.setSpeedModel(speedModel);
    double speed = vehicle.getSpeedMph(timeSec);
    speed = Math.round(speed * tu.getTraction());
    if(limitSpeed.test(tu, speed)){
      printResult(tu, timeSec, speed);
    }
  }
}
```

The API would then look like this:

```
void speedAfterStart(double timeSec, int trafficUnitsNumber,
               SpeedModel speedModel,
               BiPredicate<TrafficUnit, Double> limitSpeed);
```

The client code may be as follows:

```
  BiPredicate<TrafficUnit, Double> limitSpeed = (tu, sp) ->
    (sp > (tu.getSpeedLimitMph() + 8.0)
     && tu.getRoadCondition() == RoadCondition.DRY) ||
    (sp > (tu.getSpeedLimitMph() + 5.0)
     && tu.getRoadCondition() == RoadCondition.WET) ||
    (sp > (tu.getSpeedLimitMph() + 0.0)
     && tu.getRoadCondition() == RoadCondition.SNOW);

  api.speedAfterStart(timeSec, trafficUnitsNumber,
                      speedModel, limitSpeed);
```

This example limits traffic by the speed that exceeds a different cushion in different driving conditions. If needed, it can disregard the speed at all and limit traffic exactly the same way the previous predicate did. The only drawback of this implementation is that it is slightly less efficient because the predicate is applied after the speed calculations. This means that the speed calculation will be done for each generated traffic unit, not to a limited number, as in the previous implementation. If this is a concern, you might leave all the different signatures in the API:

```
public interface Traffic {
  void speedAfterStart(double timeSec, int trafficUnitsNumber);
  void speedAfterStart(double timeSec, int trafficUnitsNumber,
                       SpeedModel speedModel);
  void speedAfterStart(double timeSec, int trafficUnitsNumber,
                SpeedModel speedModel,
                Predicate<TrafficUnit> limitTraffic);
  void speedAfterStart(double timeSec, int trafficUnitsNumber,
                SpeedModel speedModel,
                BiPredicate<TrafficUnit,Double> limitTraffic);
}
```

Once you leave them, let the user decide which of the methods to use, more flexible or more efficient (if the default speed calculation implementation is acceptable).

There's more...

So far, we did not leave the API user a choice of the output format. Currently, it is implemented as the method `printResult()`:

```
void printResult(TrafficUnit tu, double timeSec, double speedMph) {
  System.out.println("Road " + tu.getRoadCondition() + ", tires "
                     + tu.getTireCondition() + ": "
                     + tu.getVehicleType().getType() + " speedMph ("
                     + timeSec + " sec)=" + speedMph + " mph");
}
```

To make it more flexible, we can add another parameter to our API:

```
Traffic api = new TrafficImpl(Month.APRIL, DayOfWeek.FRIDAY, 17,
                              "USA", "Denver", "Main103S");
double timeSec = 10.0;
int trafficUnitsNumber = 10;
BiConsumer<TrafficUnit, Double> output = (tm, sp) ->
  System.out.println("Road " + tm.getRoadCondition() + ", tires "
                     + tm.getTireCondition() + ": "
                     + tm.getVehicleType().getType() + " speedMph ("
                     + timeSec + " sec)=" + sp + " mph");
api.speedAfterStart(timeSec, trafficUnitsNumber, speedModel, output);
```

Notice that we take the `timeSec` value not as one of the function parameters, but from the enclosed scope of the function. We can do this because it remains constant (and can be effectively final) throughout the calculations. In the same manner, we can add any other object to the `output` function--a filename or another output device, for example--thus leaving all the output-related decisions to the API user. To accommodate this new function, the API implementation changes to the following:

```
public void speedAfterStart(double timeSec, int trafficUnitsNumber,
                            SpeedModel speedModel) {
  List<TrafficUnit> trafficUnits = FactoryTraffic
              .generateTraffic(trafficUnitsNumber, month,
                               dayOfWeek, hour, country, city,
                               trafficLight);
  for(TrafficUnit tu: trafficUnits){
    Vehicle vehicle = FactoryVehicle.build(tu);
    vehicle.setSpeedModel(speedModel);
    double speed = vehicle.getSpeedMph(timeSec);
    speed = Math.round(speed * tu.getTraction());
    printResult.accept(tu, speed);
```

```
    }
}
```

It took us a while to come to this point where the power of functional programming starts shining and justifying the effort of learning it. Yet, in conjunction with Reactive Streams, described in the next chapter, this Java addition yields even more power. In the next chapter, the motivation for this enhancement becomes even more apparent and fully appreciated.

See also

Refer to Chapter 5, *Stream Operations and Pipelines*

Stream Operations and Pipelines

In the latest Java releases (8 and 9), the collections API has gotten a major facelift with the introduction of streams and internal iteration by leveraging lambda expressions. This chapter shows you how to leverage streams and chain multiple operations on a collection to create a pipeline. Also, we would like to show you how these operations can be done in parallel. We will cover the following recipes:

- Using the new factory methods to create collection objects
- Creating and operating on streams
- Creating an operation pipeline on streams
- Parallel computations on streams

Introduction

Lambda expressions described and demonstrated in the previous chapter were introduced in Java 8. Together with functional interfaces, they added functional programming capability to Java, allowing the passing of behavior (functions) as parameters to the libraries optimized for the performance of data processing. This way, an application programmer can concentrate on the business aspects of the developed system, leaving performance aspects to the specialists--the authors of the library.

One example of such a library is the `java.util.stream` package, which is going to be the focus of this chapter. This package allows you to have a declarative presentation of the procedures that can be subsequently applied to the data, also in parallel; these procedures are presented as streams, which are objects of the `Stream` interface. For better transition from the traditional collections to streams, two default methods (`stream()` and `parallelStream()`) were added to the `java.util.Collection` interface along with the addition of new factory methods of stream generation to the `Stream` interface.

This approach takes advantage of the power of composition, discussed in one of the previous chapters. Together with other design principles--encapsulation, interface, and polymorphism--it facilitates a highly extensible and flexible design, while lambda expressions allow you to implement it in a concise and succinct manner.

Today, when the machine learning requirements of massive data processing and the fine-tuning of operations have become ubiquitous, these new features reinforce the position of Java among the few modern programming languages of choice.

Using the new factory methods to create collection objects

In this recipe, we will revisit traditional ways of creating collections and introduce the new factory methods, namely `List.of()`, `Set.of()`, `Map.of()`, and `Map.ofEntries()`, that come with Java 9.

Getting ready

Before Java 9, there were several ways of creating collections. Here is the most popular way that was used to create `List`:

```
List<String> list = new ArrayList<>();
list.add("This ");
list.add("is ");
list.add("built ");
list.add("by ");
list.add("list.add()");
//Let us print the created list:
list.forEach(System.out::print);
```

Notice the usage of a default method, `forEach(Consumer)`, added to the `Iterable` interface in Java 8.

If we run the preceding code, we get this:

```
This is built by list.add()
```

The shorter way of doing this would be by starting with an array:

```
Arrays.asList("This ", "is ", "created ", "by ",
              "Arrays.asList()").forEach(System.out::print);
```

The result is as follows:

```
This is created by Arrays.asList()
```

Similarly, while creating `Set`, we can write this:

```
Set<String> set = new HashSet<>();
set.add("This ");
set.add("is ");
set.add("built ");
set.add("by ");
set.add("set.add() ");
//Now print the created set:
set.forEach(System.out::print);
```

Alternatively, we can write the following:

```
new HashSet<>(Arrays.asList("This ", "is ", "created ", "by ",
                            "new HashSet(Arrays.asList()) "))
                            .forEach(System.out::print);
```

Here's an illustration of the results of the last two examples:

```
This set.add() is by built
This is by created new HashSet(Arrays.asList())
```

Notice that unlike `List`, the order of elements in `Set` is not preserved. It depends on the hash code implementation and can change from computer to computer. But the order remains the same between the runs on the same computer (please take note of this last fact because we will come back to it later).

The same structure, that is, the order of elements, applies to `Map` too. This is how we used to create `Map` before Java 9:

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "This ");
map.put(2, "is ");
map.put(3, "built ");
map.put(4, "by ");
map.put(5, "map.put() ");
//Print the results:
map.entrySet().forEach(System.out::print);
```

The output of the preceding code is as follows:

```
1=This 2=is 3=built 4=by 5=map.put()
```

Those who had to create collections often appreciated the JDK enhancement-Proposal 269 *Convenience Factory Methods for Collections* (JEP 269) that stated *Java is often criticized for its verbosity* and its goal of *Provide static factory methods on the collection interfaces that will create compact, unmodifiable collection instances.*

In response to the criticism and the proposal, Java 9 introduced twelve `of()` static factory methods for each of the three interfaces. The following is

the code for `List`:

```
static <E> List<E> of()  //Returns list with zero elements
static <E> List<E> of(E e1) //Returns list with one element
static <E> List<E> of(E e1, E e2)  //etc
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)
```

Twelve similar static methods were added to `Set` and `Map` too. Ten overloaded factory methods with a fixed number of elements were optimized for performance, and as stated in JEP 269, these methods *avoid array allocation, initialization, and garbage collection overhead that is incurred by varargs calls.*

The code of the same examples now becomes much more compact:

```
List.of("This ", "is ", "created ", "by ", "List.of()")
    .forEach(System.out::print);
System.out.println();
Set.of("This ", "is ", "created ", "by ", "Set.of() ")
   .forEach(System.out::print);
System.out.println();
Map.of(1, "This ", 2, "is ", 3, "built ", 4, "by ", 5,"Map.of() ")
   .entrySet().forEach(System.out::print);
```

The `System.out.println()` statement was added to inject a line break between the different types of output:

```
This is created by List.of()
by This Set.of() created is
2=is 3=built 4=by 5=Map.of() 1=This
```

One of the twelve static factory methods added to the `Map` interface was different from the other `of()` methods:

```
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,
                                ? extends V>... entries)
```

Here is an example of its usage:

```
Map.ofEntries(
  entry(1, "This "),
  entry(2, "is "),
  entry(3, "built "),
  entry(4, "by "),
  entry(5, "Map.ofEntries() ")
).entrySet().forEach(System.out::print);
```

Here's its output:

```
3=built 2=is 1=This 5=Map.ofEntries() 4=by
```

So, there is no `Map.of()` factory method for an unlimited number of elements. One has to use `Map.ofEntries()` when creating a map with more than 10 elements.

How to do it...

You may have probably noticed the `Set.of()`, `Map.of()`, and `Map.ofEntries()` methods do not preserve the order of their elements. This is different from the previous (before Java 9) instances of `Set` and `Map`; the order of elements now changes between runs even on the same computer (but does not change during the same run, no matter how many times the collection is iterated). This is an intentional feature intended to help programmers avoid reliance on a certain order because it might produce a defect as the order changes when the code is run on another computer.

Another feature of the collections generated by the new `of()` static methods of the `List`, `Set`, and `Map` interfaces is that these collections are immutable. What does this mean? Consider the following code:

```
List<String> list = List.of("This ", "is ", "immutable");
list.add("Is it?");
```

This means that the preceding code
throws `java.lang.UnsupportedOperationException` at runtime and the following code will throw the same exception too:

```
List<Integer> list = List.of(1,2,3,4,5);
list.set(2, 9);
```

Also, the collections generated by the new `of()` static methods do not allow null elements, and the following code throws
the `java.lang.NullPointerException` exception at runtime too:

```
List<String> list = List.of("This ", "is ", "not ", "created ", null);
```

There's more...

It is not an accident that non-null values and immutability guarantees were added soon after lambda and streams were introduced. As you will see in subsequent recipes, the functional programming and stream pipelines encourage a fluent style of coding (using method chaining, as we did in the case of using the `forEach()` method in the examples of this recipe). This fluent style provides more compact and readable code and the non-null guarantee helps support it by removing the need for checking the `null` value.

The immutability feature, in turn, aligns well with the, effectively, final concept for the variables in the outside context used by lambda expressions. For example, a mutable collection allows you to work around this limitation and the following code:

```
List<Integer> list = Arrays.asList(1,2,3,4,5);
list.set(2, 0);
list.forEach(System.out::print);

list.forEach(i -> {
  int j = list.get(2);
  list.set(2, j + 1);
});
System.out.println();
list.forEach(System.out::print);
```

This code produces the following output:

```
12045
12545
```

This means that it is possible--intentionally or not--to introduce a state in a lambda expression and cause different outcomes of the same function in different contexts. This is especially dangerous in parallel processing because one cannot predict the state of each possible context. This is why immutability of a collection is a helpful addition that helps make the code more robust and reliable.

See also

Refer to the following recipes of this chapter:

- Creating and operating on streams
- Creating an operation pipeline on streams

Creating and operating on streams

In this recipe, we will describe how streams can be created and the operations that can be applied to the elements emitted by the streams.

Getting ready

There are many ways to create a stream. Since Java 8, the `Collection` interface has the `stream()` method that returns a sequential stream with this collection as its source and the `parallelStream()` method that returns a possibly parallel stream with this collection as its source. This means that all the subinterfaces, including `Set` and `List`, also have these methods. Also, eight overloaded `stream()` methods were added to the `Arrays` class that created streams of different types from a corresponding array or subset.

The `Stream` interface has the `of()`, `generate()`, and `iterate()` methods. The specialized interfaces `IntStream`, `DoubleStream`, and `LongStream` have similar methods too, while `IntStream` also has the `range()` and `rangeClosed()` methods; both return `IntStream`.

There are `Files.list()`, `Files.lines()`, `Files.find()`, `BufferedReader.lines()`, and many other methods in the JDK that produce streams.

After a stream is created, various operations can be applied to its elements. A stream itself does not store data. It rather acquires them from the source (and provides or emits them to the operations) as needed. The operations can form a pipeline using the fluent style since many intermediate operations can return a stream too. Such operations are called *intermediate* operations. Examples of intermediate operations are `filter()` (selects only elements matching a criterion), `map()` (transforms elements according to a function), `distinct()` (removes duplicates), `limit()` (limits a stream to the specified number of elements), `sorted()` (transforms an unsorted stream into a sorted one), and other methods of the `Stream` interface that return `Stream` too (except those that create a stream we just mentioned).

The pipeline ends with a **terminal operation**. The processing of the stream elements actually begins only when a terminal operation is being executed. Then, all the intermediate operations (if present) start processing and the stream closes and cannot be reopened as soon as the terminal operation is

finished with the execution. Examples of terminal operations are `forEach()`, `findFirst()`, `reduce()`, `collect()`, `sum()`, `max()`, and other methods of the `Stream` interface that do not return `Stream`. Terminal operations return a result or produce a side effect.

All the `Stream` methods support parallel processing, which is especially helpful in the case of a large amount of data processed on a multicore computer. All the Java Stream API interfaces and classes are in the `java.util.stream` package.

In this recipe, we are going to mostly demonstrate sequential streams, created by the `stream()` method and similar. The processing of parallel streams is not much different than sequential streams. One just has to watch that the processing pipeline does not use a context state that can vary across different processing environments. We will discuss parallel processing in another recipe later in this chapter.

How to do it...

In this section of the recipe, we will present methods of stream creation. As mentioned in the introduction, each class that implements the `Set` interface or the `List` interface has the `stream()` method and the `parallelStream()` method that return an instance of the `Stream` interface. For now, we will only look at sequential streams created by the `stream()` method and get back to parallel streams later.

1. Consider the following examples of stream creation:

```
List.of("This", "is", "created", "by", "List.of().stream()")
    .stream().forEach(System.out::print);
System.out.println();
Set.of("This", "is", "created", "by", "Set.of().stream()")
   .stream().forEach(System.out::print);
System.out.println();
Map.of(1, "This ", 2, "is ", 3, "built ", 4, "by ", 5,
       "Map.of().entrySet().stream()")
   .entrySet().stream().forEach(System.out::print);
```

   We used the fluent style to make the code more compact and interjected `System.out.println()` in order to start a new line in the output.

2. Run the preceding examples and see the result:

```
This is created by List.of().stream()
by created This is Set.of().stream()
4=by 5=Map.of().entrySet().stream()2=is 3=built 1=This
```

   Notice that `List` preserves the order of the elements, while the order of `Set` elements changes at every run. This helps uncover the defects based on the reliance of certain order when this order is not guaranteed.

3. Look at the Javadoc of the `Arrays` class. It has eight overloaded `stream()` methods:

```
static DoubleStream stream(double[] array)
static DoubleStream stream(double[] array, int startInclusive,
                           int endExclusive)
static IntStream stream(int[] array)
static IntStream stream(int[] array, int startInclusive,
                        int endExclusive)
static LongStream stream(long[] array)
static LongStream stream(long[] array, int startInclusive,
                         int endExclusive)
static <T> Stream<T> stream(T[] array)
static <T> Stream<T> stream(T[] array, int startInclusive,
                            int endExclusive)
```

4. Write an example of the usage of the last two methods:

```
String[] array = {"That ", "is ", "an ",
                  "Arrays.stream(array)"};
Arrays.stream(array).forEach(System.out::print);
System.out.println();
String[] array1 = { "That ", "is ", "an ",
                    "Arrays.stream(array,0,2)" };
Arrays.stream(array1, 0, 2).forEach(System.out::print);
```

5. Run it and see the result:

```
That is an Arrays.stream(array)
That is
```

Notice that only the first two elements--with indexes `0` and `1`--were selected to be included in the stream, as the preceding second example intended.

6. Now open the Javadoc of the `Stream` interface and see the factory methods `of()`, `generate()`, and `iterate()`:

```
static <T> Stream<T> of(T t) //Returns Stream of one
static <T> Stream<T> ofNullable(T t)//Returns Stream of one
// element, if non-null, otherwise returns an empty Stream
static <T> Stream<T> of(T... values)
static <T> Stream<T> generate(Supplier<? extends T> s)
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
static <T> Stream<T> iterate(T seed,
  Predicate<? super T> hasNext, UnaryOperator<T> next)
```

The first two methods are simple, so we skip their demo and start with the third method, namely `of()`. It can accept either an array or just comma-delimited elements.

7. Write the example as follows:

```
String[] array = { "That ", "is ", "a ", "Stream.of(array)" };
Stream.of(array).forEach(System.out::print);
System.out.println();
Stream.of( "That ", "is ", "a ", "Stream.of(literals)" )
      .forEach(System.out::print);
```

8. Run it and observe the output:

```
That is a Stream.of(array)
That is a Stream.of(literals)
```

9. Write the examples of the usage of the `generate()` and `iterate()` methods as follows:

```
Stream.generate(() -> "generated ")
      .limit(3).forEach(System.out::print);
System.out.println();
System.out.print("Stream.iterate().limit(10): ");
Stream.iterate(0, i -> i + 1)
      .limit(10).forEach(System.out::print);
System.out.println();
System.out.print("Stream.iterate(Predicate < 10): ");
Stream.iterate(0, i -> i < 10, i -> i + 1)
      .forEach(System.out::print);
```

We had to put a limit on the size of the streams generated by the first two examples. Otherwise, they would be infinite. The third example accepts a predicate that provides the criterion for when the iteration has to stop.

10. Run the examples and observe the results:

```
Stream.generate().limit(3): generated generated generated
Stream.iterate().limit(10): 0123456789
Stream.iterate(Predicate < 10): 0123456789
```

11. Now write examples for the specialized interfaces `IntStream`, `DoubleStream`, and `LongStream`. Their implementations are optimized for performance when you process primitives by avoiding the overhead of boxing and unboxing. In addition, the `IntStream` interface has two more

stream-generating methods, namely `range()` and `rangeClosed()`. Both return `IntStream`:

```
System.out.print("IntStream.range(0,10): ");
IntStream.range(0, 9).forEach(System.out::print);
System.out.println();
System.out.print("IntStream.rangeClosed(0,10): ");
IntStream.rangeClosed(0, 9).forEach(System.out::print);
```

12. Run them and see the output:

```
IntStream.range(0,9): 012345678
IntStream.rangeClosed(0,9): 0123456789
```

As you can guess, the `range()` method produces the sequence of integers by an incremental step of `1`, starting with the left parameter but not including the right parameter. While the `rangeClosed()` method generates a similar sequence, it includes the right parameter.

13. Now let's look at the example of the `Files.list(Path dir)` method, which returns `Stream<Path>` of all the entries of the directory:

```
System.out.println("Files.list(dir): ");
Path dir = FileSystems.getDefault()
          .getPath("src/com/packt/cookbook/ch05_streams/");
try(Stream<Path> stream = Files.list(dir)) {
  stream.forEach(System.out::println);
} catch (Exception ex){ ex.printStackTrace(); }
```

The following excerpt is from the JDK API: *This method must be used within a try-with-resources statement or similar control structure to ensure that the stream's open directory is closed promptly after the stream's operations are completed*. This is what we did; we used a `try`-with-resources statement in this case. Alternatively, we could use a `try...catch...finally` construct and close the stream in the finally block and the result would not change.

14. Run the examples and observe the output:

```
Files.list(dir):
src/com/packt/cookbook/ch05_streams/api
src/com/packt/cookbook/ch05_streams/Chapter05Streams.java
src/com/packt/cookbook/ch05_streams/FactorySpeedModel.java
src/com/packt/cookbook/ch05_streams/FactoryTraffic.java
src/com/packt/cookbook/ch05_streams/FactoryVehicle.java
src/com/packt/cookbook/ch05_streams/Thing.java
```

The need to close a stream explicitly can be confusing because the `Stream` interface extends `AutoCloseable`, and with this, one would expect that a stream will be closed automatically. But that is not the case. Here is what the Javadoc for the `Stream` interface has to say about it: *Streams have a BaseStream.close() method and implement AutoCloseable. Most stream instances do not actually need to be closed after use, as they are backed by collections, arrays, or generating functions, which require no special resource management. Generally, only streams whose source is an I/O channel, such as those returned by Files.lines(Path), will require closing*. This means that a programmer has to know the source of the stream and make sure they close it if the API for the source requires it.

15. Write an example of the `Files.lines()` method's usage:

```
System.out.println("Files.lines().limit(3): ");
String file = "src/com/packt/cookbook/ch05_streams
              /Chapter05Streams.java";
try(Stream<String> stream = Files.lines(Paths.get(file))
                                          .limit(3)){
  stream.forEach(l -> {
    if( l.length() > 0 ) System.out.println("   " + l);
  } );
} catch (Exception ex){ ex.printStackTrace(); }
```

The intent was to read the first three lines of the specified file and print non-empty lines with an indentation of three spaces.

16. Run it and see the result:

```
Files.lines().limit(3):
    package com.packt.cookbook.ch05_streams;
    import com.packt.cookbook.ch05_streams.api.SpeedModel;
```

Write the example of the usage of the `ind()` method:

```
static Stream<Path> find(Path start, int maxDepth,
       BiPredicate<Path, BasicFileAttributes> matcher,
       FileVisitOption... options)
```

17. Similar to the previous case, a stream generated by this method has to be closed explicitly too. The `find()` method walks the file tree rooted at a given starting file and at the requested depth and returns the paths to the files that match the predicate (which includes file attributes). Write the following code now:

```
Path dir = FileSystems.getDefault()
           .getPath("src/com/packt/cookbook/ch05_streams/");
BiPredicate<Path, BasicFileAttributes> select =
  (p, b) -> p.getFileName().toString().contains("Factory");
try(Stream<Path> stream = Files.find(f, 2, select)){
  stream.map(path -> path.getFileName())
        .forEach(System.out::println);
} catch (Exception ex){ ex.printStackTrace(); }
```

18. Run it and you'll get the following output:

```
FactorySpeedModel.java
FactoryTraffic.java
FactoryVehicle.java
```

If necessary, `FileVisitorOption.FOLLOW_LINKS` could be included as the last parameter of `Files.find()` if we need to perform a search that would follow all symbolic links it might encounter.

The requirements for using the `BufferedReader.lines()` method, which returns `Stream<String>` of lines read from a file, is a little bit different. According to Javadoc, *The reader must not be operated on during the execution of the terminal stream operation. Otherwise, the result of the terminal stream operation is undefined*.

There are many other methods in the JDK that produce streams. But they are more specialized, and we will not demonstrate them here because of shortage of space.

How it works...

Throughout the preceding examples, we have demonstrated several stream operations already--methods of the `Stream` interface. We used `forEach()` most often and used `limit()` a few times. The first one is a terminal operation and the second one is an intermediate one. Let's look at other methods of the `Stream` interface now. Here are the intermediate operations, methods that return `Stream` and can be connected in a fluent style:

```
Stream<T> peek(Consumer<T> action)

Stream<T> distinct() //Returns stream of distinct elements
Stream<T> skip(long n)    //Discards the first n elements
Stream<T> limit(long max) //Discards elements after max
Stream<T> filter(Predicate<T> predicate)
Stream<T> dropWhile(Predicate<T> predicate)
Stream<T> takeWhile(Predicate<T> predicate)

Stream<R> map(Function<T, R> mapper)
IntStream mapToInt(ToIntFunction<T> mapper)
LongStream mapToLong(ToLongFunction<T> mapper)
DoubleStream mapToDouble(ToDoubleFunction<T> mapper)

Stream<R> flatMap(Function<T, Stream<R>> mapper)
IntStream flatMapToInt(Function<T, IntStream> mapper)
LongStream flatMapToLong(Function<T, LongStream> mapper)
DoubleStream flatMapToDouble(Function<T, DoubleStream> mapper)

static Stream<T> concat(Stream<T> a, Stream<T> b)

Stream<T> sorted()
Stream<T> sorted(Comparator<T> comparator)
```

The signatures of the preceding methods typically include `"? super T"` for an input parameter and `"? extends R"` for the result (see the Javadoc for the formal definition). We simplified them by removing these notations in order to provide a better overview of the variety and commonality of the methods. To compensate, we would like to recap the meaning of the related generic notations since they are used extensively in the Stream API and might confuse you. Let's look at the formal definition of the `flatMap()` method because it has all of them:

```
<R> Stream<R> flatMap(Function<? super T,
                      ? extends Stream<? extends R>> mapper)
```

The `<R>` interface in front of the method indicates to the compiler that it is a generic method (the one with its own type parameters). Without it, the compiler would be looking for the definition of a class or the `R` interface. The type `T` is not listed in front of the method because it is included in the `Stream<T>` interface definition. The `? super T` notation means that the type `T` is allowed here or in its superclass. The `? extends R` notation means that the type `R` is allowed here or its subclass. The same applies to `? extends Stream...` .

Now let's get back to our (simplified) list of intermediate operations, methods of the `Stream` interface. We have broken them into several groups by similarity. The first group contains only one `peek()` method that allows you to apply the `Consumer` function to each of the stream elements without affecting it whatsoever because the `Consumer` function does not return anything. It is typically used for debugging:

```
int sum = Stream.of( 1,2,3,4,5,6,7,8,9 )
  .filter(i -> i % 2 != 0)
  .peek(i -> System.out.print(i))
  .mapToInt(Integer::intValue)
  .sum();
System.out.println("\nsum = " + sum);
```

The result is as follows:

```
13579
sum = 25
```

In the second group of intermediate operations, the first three are self-explanatory. The `filter()` method is one of the most used operations. It does what its name suggests; it discards the elements that match the criterion passed as the `Predicate` function to the method. We saw an example of its usage in the last snippet of code. The `dropWhile()` method discards the elements as long as the criterion is met (then allows the rest of the stream elements to flow to the next operation). The `takeWhile()` method does the opposite; it allows the elements to flow as long as the criterion is met (then discards the rest of the elements). Here is an example of this:

```
System.out.println("Files.lines().dropWhile().takeWhile(): ");
String file = "src/com/packt/cookbook/ch05_streams
```

```
                /Chapter05Streams.java";
try(Stream<String> stream = Files.lines(Paths.get(file))){
  stream.dropWhile(l -> !l.contains("dropWhile().takeWhile()"))
        .takeWhile(l -> !l.contains("} catc"+"h"))
        .forEach(System.out::println);
} catch (Exception ex){ ex.printStackTrace(); }
```

This code reads the file where this code is stored. It discards all the first lines of the file that do not have the `dropWhile().takeWhile()` substring, then allows all the lines to flow until the `} catch` substring is found. Notice that we had to break this string into `"} catch" + "h"` so that the criterion would not return `true` for this line. The result of this is as follows:

```
Files.lines().dropWhile().takeWhile():
        System.out.println("Files.lines().dropWhile().takeWhile(): ");
        String file = "src/com/packt/cookbook/ch05_streams/Chapter05Streams.java";
        try(Stream<String> stream = Files.lines(Paths.get(file))){
            stream.dropWhile(l -> !l.contains("dropWhile().takeWhile()"))
                  .takeWhile(l -> !l.contains("} catc"+"h"))
                  .forEach(System.out::println);
```

The group of `map()` operations is pretty straightforward too. Such an operation transforms each element of the stream by applying to it a function that was passed in as a parameter. We have already seen an example of the usage of the `mapToInt()` method. Here is another example of the `map()` operation:

```
Stream.of( "That ", "is ", "a ", "Stream.of(literals)" )
      .map(s -> s.contains("i")).forEach(System.out::println);
```

In this example, we transform `String` literals into `boolean`. The result is as follows:

```
false
true
false
true
```

The next group of intermediate operations, called `flatMap()`, provides more complex processing. A `flatMap()` operation applies the passed-in function (that returns a stream) to each of the elements so that the operation could produce a stream composed of the streams extracted from each of the elements. Here's an example of this:

```
Stream.of( "That ", "is ", "a ", "Stream.of(literals)" )
      .filter(s -> s.contains("Th"))
      .flatMap(s -> Pattern.compile("(?!^)").splitAsStream(s))
      .forEach(System.out::print);
```

From the input stream, the preceding code selects only literals that contain `Th` and converts them into a stream of characters, which are then printed out by `forEach()`. The result of this is as follows:

```
T
h
a
t
```

The `concat()` method creates a stream from two input streams so that all the elements of the first stream are followed by all the elements of the second stream. Here's an example of this:

```
Stream.concat(Stream.of(4,5,6), Stream.of(1,2,3))
      .forEach(System.out::print);
```

The result is as follows:

```
456123
```

In case there are more than two stream concatenations, you can write the following:

```
Stream.of(Stream.of(4,5,6), Stream.of(1,2,3), Stream.of(7,8,9))
      .flatMap(Function.identity())
      .forEach(System.out::print);
```

Here, `Function.identity()` is a function that returns its input argument (because we do not need to transform the input streams but just pass them as is to the resulting stream). The result is as follows:

```
456123789
```

The last group of intermediate operations is composed of the `sorted()` methods that sort the stream elements in a natural (without parameters) or specified (according to the passed-in `Comparator`) order. It is a stateful

operation (as well as `distinct()`, `limit()`, and `skip()`) that yields a non-deterministic result in the case of parallel processing; this is, however, the topic of a later recipe.

Now let's look at terminal operations (we simplified their signature too by removing `? super T` and `? extends R`):

```
long count()            //Returns count of elements
Optional<T> max(Comparator<T> comparator) //Max according
                                          // to Comparator
Optional<T> min(Comparator<T> comparator) //Min according
                                          // to Comparator

Optional<T> findAny() //Returns any or empty Optional
Optional<T> findFirst()  //Returns the first element
                         // or empty Optional
boolean allMatch(Predicate<T> predicate) //All elements
                                         // match Predicate?
boolean anyMatch(Predicate<T> predicate)  //Any element
                                         // match Predicate?
boolean noneMatch(Predicate<T> predicate) //No element
                                         // match Predicate?

void forEach(Consumer<T> action) //Apply action to each el
void forEachOrdered(Consumer<T> action)

Optional<T> reduce(BinaryOperator<T> accumulator)
T reduce(T identity, BinaryOperator<T> accumulator)
U reduce(U identity, BiFunction<U,T,U> accumulator,
         BinaryOperator<U> combiner)
R collect(Collector<T,A,R> collector)
R collect(Supplier<R> supplier, BiConsumer<R,T> accumulator,
          BiConsumer<R,R> combiner)
Object[] toArray()
A[] toArray(IntFunction<A[]> generator)
```

The first two groups are self-explanatory, but we need to say a few words about `Optional`. The Javadoc defines it this way: *A container object which may or may not contain a non-null value. If a value is present, isPresent() returns true and get() returns the value*. It allows you to avoid `NullPointerException` or check for `null` (which would break the one-line style of code writing). For the same reason, it has its own methods: `map()`, `filter()`, and `flatMap()`. In addition, Optional has methods that include the `isPresent()` check implicitly:

- `ifPresent(Consumer<T> action)`: This performs the action with the value (if present, or it does nothing).

- `ifPresentOrElse(Consumer<T> action, Runnable emptyAction)`: This performs the given action with the value (if present, or it performs the given empty-based action).
- `or(Supplier<Optional<T>> supplier)`: This returns an `Optional` class describing the value (if present, or it returns an `Optional` class produced by the supplying function).
- `orElse(T other)`: This returns the value (if present, or it returns `other`).
- `orElseGet(Supplier<T> supplier)`: This returns the value (if present, or it returns the result produced by the supplying function).
- `orElseThrow(Supplier<X> exceptionSupplier)`: This returns the value (if present, or it throws an exception produced by the exception-supplying function).

Note that `Optional` is used as a return value in cases when `null` is a possible result. Here is an example of the usage. We reimplemented the stream-concatenating code using the `reduce()` operation that returns `Optional`:

```
Stream.of(Stream.of(4,5,6), Stream.of(1,2,3), Stream.of(7,8,9))
      .reduce(Stream::concat)
      .orElseGet(Stream::empty)
      .forEach(System.out::print);
```

The result is the same as in the previous implementation with a `flatMap()` method:

```
456123789
```

The next group of terminal operations is referred to as `forEach()`. These operations guarantee that the given function will be applied to each element of the stream, but `forEach()` does not say anything about the order, which might be changed for better performance. By contrast, `forEachOrdered()` guarantees not only the processing of all the elements of the stream, but also doing this in the order specified by its source, regardless of whether the stream is sequential or parallel. Here are a couple of examples of this:

```
Stream.of("3","2","1").parallel().forEach(System.out::print);
System.out.println();
Stream.of("3","2","1").parallel().forEachOrdered(System.out::print);
```

The result is as follows:

```
213

321
```

As you can see, in the case of parallel processing, `forEach()` does not guarantee the order, while `forEachOrdered()` does. Here is another example of using both `Optional` and `forEach()`:

```
Stream.of( "That ", "is ", "a ", null, "Stream.of(literals)" )
      .map(Optional::ofNullable)
      .filter(Optional::isPresent)
      .map(Optional::get)
      .map(String::toString)
      .forEach(System.out::print);
```

We could not use `Optional.of()` and used `Optional.ofNullable()` instead because `Optional.of()` would throw `NullPointerException` on `null`. In such a case, `Optional.ofNullable()` just returns `Optional` empty. The result is as follows:

```
That is a Stream.of(literals)
```

Now let's talk about the next group of terminal operations, called `reduce()`. Each of the three overloaded methods returns a single value after processing all the stream elements. Among the most simple examples are finding a sum of the stream elements in case they are numbers, or max, min, and similar. But a more complex result can be constructed too for a stream of objects of any type.

The first method, namely `reduce(BinaryOperator<T> accumulator)`, returns `Optional` because it is the responsibility of the provided accumulator function to calculate the result. And, the authors of the JDK implementation cannot guarantee it will always have some value:

```
int sum = Stream.of(1,2,3).reduce((p,e) -> p + e).orElse(0);
System.out.println("Stream.of(1,2,3).reduce(acc): " +sum);
```

The passed-in function is provided the result of the previous result of the same function (as the first parameter `p`) and the next element of the stream (as the second parameter `e`). For the very first element, `p` gets its value, while `e` is the second element. The result is as follows:

```
Stream.of(1,2,3).reduce(acc): 6
```

To avoid the extra step with `Optional`, the second method returns the value provided as the first parameter `identity` of the type `T` (which is the type of the elements of `Stream<T>`) in case the stream is empty. This parameter has to comply with this requirement (from Javadoc): *for all t, accumulator.apply(identity, t) is equal to t*. In our case, it has to be `0` for it to comply with `0 + e == e`. Here is an example of how to use the second method:

```
int sum = Stream.of(1,2,3).reduce(0, (p,e) -> p + e);
System.out.println("Stream.of(1,2,3).reduce(0,acc): " +sum);
```

The result is the same as with the first `reduce()` method. The third one converts the value of the type `T` into a value of the type `U` with the help of the `BiFunction<U,T,U>` function. Then, the result (of the type `R`) goes through the same logic of processing it as the type `T`, as in the previous method. Here is an example of this:

```
String sum = Stream.of(1,2,3).reduce("", (p,e) -> p + e.toString(),
                                         (x,y) -> x + "," + y);
System.out.println("Stream.of(1,2,3).reduce(,acc,comb): " + sum);
```

One naturally expects to see the result as `1,2,3`. Instead, we see the following:

```
Stream.of(1,2,3).reduce(,acc,comb): 123
```

Make the stream parallel like so:

```
String sum = Stream.of(1,2,3).parallel()
                   .reduce("", (p,e) -> p + e.toString(),
                           (x,y) -> x + "," + y);
System.out.println("Stream.of(1,2,3).reduce(,acc,comb): " + sum);
```

Only if you do this, you will see what's expected:

```
Stream.of(1,2,3).parallel.reduce(,acc,comb): 1,2,3
```

This means that the combiner is called only for parallel processing in order to assemble (combine) the results of different streams (processed in parallel). This is the only deviation we have noticed so far from the declared intent of providing the same behavior for sequential and parallel streams. But there are many ways to accomplish the same result without using this third version of `reduce()`. For example, consider this code:

```
String sum = Stream.of(1,2,3)
                   .map(i -> i.toString() + ",")
                   .reduce("", (p,e) -> p + e);
System.out.println("Stream.of(1,2,3).map.reduce(,acc): "
                   + sum.substring(0, sum.length()-1));
```

It produces the same result, as follows:

```
String sum = Stream.of(1,2,3).parallel()
                   .map(i -> i.toString() + ",")
                   .reduce("", (p,e) -> p + e);
System.out.println("Stream.of(1,2,3).map.reduce(,acc): "
                   + sum.substring(0, sum.length()-1));
```

Here is the result:

```
Stream.of(1,2,3).map.reduce(,acc): 1,2,3
```

The next group of intermediate operations, called `collect()`, consists of two methods. The first one accepts `Collector` as a parameter. It is much more popular than the second one because it is backed up by the `Collectors` class, which provides a wide variety of implementations of the `Collector` interface. We encourage you to go through the Javadoc of the `Collectors` class and see what it offers. Let's discuss a few examples of this. First, we'll create a small `demo` class:

```
public class Thing {
  private int someInt;
  public Thing(int i) { this.someInt = i; }
  public int getSomeInt() { return someInt; }
  public String getSomeStr() {
```

```
    return Integer.toString(someInt); }
}
```

We can use it to demonstrate a few collectors:

```
double aa = Stream.of(1,2,3).map(Thing::new)
                  .collect(Collectors.averagingInt(Thing::getSomeInt));
System.out.println("stream(1,2,3).averagingInt(): " + aa);

String as = Stream.of(1,2,3).map(Thing::new).map(Thing::getSomeStr)
                  .collect(Collectors.joining(","));
System.out.println("stream(1,2,3).joining(,): " + as);

String ss = Stream.of(1,2,3).map(Thing::new).map(Thing::getSomeStr)
                  .collect(Collectors.joining(",", "[", "]"));
System.out.println("stream(1,2,3).joining(,[,]): " + ss);
```

The result will be as follows:

```
stream(1,2,3).averagingInt(): 2.0
stream(1,2,3).joining(,): 1,2,3
stream(1,2,3).joining(,[,]): [1,2,3]
```

The joining collector is a source of joy for any programmer who has ever had to write code that checks whether the added element is the first, the last, or removes the last character (like we did in the last example of the `reduce()` operation). The `joining()` collector does this behind the scenes. All the programmer has to pass are the delimiter, prefix, and suffix.

Most programmers will never need to write a custom collector. But in case there is a need, one can use the second method `collect()` of `Stream` (and provide functions that will compose a collector) or use one of the two static methods `Collector.of()` that generates a collector that can be reused.

If you compare the `reduce()` and `collect()` operations, you'll notice that the primary purpose of `reduce()` is to operate on immutable objects and primitives. The result of `reduce()` is one value that is typically (but not exclusively) of the same type as the elements of the stream. The `collect()`, by contrast, produces the result of a different type wrapped in a mutable container. The most popular usage of `collect()` is centered around producing `List`, `Set`, or `Map` using the corresponding `Collectors.toList()`, `Collectors.toSet()`, or `Collectors.toMap()` collector.

The last group of terminal operations consists of two methods `toArray()`. One of them returns `Object[]`, another one returns an array of the specified type. Let's look at the examples of their usage:

```
Object[] os = Stream.of(1,2,3).toArray();
Arrays.stream(os).forEach(System.out::print);
System.out.println();
String[] sts = Stream.of(1,2,3).map(i -> i.toString())
                    .toArray(String[]::new);
Arrays.stream(sts).forEach(System.out::print);
```

The output of these examples is as follows:



The first example is quite straightforward. It is worth commenting though that we cannot write the following:

```
Stream.of(1,2,3).toArray().forEach(System.out::print);
```

This is because `toArray()` is a terminal operation and the stream is closed automatically after it. That's why we have to open a new stream in the second line.

The second example--with the overloaded `A[] toArray(IntFunction<A[]> generator)` method--is more complicated. The Javadoc says this: *The generator function takes an integer, which is the size of the desired array, and produces an array of the desired size*. This means that the method reference to a `toArray(String[]::new)` constructor in the last example is a shorter version of `toArray(size -> new String[size])`.

There's more...

The `java.util.stream` package also provides specialized interfaces, namely `IntStream`, `DoubleStream`, and `LongStream`, that are optimized for processing streams of values of corresponding primitive types. It is very convenient to use them in the case of reducing operations. For example, they have `max()`, `min()`, `average()`, `sum()`, and many other simplified (tuned for performance) methods that can be called on these streams directly as intermediate and terminal operations.

See also

Refer to the following recipes in this chapter:

- Create operation pipeline on streams
- Parallel computations on streams

Creating an operation pipeline on streams

In this recipe, you will learn how to build a pipeline from `Stream` operations.

Getting ready

In the previous chapter, while creating a lambda-friendly API, we ended up with the following API method:

```
public interface Traffic {
  void speedAfterStart(double timeSec,
    int trafficUnitsNumber, SpeedModel speedModel,
    BiPredicate<TrafficUnit, Double> limitTraffic,
    BiConsumer<TrafficUnit, Double> printResult);
 }
```

The specified number of `TrafficUnit` instances were produced inside the `speedAfterStart()` method. They were limited by the `limitTrafficAndSpeed` function and processed according to the `speedModel` function inside the `speedAfterStart()` method. The results were shown on the device and in the format specified by the `printResults` function.

It is a very flexible design that allows you to have quite a range of experimentation via the modification of functions that are passed to the API. In reality, though, especially during the early stages of data analysis, having an API creates an overhead. If the provided flexibility is not enough, one needs to change it and build and deploy a new version. It takes time and the implementation lacks transparency.

To be fair, the encapsulation of the implementation details was one of the API design requirements. But it works well with the established processes that need to be wrapped as a product for a population of users. The situation radically changes during the research phase. When new algorithms are developed and tested and when the need for processing a large amount of data presents its own challenge, transparency across all the layers of the developed system becomes a foundational requirement. Without it, many of today's successes in the big data analysis world would be impossible.

Streams and the pipelines of stream operations address the problem of transparency and minimize the overhead of writing infrastructural code.

How to do it...

Let's recall how a user called the lambda-friendly API:

```
double timeSec = 10.0;
int trafficUnitsNumber = 10;

SpeedModel speedModel = (t, wp, hp) -> ...;
BiConsumer<TrafficUnit, Double> printResults = (tu, sp) -> ...;
BiPredicate<TrafficUnit, Double> limitSpeed = (tu, sp) -> ...;

Traffic api = new TrafficImpl(Month.APRIL, DayOfWeek.FRIDAY, 17,
                              "USA", "Denver", "Main103S");
api.speedAfterStart(timeSec, trafficUnitsNumber, speedModel,
                    limitSpeed, printResults);
```

As we have already noticed, the freedom of the speed calculation manipulation with such an API (without changing it) extends to the formula of the speed calculation, device and format of the output, and the selection of traffic. This is not bad in our simplistic application, but it might not cover all the possibilities of the model evolution in the case of more complex calculations. But it is a good start that allows us to construct the stream and the pipeline of operations for more transparency and flexibility of experimentation.

Now let's look at the API implementation:

```
double timeSec = 10.0;
int trafficUnitsNumber = 10;

SpeedModel speedModel = (t, wp, hp) -> ...;
BiConsumer<TrafficUnit, Double> printResults = (tu, sp) -> ...;
BiPredicate<TrafficUnit, Double> limitSpeed = (tu, sp) -> ...;
List<TrafficUnit> trafficUnits = FactoryTraffic
     .generateTraffic(trafficUnitsNumber, Month.APRIL,
                      DayOfWeek.FRIDAY, 17, "USA", "Denver",
                      "Main103S");
for(TrafficUnit tu: trafficUnits){
  Vehicle vehicle = FactoryVehicle.build(tu);
  vehicle.setSpeedModel(speedModel);
  double speed = vehicle.getSpeedMph(timeSec);
  speed = Math.round(speed * tu.getTraction());
    if(limitSpeed.test(tu, speed)){
```

```
      printResults.accept(tu, speed);
    }
  }
```

Next, we convert the `for` loop into a stream of traffic units and apply the same functions directly to the elements of the stream. But first, we can request the traffic-generating system to supply us with `Stream` instead of `List` of data. This allows you to avoid the storing of all the data in the memory:

```
Stream<TrafficUnit> stream = FactoryTraffic
       .getTrafficUnitStream(trafficUnitsNumber, Month.APRIL,
                             DayOfWeek.FRIDAY, 17, "USA", "Denver",
                             "Main103S");
```

This allows you to process an endless number of traffic units without storing more than one unit at a time in the memory. In the demo code, we still create `List`, so the streaming does not save us memory. But there are real systems, such as sensors, that can provide streams without storing all of the data in the memory first.

We will also create a convenience method:

```
Stream<TrafficUnit>getTrafficUnitStream(int trafficUnitsNumber){
  return FactoryTraffic
         .getTrafficUnitStream(trafficUnitsNumber,Month.APRIL,
                               DayOfWeek.FRIDAY, 17, "USA", "Denver",
                               "Main103S");
}
```

With this, we will now write the following:

```
getTrafficUnitStream(trafficUnitsNumber).map(tu -> {
  Vehicle vehicle = FactoryVehicle.build(tu);
  vehicle.setSpeedModel(speedModel);
  return vehicle;
})
.map(v -> {
  double speed = v.getSpeedMph(timeSec);
  return Math.round(speed * tu.getTraction());
})
.filter(s -> limitSpeed.test(tu, s))
.forEach(tuw -> printResults.accept(tu, s));
```

We mapped (transform) `TrafficUnit` to `Vehicle`, then mapped `Vehicle` to `speed`, then used the current `TrafficUnit` instance and calculated `speed` to limit the

traffic and print results. If you have this code in a modern editor, you will notice that it does not compile because after the first map, the current `TrafficUnit` element is not accessible anymore; it is replaced by `Vehicle`. This means we need to carry along the original elements and add new values on the way. To accomplish this, we need a container--some kind of a traffic unit wrapper. Let's create one:

```
private static class TrafficUnitWrapper {
  private double speed;
  private Vehicle vehicle;
  private TrafficUnit trafficUnit;
  public TrafficUnitWrapper(TrafficUnit trafficUnit){
    this.trafficUnit = trafficUnit;
  }
  public TrafficUnit getTrafficUnit(){ return this.trafficUnit; }
  public Vehicle getVehicle() { return vehicle; }
  public void setVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
  }
  public double getSpeed() { return speed; }
  public void setSpeed(double speed) { this.speed = speed; }
}
```

Now we can build a pipeline that works:

```
getTrafficUnitStream(trafficUnitsNumber)
   .map(TrafficUnitWrapper::new)
   .map(tuw -> {
  Vehicle vehicle = FactoryVehicle.build(tuw.getTrafficUnit());
  vehicle.setSpeedModel(speedModel);
  tuw.setVehicle(vehicle);
  return tuw;
})
.map(tuw -> {
  double speed = tuw.getVehicle().getSpeedMph(timeSec);
  speed = Math.round(speed * tuw.getTrafficUnit()
                               .getTraction());
  tuw.setSpeed(speed);
  return tuw;
})
.filter(tuw -> limitSpeed.test(tuw.getTrafficUnit(),
                               tuw.getSpeed()))
.forEach(tuw -> printResults.accept(tuw.getTrafficUnit(),
                                    tuw.getSpeed()));
```

The code looks a bit verbose, especially the `Vehicle` and `SpeedModel` setting. We can hide these plumbing details by moving them to the `TrafficUntiWrapper` class:

```
private static class TrafficUnitWrapper {
  private double speed;
  private Vehicle vehicle;
  private TrafficUnit trafficUnit;
  public TrafficUnitWrapper(TrafficUnit trafficUnit){
    this.trafficUnit = trafficUnit;
    this.vehicle = FactoryVehicle.build(trafficUnit);
  }
  public TrafficUnitWrapper setSpeedModel(SpeedModel speedModel) {
    this.vehicle.setSpeedModel(speedModel);
    return this;
  }
  pubic TrafficUnit getTrafficUnit(){ return this.trafficUnit; }
  public Vehicle getVehicle() { return vehicle; }
  public double getSpeed() { return speed; }
  public TrafficUnitWrapper setSpeed(double speed) {
    this.speed = speed;
    return this;
  }
}
```

Notice how we return `this` from the `setSpeedModel()` and `setSpeed()` methods. This allows us to preserve the fluent style. Now the pipeline looks much cleaner:

```
getTrafficUnitStream(trafficUnitsNumber)
   .map(TrafficUnitWrapper::new)
   .map(tuw -> tuw.setSpeedModel(speedModel))
   .map(tuw -> {
  double speed = tuw.getVehicle().getSpeedMph(timeSec);
  speed = Math.round(speed * tuw.getTrafficUnit()
                               .getTraction());
  return tuw.setSpeed(speed);
})
.filter(tuw -> limitSpeed.test(tuw.getTrafficUnit(),
                               tuw.getSpeed()))
 .forEach(tuw -> printResults.accept(tuw.getTrafficUnit(),
                                     tuw.getSpeed()));
```

If there is no need to keep the formula for the speed calculations easily accessible to the modification, we can move it to the `TrafficUnitWrapper` class too by changing the `setSpeed()` method to `calcSpeed()`:

```
public TrafficUnitWrapper calcSpeed(double timeSec) {
  double speed = this.vehicle.getSpeedMph(timeSec);
  this.speed = Math.round(speed * this.trafficUnit
                                    .getTraction());
```

```
  return this;
}
```

So, the pipeline becomes even less verbose:

```
getTrafficUnitStream(trafficUnitsNumber)
   .map(TrafficUnitWrapper::new)
   .map(tuw -> tuw.setSpeedModel(speedModel))
   .map(tuw -> tuw.calcSpeed(timeSec))
   .filter(tuw -> limitSpeed.test(tuw.getTrafficUnit(),
                                  tuw.getSpeed()))
   .forEach(tuw -> printResults.accept(tuw.getTrafficUnit(),
                                       tuw.getSpeed()));
```

Based on this technique, we can now create a method that calculates traffic density--the count of vehicles in each lane of a multilane road for the given speed limit in each of these lanes:

```
Integer[] trafficByLane(Stream<TrafficUnit> stream,
                 int trafficUnitsNumber, double timeSec,
                 SpeedModel speedModel, double[] speedLimitByLane) {
  int lanesCount = speedLimitByLane.length;
  Map<Integer, Integer> trafficByLane = stream
               .limit(trafficUnitsNumber)
               .map(TrafficUnitWrapper::new)
               .map(tuw -> tuw.setSpeedModel(speedModel))
               .map(tuw -> tuw.calcSpeed(timeSec))
               .map(speed -> countByLane(lanesCount,
                                speedLimitByLane, speed))
               .collect(Collectors
                   .groupingBy(CountByLane::getLane, Collectors
                   .summingInt(CountByLane::getCount)));
  for(int i = 1; i <= lanesCount; i++){
    trafficByLane.putIfAbsent(i, 0);
  }
  return trafficByLane.values().toArray(new Integer[lanesCount]);
}
```

It uses two private classes, as follows:

```
private class CountByLane {
  int count, lane;
  private CountByLane(int count, int lane){
    this.count = count;
    this.lane = lane;
  }
  public int getLane() { return lane; }
  public int getCount() { return count; }
}
```

It also uses the following:

```
private static class TrafficUnitWrapper {
  private Vehicle vehicle;
  private TrafficUnit trafficUnit;
  public TrafficUnitWrapper(TrafficUnit trafficUnit){
    this.vehicle = FactoryVehicle.build(trafficUnit);
    this.trafficUnit = trafficUnit;
  }
  public TrafficUnitWrapper setSpeedModel(SpeedModel speedModel) {
    this.vehicle.setSpeedModel(speedModel);
    return this;
  }
  public double calcSpeed(double timeSec) {
    double speed = this.vehicle.getSpeedMph(timeSec);
    return Math.round(speed * this.trafficUnit.getTraction());
  }
}
```

It uses the private method too:

```
private CountByLane countByLane(int lanesNumber, double[] speedLimit,
                                double speed){
  for(int i = 1; i <= lanesNumber; i++){
    if(speed <= speedLimit[i - 1]){
      return new CountByLane(1, i);
    }
  }
  return new CountByLane(1, lanesNumber);
}
```

In Chapter 15, *Testing*, we will discuss this method (of the `TrafficDensity` class) in more detail and revisit this implementation to allow you to have better unit testing.

This is why writing a unit test parallel to the code development brings higher productivity by eliminating the need for changing the code afterward. It also results in more testable (better quality) code.

There's more...

The pipeline allows you to add another filter (or any other operation) easily:

```
Predicate<TrafficUnit> limitTraffic = tu ->
    tu.getVehicleType() == Vehicle.VehicleType.CAR
    || tu.getVehicleType() == Vehicle.VehicleType.TRUCK;

getTrafficUnitStream(trafficUnitsNumber)
   .filter(limitTraffic)
   .map(TrafficUnitWrapper::new)
   .map(tuw -> tuw.setSpeedModel(speedModel))
   .map(tuw -> tuw.calcSpeed(timeSec))
   .filter(tuw -> limitSpeed.test(tuw.getTrafficUnit(),
                                  tuw.getSpeed()))
   .forEach(tuw -> printResults.accept(tuw.getTrafficUnit(),
                                       tuw.getSpeed()));
```

It is especially important when many types of data have to be processed. It's worth mentioning that having a filter before the calculations is the best way to improve performance because it allows you to avoid unnecessary calculations.

Another major advantage of using a stream is that the process can be made parallel without extra coding. All one needs to do is change the first line of the pipeline to `getTrafficUnitStream(trafficUnitsNumber).parallel()` (assuming the source does not generate the parallel stream, which can be identified by the `.isParallel()` operation). We will talk about this in more detail in the next recipe.

See also

Refer to the following recipe in this chapter:

- Parallel computations on streams

Parallel computations on streams

In the previous recipes, we have already demonstrated some of the parallel stream processing techniques. In this recipe, we will discuss this in greater detail and share the best practices and possible problems and how to avoid them.

Getting ready

It is tempting to just set up all the streams parallelly and not think about it again. Unfortunately, parallelism does not always provides an advantage. In fact, it incurs an overhead because of the worker threads' coordination. Besides, some stream sources are sequential in nature and some operations may share the same (synchronized) resource. Even worse, the usage of a stateful operation can lead to an unpredictable result. It is not that you cannot use a stateful operation for a parallel stream; it requires careful planning and clear understanding of the state management.

How to do it...

As mentioned in the previous recipe, a parallel stream can be created by the `parallelStream()` method of a collection or the `parallel()` method applied to a stream. Conversely, the existing parallel stream can be converted into a sequential one by calling the stream using the `sequential()` method.

As the first best practice, use a sequential stream by default and start thinking about the parallel one only if you have to and you can. The have to usually comes up if the performance is not good enough and a large amount of data has to be processed. The possibilities are limited by the nature of the stream source and operations. Some of them cannot be either processed in parallel or produce non-deterministic results. For example, reading from a file is sequential and a file-based stream does not perform better in parallel. Any blocking operation also negates performance improvement in parallel.

One of the areas where sequential and parallel streams are different is ordering. Here is an example of this:

```
List.of("This ", "is ", "created ", "by ",
        "List.of().stream()").stream()
        .forEach(System.out::print);
System.out.println();
List.of("This ", "is ", "created ", "by ",
        "List.of().parallelStream()")
    .parallelStream()
    .forEach(System.out::print);
```

The result is as follows:

```
This is created by List.of().stream()
created This is by List.of().parallelStream()
```

As you can see, `List` preserves the order of the elements but does not keep it in the case of parallel streams.

In the *Creating and operating on streams* recipe, we showed that with the `reduce()` and `collect()` operations, a combiner is called only for a parallel stream. So, it is not needed for a sequential stream, but it must be present while operating on a parallel one. Without it, the results of multiple workers are not correctly aggregated.

We have also demonstrated the stateful operations `sorted()`, `distinct()`, `limit()`, and `skip()` yielding a non-deterministic result in the case of parallel processing. If an order is important, you can rely on the stream method.

If an order is important, we have shown that you can rely on the `forEachOrdered()` stream method. It not only guarantees the processing of all the elements of the stream, but also doing it in the order specified by its source, regardless of whether the stream is sequential or parallel.

Whether a parallel stream is created by the `parallelStream()` method of a collection or by the `parallel()` method applied to a stream, the underlying implementation uses the same `ForkJoin` framework introduced in Java 7. The stream is broken down into segments that are then given to different worker threads for processing. On a computer with only one processor, it does not have an advantage, but on a multicore computer, worker threads can be executed by different processors. Even more, if one worker becomes idle, it can *steal* a part of the job from a busy one. The results are then collected from all the workers and aggregated for the terminal operation completion (that is, when a combiner of a collect operation becomes busy).

Generally speaking, if there is a resource that is not safe for a concurrent access, it is not safe to use it during parallel stream processing. Consider these two examples (`ArrayList` is not known to be threadsafe):

```
List<String> wordsWithI = new ArrayList<>();
Stream.of("That ", "is ", "a ", "Stream.of(literals)")
      .parallel()
      .filter(w -> w.contains("i"))
      .forEach(wordsWithI::add);
System.out.println(wordsWithI);
System.out.println();

wordsWithI = Stream.of("That ", "is ", "a ", "Stream.of(literals)" )
                   .parallel()
                   .filter(w -> w.contains("i"))
```

```
                    .collect(Collectors.toList());
System.out.println(wordsWithI);
```

If run several times, this code may produce the following result:

```
[Stream.of(literals)]
[is , Stream.of(literals)]
```

The `Collectors.toList()` method always generates the same list that consists of `is` and `Stream.of(literals)`, while `forEach()` misses either `is` or `Stream.of(literals)` once in a while.



If possible, try using collectors constructed by the `Collectors` class first and avoid shared resource during parallel computations.

Overall, using stateless functions is your best bet for parallel stream pipelines. If in doubt, test your code, most importantly, run the same test many times to check whether the result is stable.

See also

Refer to the `Chapter 7`, *Concurrent and Multithreaded Programming* for more examples of the usage of classes from the package `java.util.stream`.

Database Programming

This chapter covers both basic and commonly used interactions between a Java application and a **database** (**DB**), right from connecting to the DB and performing CRUD operations to creating transactions, storing procedures, and working with **large objects** (**LOBs**). We will cover the following recipes:

- Connecting to a database using JDBC
- Setting up the tables required for DB interactions
- Performing CRUD operations
- Using prepared statements
- Using transactions
- Working with large objects
- Executing stored procedures

Introduction

It is difficult to imagine a complex software application that does not use some kind of data storage. A structured and accessible data storage is called a database. This is why any modern language implementation includes a framework that allows you to access the DB and **create, read, update, and delete** (**CRUD**) data in it. In Java, it is the **Java Database Connectivity** (**JDBC**) API that provides access to *virtually any data source, from relational databases to spreadsheets and flat files*, according to the Javadoc.

The `java.sql` and `javax.sql` packages that compose the JDBC API are included in the **Java Platform Standard Edition** (**Java SE**). The `java.sql` package provides *the API for accessing and processing data stored in a data source (usually a relational database)*. The `javax.sql` package provides the API for server-side data source access and processing. Specifically, it provides the `DataSource` interface for establishing a connection with a database, connection and statement pooling, distributed transactions, and rowsets. We will discuss each of these features in greater detail in the recipes of this chapter.

However, to actually connect `DataSource` to a physical database, one also needs a database-specific driver (provided by a database vendor, such as MySQL, Oracle, PostgreSQL, or SQL server database, to name a few). These might be written in Java or in a mixture of Java and **Java Native Interface** (**JNI**) native methods. This driver implements the JDBC API.

Working with a database involves eight steps:

1. Installing the database by following the vendor instructions.
2. Adding the dependency on a `.jar` to the application with the database-specific driver.
3. Creating a user, database, and database schema: tables, views, stored procedures, and so on.

4. Connecting to the database from the application.
5. Constructing an SQL statement.
6. Executing the SQL statement.
7. Using the result of the execution.
8. Closing the database connection and other resources.

Steps 1-3 are done only once at the database setup stage before the application is run.

Steps 4-8 are performed by the application repeatedly as needed.

Steps 5-7 can be repeated multiple times with the same database connection.

Connecting to a database using JDBC

In this recipe, you will learn how to connect to a database.

How to do it...

1. Select the database you would like to work with. There are good commercial databases and good open source databases. The only thing we are going to assume is that the database of your choice supports **Structured Query Language** (**SQL**), which is a standardized language that allows you to perform CRUD operations on a database. In our recipes, we will use the standard SQL and avoid constructs and procedures specific to a particular database type.
2. If the database is not installed yet, follow the vendor instructions and install it. Then, download the database driver. The most popular ones are types 4 and 5, written in Java. They are very efficient and talk to the database server through a socket connection. If a `.jar` file with such a driver is placed on the classpath, it is loaded automatically. Type 4 and 5 drivers are database specific because they use database native protocol for accessing the database. We are going to assume that you are using a driver of such a type.

   If your application has to access several types of databases, then you need a driver of type 3. Such a driver can talk to different databases via a middleware application server.

   Drivers of type 1 and 2 are used only when there are no other driver types available for your database.

3. Set the downloaded `.jar` file with the driver on your application's classpath. Now you can create a database and access it from your application.
4. Your database might have a console, a GUI, or some other way to interact with it. Read the instructions and create first a user, that is, `cook`, and then a database, namely `cookbook`.

   For example, here are the commands that do this for PostgreSQL:

```
CREATE USER cook SUPERUSER;
CREATE DATABASE cookbook OWNER cook;
```

We selected the `SUPERUSER` role for our user; however, a good security practice is to assign such a powerful role to an administrator and create another application-specific user who can manage data but cannot change the database structure. It is good practice to create another logical layer, called schema, that can have its own set of users and permissions. This way, several schemas in the same database could be isolated, and each user (one of them is your application) will only have access to certain schemas.

5. Also, at the enterprise level, the common practice is to create synonyms for the database schema so that no application can access the original structure directly. You can also create a password for each user, but, again, for the purpose of this book, this is not needed. So we leave it to the database administrators to establish the rules and guidelines suitable to the particular working conditions of each enterprise.

   Now we connect our application to the database. In our demonstration, we will use, as you may have probably guessed by now, the open source (free) PostgreSQL database.

How it works...

Here is the code fragment that creates connection to our local PostgreSQL database:

```
String URL = "jdbc:postgresql://localhost/cookbook";
Properties prop = new Properties( );
//prop.put( "user", "cook" );
//prop.put( "password", "secretPass123" );
Connection conn = DriverManager.getConnection(URL, prop);
```

The commented lines show how you can set a user and password for your connection. Since, for this demo, we keep the database open and accessible to anyone, we could use an overloaded `DriverManager.getConnection(String url)` method. However, we will show the most general implementation that would allow anyone to read from a property file and pass other useful values (`ssl` as true/false, `autoReconnect` as true/false, `connectTimeout` in seconds, and so on) to the connection-creating method. Many keys for the passed-in properties are the same for all major database types, but some of them are database-specific.

Alternatively, for passing only a user and password, we could use the third overloaded version, namely `DriverManager.getConnection(String url, String user, String password)`. It's worth mentioning that it is good practice to keep the password encrypted. We are not going to show how to do this, but there are plenty of guides available online.

Also, the `getConnection()` method throws `SQLException`, so we need to wrap it in a `try...catch` block.

To hide all of this and other plumbing, it is a good idea to keep the connection-establishing code inside a method:

```
private static Connection getDbConnection(){
  String url = "jdbc:postgresql://localhost/cookbook";
  try {
    return DriverManager.getConnection(url);
  }
  catch(Exception ex) {
```

```
    ex.printStackTrace();
    return null;
  }
}
```

Another way of connecting to a database is to use the `DataSource` interface. Its implementation is typically included in the same `.jar` file as the database driver. In the case of PostgreSQL, there are two classes that implement the `DataSource` interface: `org.postgresql.ds.PGSimpleDataSource` and `org.postgresql.ds.PGPoolingDataSource`. We can use them instead of `DriverManager`. Here is an example of the usage of `PGSimpleDataSource`:

```
private static Connection getDbConnection(){
  PGSimpleDataSource source = new PGSimpleDataSource();
  source.setServerName("localhost");
  source.setDatabaseName("cookbook");
  source.setLoginTimeout(10);
  try {
    return source.getConnection();
  }
  catch(Exception ex) {
    ex.printStackTrace();
    return null;
  }
}
```

And the following is an example of the usage of `PGPoolingDataSource`:

```
private static Connection getDbConnection(){
  PGPoolingDataSource source = new PGPoolingDataSource();
  source.setServerName("localhost");
  source.setDatabaseName("cookbook");
  source.setInitialConnections(3);
  source.setMaxConnections(10);
  source.setLoginTimeout(10);
  try {
    return source.getConnection();
  }
  catch(Exception ex) {
    ex.printStackTrace();
    return null;
  }
}
```

The last version of the `getDbConnection()` method is usually the preferred way of connecting because it allows you to use connection pooling and some other features, in addition to those available when connecting via `DriverManager`.

Whatever version of the `getDbConnection()` implementation you choose, you'll need to use it in all the code examples the same way.

There's more...

It is good practice to think about closing the connection as soon as you create it. The way to do this is using the `try-with-resources` construct, which ensures that the resource is closed at the end of the `try...catch` block:

```
try (Connection conn = getDbConnection()) {
  // code that uses the connection to access the DB
}
catch(Exception ex) {
  ex.printStackTrace();
}
```

Such a construct can be used with any object that implements the `java.lang.AutoCloseable` or `java.io.Closeable` interface.

See also

Refer to the following recipe in this chapter:

- Setting up the tables required for DB interactions

Setting up the tables required for DB interactions

In this recipe, you will learn how to create, change, and delete tables and other logical database constructs that compose a database schema.

Getting ready

The standard SQL statement for table creation looks like this:

```
CREATE TABLE table_name (
  column1_name data_type(size),
  column2_name data_type(size),
  column3_name data_type(size),
  ....
);
```

Here, `table_name` and `column_name` have to be alphanumeric and unique (inside the schema) identifiers. The limitations for the names and possible data types are database-specific. For example, Oracle allows the table name to have 128 characters, while in PostgreSQL, the max length of the table name and column name is 63 characters. There are differences in the data types too, so read the database documentation.

How it works...

Here is an example of a command that creates the `traffic_unit` table in PostgreSQL:

```
CREATE TABLE traffic_unit (
  id SERIAL PRIMARY KEY,
  vehicle_type VARCHAR NOT NULL,
  horse_power integer NOT NULL,
  weight_pounds integer NOT NULL,
  payload_pounds integer NOT NULL,
  passengers_count integer NOT NULL,
  speed_limit_mph double precision NOT NULL,
  traction double precision NOT NULL,
  road_condition VARCHAR NOT NULL,
  tire_condition VARCHAR NOT NULL,
  temperature integer NOT NULL
);
```

Here, we did not set the size of the columns of the type `VARCHAR`, thus allowing those columns to store values of any length. The `integer` type, in this case, allows you to store numbers from -2147483648 to +2147483647. The `NOT NULL` type was added because, by default, the column would be nullable and we wanted to make sure that all the columns would be populated for each record.

We also identified the `id` column as `PRIMARY KEY`, which indicates that the column (or the combination of columns) uniquely identifies the record. For example, if there is a table that contains information about all the people of all the countries, the unique combination would *probably* be their full name, address, and date of birth. Well, it is plausible to imagine that in some household, twins are born and given the same name, so we said probably. If the chance of such an occasion is high, we would need to add another column to the primary key combination, which is the order of birth, with the default value of 1. Here is how we can do this in PostgreSQL:

```
CREATE TABLE person (
  name VARCHAR NOT NULL,
  address VARCHAR NOT NULL,
  dob date NOT NULL,
  order integer DEFAULT 1 NOT NULL,
```

```
  PRIMARY KEY (name,address,dob,order)
);
```

In the case of the `traffic_unit` table, there is no combination of columns that can serve as a primary key. Many cars have the same values. But we need to refer to a `traffic_unit` record so we could know which units have been selected and processed and which were not, for example. This is why, we added an `id` column to populate it with a unique generated number, and we would like the database to generate this primary key automatically.

If you look at the generated table description (`\d traffic_unit`), you will see the `nextval('traffic_unit_id_seq'::regclass)` function assigned to the `id` column. This function generates numbers sequentially, starting with 1. If you need some different behavior, create the sequence number generator manually. Here's an example of how to do this:

```
CREATE SEQUENCE traffic_unit_id_seq
START WITH 1000 INCREMENT BY 1
NO MINVALUE NO MAXVALUE CACHE 10;
ALTER TABLE ONLY traffic_unit ALTER COLUMN id SET DEFAULT
nextval('traffic_unit_id_seq'::regclass);
```

This sequence starts from 1,000 and caches 10 numbers for better performance in case there is a need to generate numbers in quick succession.

According to the code examples shared in the previous chapters, the values of `vehicle_type`, `road_condition`, and `tire_condition` are limited by the `enum` type in the code. That's why when the `traffic_unit` table is populated, we would like to make sure that only the values present in the code in `enum` types are set in the columns. To accomplish this, we'll create a lookup table called `enums` and populate it with the values from our `enum` types:

```
CREATE TABLE enums (
  id integer PRIMARY KEY,
  type VARCHAR NOT NULL,
  value VARCHAR NOT NULL
);

insert into enums (id, type, value) values
(1, 'vehicle', 'car'),
(2, 'vehicle', 'truck'),
(3, 'vehicle', 'crewcab'),
(4, 'road_condition', 'dry'),
```

```
(5, 'road_condition', 'wet'),
(6, 'road_condition', 'snow'),
(7, 'tire_condition', 'new'),
(8, 'tire_condition', 'worn');
```

PostgreSQL has an `enum` data type, but it incurs an overhead if the list of possible values is not fixed and has to be changed over time. We think it is quite possible that the list of values in our application will expand. So, we decided not to use a database `enum` type but create the lookup table ourselves.

Now we can refer to the values of the `enums` table from the `traffic_unit` table using their ID as a foreign key. First, we delete the table:

```
drop table traffic_unit;
```

Then we recreate it with a slightly different SQL command:

```
CREATE TABLE traffic_unit (
  id SERIAL PRIMARY KEY,
  vehicle_type integer REFERENCES enums (id),
  horse_power integer NOT NULL,
  weight_pounds integer NOT NULL,
  payload_pounds integer NOT NULL,
  passengers_count integer NOT NULL,
  speed_limit_mph double precision NOT NULL,
  traction double precision NOT NULL,
  road_condition integer REFERENCES enums (id),
  tire_condition integer REFERENCES enums (id),
  temperature integer NOT NULL
);
```

The columns `vehicle_type`, `road_condition`, and `tire_condition` must now be populated by the primary key of the corresponding record of the `enums` table. This way, we can make sure that our traffic-analyzing code will be able to match the values in these columns to the values of the `enum` types in the code.

There's more...

The `enums` table should not have a duplicate combination of type-value because this might lead to confusion, especially when the code that populates the `traffic_unit` table looks up the necessary `id` in the `enums` table. Instead of the one value that is expected, the query will return two. Which one to pick, then? To avoid duplication, we can add a unique constraint to the `enums` table:

```
ALTER TABLE enums ADD CONSTRAINT enums_unique_type_value
UNIQUE (type, value);
```

Now if we try to add a duplicate, the database will not allow it.

Another important consideration of database table creation is whether an index has to be added. An index is a data structure that helps to accelerate data searches in the table without having to check every table record. It can include one or more columns of a table. For example, an index for a primary key is created automatically. If you bring up the description of the table we have created already, you will see the following:

```
 Indexes: "traffic_unit_pkey" PRIMARY KEY, btree (id)
```

We can also add an index ourselves if we think (and have proven by experimentation) it will help the application performance. In the case of `traffic_unit`, we discovered that our code often searches this table by `vehicle_type` and `passengers_count`. So we measured the performance of our code during the search and added these two columns to the index:

```
CREATE INDEX idx_traffic_unit_vehicle_type_passengers_count
ON traffic_unit USING btree (vehicle_type,passengers_count);
```

Then we measured the performance again. If it had improved, we would have left the index in place, but in our case, we removed it:

```
drop index idx_traffic_unit_vehicle_type_passengers_count;
```

We did this because an index has an overhead of additional writes and storage space.

In our examples of primary key, constraints, and indexes, we followed the naming convention of PostgreSQL. If you use a different database, we suggest you look up its naming convention and follow it, so that your naming aligns with the names created automatically.

See also

Refer to the following recipes in this chapter:

- Performing CRUD operations
- Working with large objects

Performing CRUD operations

In this recipe, you will learn how to populate, read, change, and delete data in the database.

Getting ready

We have already seen examples of SQL statements that create (populate) data in the database:

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

We've also seen examples of instances where several table records have to be added:

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3, ... ),
       (value21,value22,value23, ...),
       ( ...                       );
```

If a column has a default value specified, there is no need to list it in the `INSERT INTO` statement, unless a different value has to be inserted.

The reading of the data is done by a `SELECT` statement:

```
SELECT column_name,column_name
FROM table_name WHERE some_column=some_value;
```

This is also done when all the columns have to be selected in an order:

```
SELECT * FROM table_name WHERE some_column=some_value;
```

Here's a general definition of the `WHERE` clause:

```
WHERE column_name operator value
Operator:
  = Equal
  <> Not equal. In some versions of SQL, !=
  > Greater than
  < Less than
  >= Greater than or equal
  <= Less than or equal
  BETWEEN Between an inclusive range
  LIKE Search for a pattern
  IN To specify multiple possible values for a column
```

The `column_name operator value` construct can be combined the with logical operators `AND` and `OR` and grouped with the brackets `(` and `)`.

The data can be changed with the `UPDATE` statement:

```
UPDATE table_name SET column1=value1,column2=value2,...
WHERE-clause;
```

Alternatively, it can be deleted with the `DELETE` statement:

```
DELETE FROM table_name WHERE-clause;
```

Without the `WHERE` clause, all the records of the table are going to be affected by the `UPDATE` or `DELETE` statement.

How to do it...

We have already seen an `INSERT` statement. Here is an example of other types of statements:

```
cookbook=# select * from enums;
 id |      type      |  value
----+----------------+---------
  1 | vehicle        | car
  2 | vehicle        | truck
  3 | vehicle        | crewcab
  4 | road_condition | dry
  5 | road_condition | wet
  6 | road_condition | snow
  7 | tire_condition | new
  8 | tire_condition | worn
(8 rows)
```

The preceding `SELECT` statement requires bringing up all the columns of all the rows of the table. If the number of rows was bigger than the number of lines on the screen, the database console would show only the first screen and you would need to type a command (database-specific) to show the next screen:

```
cookbook=# select * from enums where (type = 'vehicle' AND value != 'crewcab') OR value = 'new';
 id |      type      | value
----+----------------+-------
  1 | vehicle        | car
  2 | vehicle        | truck
  7 | tire_condition | new
(3 rows)
```

This `SELECT` statement has a `WHERE` clause that requires you to show only those rows where the value in the `type` column is `vehicle` and the value in the `value` column is not `crewcab`. It also requires you to show the rows where the value in the `value` column is `new`:

```
cookbook=# update enums set value = 'NEW' where value = 'new';
UPDATE 1
cookbook=# delete from enums where value != 'NEW';
DELETE 7
cookbook=# select * from enums;
 id |      type      | value
----+----------------+-------
  7 | tire_condition | NEW
(1 row)
```

The preceding screenshot captures three statements. The first one is an `UPDATE` statement that requires you to change the value in the `value` column to `NEW`, but only in the rows where the value in the `value` column is `new` (apparently, the value is case-sensitive). The second statement requires you to delete all the rows that do not have the value `NEW` in the `value` column. The third statement is `SELECT`, which we just discussed.

It worth noting that we would not be able to delete the records of the `enums` table if these records were referred to as foreign keys in the `traffic_unit` table. Only after deleting the corresponding records of the `traffic_unit` table would we be able to do this. But, for now, that is, for demonstration purposes, we keep the `traffic_unit` table empty.

To execute any of the CRUD operations in the code, one has to acquire a JDBC connection first, then create and execute a statement:

```
try (Connection conn = getDbConnection()) {
  try (Statement st = conn.createStatement()) {
    boolean res = st.execute("select id, type, value from enums");
    if (res) {
      ResultSet rs = st.getResultSet();
      while (rs.next()) {
        int id = rs.getInt(1);
        String type = rs.getString(2);
        String value = rs.getString(3);
        System.out.println("id = " + id + ", type = "
                           + type + ", value = " + value);
      }
    } else {
      int count = st.getUpdateCount();
      System.out.println("Update count = " + count);
    }
  }
} catch (Exception ex) { ex.printStackTrace(); }
```

It is good practice to use the `try-with-resources` construct for the `Statement` object too. The closing of the `Connection` object would close the `Statement` object automatically. However, when you close the `Statement` object explicitly, the cleanup happens immediately instead of waiting for the necessary checks and actions to propagate through the layers of the framework.

The `execute()` method is the most generic one among the three methods that can execute a statement. The other two include `executeQuery()` (for `SELECT`

statements only) and `executeUpdate()` (for `UPDATE`, `DELETE`, `CREATE`, or `ALTER` statements). As you can see in our example, the `execute()` method returns `boolean`, which indicates whether the result is a `ResultSet` object or just a count. This means that `execute()` acts as `executeQuery()` for the `SELECT` statement and `executeUpdate()` for the other statements that we just listed.

We can demonstrate this by running the preceding code with the following sequence of statements:

```
"select id, type, value from enums"
"insert into enums (id, type, value)" + " values(1,'vehicle','car')"
"select id, type, value from enums"
"update enums set value = 'bus' where value = 'car'"
"select id, type, value from enums"
"delete from enums where value = 'bus'"
"select id, type, value from enums"
```

The result will be as follows:

```
id = 7, type = tire_condition, value = NEW

Update count = 1

id = 7, type = tire_condition, value = NEW
id = 1, type = vehicle, value = car

Update count = 1

id = 7, type = tire_condition, value = NEW
id = 1, type = vehicle, value = bus

Update count = 1

id = 7, type = tire_condition, value = NEW
```

We carried out the positional extraction of the values from `ResultSet` because this is more efficient than using the column name (as in `rs.getInt("id")` or `rs.getInt("type")`). The difference in performance is very small, though, and becomes important only when the operation happens many times. Only the actual measuring and testing can tell you whether the difference in the case of your application is significant or not. Bear in mind that getting values by name provides better code readability, which pays well in the long term during application maintenance.

We used the `execute()` method for demonstration purposes. In practice, the `executeQuery()` method is used for `SELECT` statements because the programmer usually has to extract the data in a way specific to the executed

SQL statement. By contrast, the call to `executeUpdate()` can be wrapped in a generic method:

```
private static void executeUpdate(String sql){
  try (Connection conn = getDbConnection()) {
    try (Statement st = conn.createStatement()) {
      int count = st.executeUpdate(sql);
      System.out.println("Update count = " + count);
    }
  } catch (Exception ex) { ex.printStackTrace(); }
}
```

There's more...

SQL is a rich language, and we do not have enough space to cover all its features. We would just like to enumerate a few of its most popular ones so you are aware of their existence and could look them up when needed:

- The `SELECT` statement allows the use of the `DISTINCT` keyword to get rid off all the duplicate values
- Adding the `ORDER BY` keyword presents the result in the specified order
- The keyword `LIKE` allows you to set the search pattern to the `WHERE` clause
- The search pattern can use several wildcard: `%`, `_`, `[charlist]`, `[^charlist]`, or `[!charlist]`
- Matching values can be enumerated with the `IN` keyword
- The `SELECT` statement can include several tables using the `JOIN` clause
- `SELECT * INTO table_2 from table_1 creates table_2` and copies data from `table_1`
- `TRUNCATE` is faster and uses fewer resources when removing all the rows of a table

There are many other useful methods in the `ResultSet` interface as well. Here is an example of how some of its methods can be used to write generic code that would traverse the returned result and use metadata to print out the column name and the returned value:

```
private static void traverseRS(String sql){
  System.out.println("traverseRS(" + sql + "):");
  try (Connection conn = getDbConnection()) {
    try (Statement st = conn.createStatement()) {
      try(ResultSet rs = st.executeQuery(sql)){
        int cCount = 0;
        Map<Integer, String> cName = new HashMap<>();
        while (rs.next()) {
          if (cCount == 0) {
            ResultSetMetaData rsmd = rs.getMetaData();
            cCount = rsmd.getColumnCount();
            for (int i = 1; i <= cCount; i++) {
              cName.put(i, rsmd.getColumnLabel(i));
            }
          }
```

```
          List<String> l = new ArrayList<>();
          for (int i = 1; i <= cCount; i++) {
            l.add(cName.get(i) + " = " + rs.getString(i));
          }
          System.out.println(l.stream()
                              .collect(Collectors.joining(", ")));
        }
      }
    }
  } catch (Exception ex) { ex.printStackTrace(); }
}
```

We used `ResultSetMetaData` only once to collect the returned column names and the length (number of columns) of one row. Then, we extracted the values from each row by position and created `List<String>` elements with the corresponding column names. To print, we used something we are already familiar with--a programmer's delight--the joining collector (we discussed this in a previous chapter). If we call the `traverseRS("select * from enums")` method, the result will be as follows:

```
id = 7, type = tire_condition, value = NEW
```

See also

Refer to the following recipes in this chapter:

- Using prepared statements
- Using transactions
- Working with large objects
- Executing stored procedures

Using prepared statements

In this recipe, you will learn how to use a prepared statement: a statement template that can be stored in the database and executed efficiently with different input values.

Getting ready

An object of `PreparedStatement`--a subinterface of `Statement`--can be precompiled and stored in the database and then used to efficiently execute the SQL statement multiple times for different input values. Similar to an object of `Statement` (created by the `createStatement()` method), it can be created by the `prepareStatement()` method of the same `Connection` object.

There is also a third version of a statement that creates a method called `prepareCall()` that, in turn, creates the `CallableStatement` object used to execute a database-stored procedure, but we will discuss this in a separate recipe later.

The same SQL statement that was used to generate `Statement` can be used to generate `PreparedStatement` too. In fact, it is a good idea to consider using `PrepdaredStatement` for any SQL statement that is called multiple times because it performs better than `Statement`. To do this, all we need to change are these two lines in the sample code of the previous section:

```
try (Statement st = conn.createStatement()) {
  boolean res = st.execute("select * from enums");
```

We change these lines to the following:

```
try (PreparedStatement st =
          conn.prepareStatement("select * from enums")) {
  boolean res = st.execute();
```

How to do it...

The true usefulness of `PreparedStatement` shines because of its ability to accept parameters--the input values that substitute (in the order of appearance) the `?` symbol. Here's an example of this:

```
traverseRS("select * from enums");
System.out.println();
try (Connection conn = getDbConnection()) {
  String[][] values = {{"1", "vehicle", "car"},
                       {"2", "vehicle", "truck"}};
  String sql = "insert into enums (id, type, value) values(?, ?, ?)");
  try (PreparedStatement st = conn.prepareStatement(sql) {
    for(String[] v: values){
      st.setInt(1, Integer.parseInt(v[0]));
      st.setString(2, v[1]);
      st.setString(3, v[2]);
      int count = st.executeUpdate();
      System.out.println("Update count = " + count);
    }
  }
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from enums");
```

The result of this is as follows:

```
id = 7, type = tire_condition, value = NEW

Update count = 1
Update count = 1

id = 7, type = tire_condition, value = NEW
id = 1, type = vehicle, value = car
id = 2, type = vehicle, value = truck
```

There's more...

It is not a bad idea to always use prepared statements for CRUD operations. They might be slower if executed only once, but you can test and see whether this is the price you are willing to pay. What you get with prepared statements is consistent (better readable) code, more security (prepared statements are not vulnerable to SQL injection), and one fewer decision to make (just reuse the same code everywhere).

See also

Refer to the following recipes in this chapter:

- Using transactions
- Working with large objects
- Executing stored procedures

Using transactions

In this recipe, you will learn what a database transaction is and how it can be used in Java code.

Getting ready

A transaction is a unit of work that includes one or many operations that change data. If successful, all the data changes are *committed* (applied to the database). If one of the operations errors out or the transaction is *rolled back*, then none of the changes included in the transaction will be applied to the database.

Transaction properties are set up on the `Connection` object. They can be changed without closing the connection, so different transactions can reuse the same `Connection` object.

JDBC allows transaction control only for CRUD operations. Table modification (`CREATE TABLE`, `ALTER TABLE`, and so on) is committed automatically and cannot be controlled from the Java code.

By default, a CRUD operation transaction is set to be autocommitted too. This means that every data change that was introduced by an SQL statement is applied to the database as soon as the execution of the SQL statement is completed. All the preceding examples use this default behavior.

To change this, one has to use the `setAutoCommit()` method of the `Connection` object. If set to false, that is, `setAutoCommit(false)`, the data changes will not be applied to the database until the `commit()` method on the `Connection` object is invoked. If the `rollback()` method is called, all the data changes since the beginning of the transaction or since the last call to `commit()` would be discarded.

Explicit programmatic transaction management improves performance, but it is insignificant in the case of short atomic operations that are called once and not very often. Taking over transaction control becomes crucial when several operations introduce changes that have to be applied, either all together or none of them. It allows you to group database changes into atomic units and thus avoid accidental violation of data integrity.

How to do it...

First, let's add an output to the `traverseRS()` method:

```
private static void traverseRS(String sql){
  System.out.println("traverseRS(" + sql + "):");
  try (Connection conn = getDbConnection()) {
    ...
  }
}
```

This will help you analyze the output when many different SQL statements are executed in the same demo example.

Now let's run the following code that reads data from the `enums` table, then inserts a row, and then reads all the data from the table again:

```
traverseRS("select * from enums");
System.out.println();
try (Connection conn = getDbConnection()) {
  conn.setAutoCommit(false);
  String sql = "insert into enums (id, type, value) "
                       + " values(1,'vehicle','car')";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    System.out.println(sql);
    System.out.println("Update count = " + st.executeUpdate());
  }
  //conn.commit();
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from enums");
```

Note that we took over transaction control by calling `conn.setAutoCommit(false)`. The result is as follows:

```
traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW

insert into enums (id, type, value) values(1,'vehicle','car')
Update count = 1

traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW
```

As you can see, the changes were not applied because the call to `commit()` was commented out. When we uncomment it, the result is as

follows:

```
traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW

insert into enums (id, type, value) values(1,'vehicle','car')
Update count = 1

traverseRS(select * from enums):
id = 1, type = vehicle, value = car
id = 7, type = tire_condition, value = NEW
```

Now let's execute two inserts, but introduce a spelling error in the second insert:

```
traverseRS("select * from enums");
System.out.println();
try (Connection conn = getDbConnection()) {
  conn.setAutoCommit(false);
  String sql = "insert into enums (id, type, value) "
                       + " values(1,'vehicle','car')";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    System.out.println(sql);
    System.out.println("Update count = " + st.executeUpdate());
  }
  conn.commit();
  sql = "insert into enums (id, type, value) "
                     + " values(2,'vehicle','truck')";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    System.out.println(sql);
    System.out.println("Update count = " + st.executeUpdate());
  }
  conn.commit();
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from enums");
```

We get a stack trace (we do not show it to save space) of the error:

```
org.postgresql.util.PSQLException: ERROR: syntax error at or near "inst"
```

The result is as follows:

```
traverseRS(select * from enums):
id = 1, type = vehicle, value = car
id = 7, type = tire_condition, value = NEW
```

The second row was not inserted. If there was no `conn.commit()` after the first `INSERT INTO` statement, the first insert would not be applied either. This is the advantage of the programmatic transaction control in the case of many

independent data changes: if one fails, we can skip it and continue applying other changes.

Now let's try to insert three rows with an error (by not setting a number as the `id` value) in the second row:

```
traverseRS("select * from enums");
System.out.println();
try (Connection conn = getDbConnection()) {
  conn.setAutoCommit(false);
  String[][] values = { {"1", "vehicle", "car"},
                        {"b", "vehicle", "truck"},
                        {"3", "vehicle", "crewcab"} };
  String sql = "insert into enums (id, type, value) "
                            + " values(?, ?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    for (String[] v: values){
      try {
        System.out.print("id=" + v[0] + ": ");
        st.setInt(1, Integer.parseInt(v[0]));
        st.setString(2, v[1]);
        st.setString(3, v[2]);
        int count = st.executeUpdate();
        conn.commit();
        System.out.println("Update count = "+count);
      } catch(Exception ex){
        //conn.rollback();
        System.out.println(ex.getMessage());
      }
    }
  }
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from enums");
```

We put each insert execution in the `try...catch` block and commit the changes before printing out the result (update count or error message). The result will be as follows:

```
traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW

id=1: Update count = 1
id=b: For input string: "b"
id=3: Update count = 1

traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW
id = 1, type = vehicle, value = car
id = 3, type = vehicle, value = crewcab
```

You can see that the second row was not inserted, although `conn.rollback()` was commented out. Why? This is because the only SQL statement included in this transaction failed, so there was nothing to roll back.

Now let's create a `test` table using the database console:

```
cookbook=# create table test (name varchar not null);
CREATE TABLE
cookbook=# select * from test;
 name
------
(0 rows)
```

We will use this table to record the vehicle types of the records inserted in the `enums` table:

```
traverseRS("select * from enums");
System.out.println();
try (Connection conn = getDbConnection()) {
  conn.setAutoCommit(false);
  String[][] values = { {"1", "vehicle", "car"},
                        {"b", "vehicle", "truck"},
                        {"3", "vehicle", "crewcab"} };
  String sql = "insert into enums (id, type, value) " +
                                      " values(?, ?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    for (String[] v: values){
      try(Statement stm = conn.createStatement()) {
        System.out.print("id=" + v[0] + ": ");
        stm.execute("insert into test values('"+ v[2] + "')");
        st.setInt(1, Integer.parseInt(v[0]));
        st.setString(2, v[1]);
        st.setString(3, v[2]);
        int count = st.executeUpdate();
        conn.commit();
        System.out.println("Update count = " + count);
      } catch(Exception ex){
         //conn.rollback();
         System.out.println(ex.getMessage());
      }
    }
  }
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from enums");
System.out.println();
traverseRS("select * from test");
```

With `conn.rollback()` commented out, the result will be as follows:

```
traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW

id=1: Update count = 1
id=b: For input string: "b"
id=3: Update count = 1

traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW
id = 1, type = vehicle, value = car
id = 3, type = vehicle, value = crewcab

traverseRS(select * from test):
name = car
name = truck
name = crewcab
```

The row with `truck` was not inserted in the `enums` table but added to the `test` table, although our intent was to record all the vehicles inserted in `enums`, and only them, in the `test` table. This is when the usefulness of a rollback can be demonstrated. If we uncomment `conn.rollback()`, the result will be as follows:

```
traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW

id=1: Update count = 1
id=b: For input string: "b"
id=3: Update count = 1

traverseRS(select * from enums):
id = 7, type = tire_condition, value = NEW
id = 1, type = vehicle, value = car
id = 3, type = vehicle, value = crewcab

traverseRS(select * from test):
name = car
name = crewcab
```

There's more...

Another important property of a transaction is the *transaction isolation level*. It defines the boundaries between database users. For example, can other users see your database changes before they are committed? The higher the isolation (the highest is *serializable*), the more the time it takes a transaction to complete in the case of concurrent access to the same records. The less restrictive the isolation (the least restrictive is *read uncommitted*), the dirtier the data; this is because other users can get the values you are not going to commit eventually.

Usually, it is enough to use the default level, which (although it may be different for different databases) is typically `TRANSACTION_READ_COMMITTED`. JDBC allows you to get the current transaction isolation level using the `Connection` method called `getTransactionIsolation()`, while the `setTransactionIsolation()` method allows you to set any other level as needed.

In the case of complex decision-making logic about which changes need to be committed and which need to be rolled back, one can use two `Connection` methods to create and delete *savepoints*. The `setSavepoint(String savepointName)` method creates a new savepoint and returns a `Savepoint` object, which can later be used to roll back all the changes up to this point using the `rollback (Savepoint savepoint)` method. A savepoint can be deleted by calling `releaseSavepoint(Savepoint savepoint)`.

The most complex type of database transactions is *distributed transactions*. They are sometimes called *global transactions*, *XA transactions*, or *JTA transactions* (the latter is a Java API that consists of two Java packages, namely `javax.transaction` and `javax.transaction.xa`). They allow you to create and execute a transaction that spans operations across two different databases. Providing a detailed overview of distributed transactions is outside the scope of this book.

Working with large objects

In this recipe, you will learn how to store and retrieve a LOB that can be one of the three types: **Binary Large Object** (**BLOB**), **Character Large Object** (**CLOB**), and **National Character Large Object** (**NCLOB**).

Getting ready

The actual processing of LOB objects inside a database is vendor-specific, but JDBC APIs hide these implementation details from the application by representing the three LOB types as interfaces: `java.sql.Blob`, `java.sql.Clob`, and `java.sql.NClob`.

`Blob` is usually used to store images or other non-alphanumeric data. On the way to the database, an image can be converted into a stream of bytes and stored using the `INSERT INTO` statement. The `Blob` interface allows you to find the length of the object and convert it into an array of bytes that can be processed by Java for the purpose of displaying the image, for example.

`Clob` allows you to store character data. `NClob` stores Unicode character data as a way to support internationalization. It extends the `Clob` interface and provides the same methods. Both interfaces allow you to find the length of LOB and a substring inside the value.

The methods in the `ResultSet`, `CallableStatement` (we will discuss this in the next recipe) and `PreparedStatement` interfaces allow an application to store and access the stored value in a variety of ways: some of them via setters and getters of the corresponding objects, while others as `bytes[]`, or as a binary, character, or ASCII stream.

How to do it...

Each database has its specific way of storing a LOB. In the case of PostgreSQL, `Blob` is usually mapped to the `OID` or `BYTEA` data type, while `Clob` and `NClob` are mapped to the `TEXT` type. So let's write a new method that will allow us to create tables programmatically:

```
private static void execute(String sql){
  try (Connection conn = getDbConnection()) {
    try (PreparedStatement st = conn.prepareStatement(sql)) {
      st.execute();
    }
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

This differs from `executeUpdate()`, as it calls the `execute()` method of `PreparedStatement` instead of `executeUpdate()`. In principle, we can use `execute()` instead of `executeUpdate()` everywhere, but in our implementation of `executeUpdate()`, we expect a return value (`count`), which is not returned in the case of creating a table; therefore, we wrote this new method. Now we can create three tables:

```
execute("create table images (id integer, image bytea)");
execute("create table lobs (id integer, lob oid)");
execute("create table texts (id integer, text text)");
```

Look at the JDBC interfaces `PreparedStatement` and `ResultSet` and you'll notice the setters and getters for the objects--`get/setBlob()`, `get/setClob()`, `get/setNClob()`, `get/setBytes()`-- in memory and the methods that use `InputStream` and `Reader` (`get/setBinaryStream()`, `get/setAsciiStream()`, or `get/setCharacterStream()`). The big advantage of streaming methods is that they move data between the database and source without storing the whole LOB in memory.

However, the object's setters and getters are closer to our heart as they are used to object-oriented coding. So we will start with them, using not too big objects, for demo purposes. We expect the code to work just fine:

```
try (Connection conn = getDbConnection()) {
  String sql = "insert into images (id, image) values(?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    File file = new File("src/com/packt/cookbook/ch06_db/image1.png");
    FileInputStream fis = new FileInputStream(file);
    Blob blob = conn.createBlob();
    OutputStream out = blob.setBinaryStream(1);
    int i = -1;
    while ((i = fis.read()) != -1) {
      out.write(i);
    }
    st.setBlob(2, blob);
    int count = st.executeUpdate();
    System.out.println("Update count = " + count);
  }
} catch (Exception ex) { ex.printStackTrace(); }
```

Alternatively, in the case of `Clob`, we write this code:

```
try (Connection conn = getDbConnection()) {
  String sql = "insert into texts (id, text) values(?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    File file = new File("src/com/packt/cookbook/ch06_db/"
                         + "Chapter06Database.java");
    Reader reader = new FileReader(file);
    st.setClob(2, reader);
    int count = st.executeUpdate();
    System.out.println("Update count = " + count);
  }
} catch (Exception ex) { ex.printStackTrace(); }
```

It turns out not all methods available in the JDBC API are actually implemented by the drivers of all the databases. For example, `createBlob()` seems to work just fine for Oracle and MySQL, but in the case of PostgreSQL, we get this:

```
traverseRS(select * from images):
java.sql.SQLFeatureNotSupportedException: Method org.postgresql.jdbc.PgConnection.createBlob() is not yet implemented.
    at org.postgresql.Driver.notImplemented(Driver.java:640)
```

For the `Clob` example, we get this:

```
traverseRS(select * from texts):

java.sql.SQLFeatureNotSupportedException: Method org.postgresql.jdbc.PgPreparedStatement.setClob(int, Reader) is not yet implemented.
    at org.postgresql.Driver.notImplemented(Driver.java:640)
```

We can try to retrieve an object from `ResultSet` via the getter as well:

```
String sql = "select image from images";
try (PreparedStatement st = conn.prepareStatement(sql)) {
  st.setInt(1, 100);
  try(ResultSet rs = st.executeQuery()){
    while (rs.next()){
      Blob blob = rs.getBlob(1);
      System.out.println("blob length = " + blob.length());
    }
  }
}
```

The result will be as follows:

```
org.postgresql.util.PSQLException: Bad value for type long : \x89504e470d0a1a0a0000000d49484452000
    at org.postgresql.jdbc.PgResultSet.toLong(PgResultSet.java:2861)
    at org.postgresql.jdbc.PgResultSet.getLong(PgResultSet.java:2072)
    at org.postgresql.jdbc.PgResultSet.getBlob(PgResultSet.java:420)
```

Apparently, knowing the JDBC API is not enough; one has to read the documentation for the database too. Here is what the documentation for PostgreSQL (`https://jdbc.postgresql.org/documentation/80/binary-data.html`) has to say about LOB handling:

To use the BYTEA data type you should simply use the getBytes(), setBytes(), getBinaryStream(), or setBinaryStream() methods.
To use the Large Object functionality you can use either the LargeObject class provided by the PostgreSQL JDBC driver, or by using the getBLOB() and setBLOB() methods.

Also, *you must access Large Objects within an SQL transaction block. You can start a transaction block by calling setAutoCommit(false)*.

Without knowing such specifics, figuring out a way to handle LOBs would require a lot of time and cause much frustration.

When dealing with LOBs, we will use the streaming methods first because streaming directly from the source into the database or the other way around does not consume memory as much as the setters and getters do (which have to load LOB in memory first). Here is the code that streams `Blob` in/from PostgreSQL:

```
traverseRS("select * from images");
System.out.println();
```

```
try (Connection conn = getDbConnection()) {
  String sql = "insert into images (id, image) values(?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    File file = new File("src/com/packt/cookbook/ch06_db/image1.png");
    FileInputStream fis = new FileInputStream(file);
    st.setBinaryStream(2, fis);
    int count = st.executeUpdate();
    System.out.println("Update count = " + count);
  }
  sql = "select image from images where id = ?";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    try(ResultSet rs = st.executeQuery()){
      while (rs.next()){
        try(InputStream is = rs.getBinaryStream(1)){
          int i;
          System.out.print("ints = ");
          while ((i = is.read()) != -1) {
            System.out.print(i);
          }
        }
      }
    }
  }
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from images");
```

This will be your result. We have cut the screenshot arbitrarily on the right-hand side; otherwise, it is very long horizontally:

```
traverseRS(select * from images):

Update count = 1
ints = 13780787113102610000137372688200175000174860001917347270012241056767807367673280114111102105

traverseRS(select * from images):
id = 100, image = \x89504e470d0a1a0a0000000d494844520000014b000000ae0806000000bf492f1b00000c186943
```

Another way to process the retrieved image is to use `byte[]`:

```
try (Connection conn = getDbConnection()) {
  String sql =  "insert into images (id, image) values(?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    File file = new File("src/com/packt/cookbook/ch06_db/image1.png");
    FileInputStream fis = new FileInputStream(file);
    byte[] bytes = fis.readAllBytes();
    st.setBytes(2, bytes);
    int count = st.executeUpdate();
    System.out.println("Update count = " + count);
  }
  sql = "select image from images where id = ?";
  System.out.println();
  try (PreparedStatement st = conn.prepareStatement(sql)) {
```

```
    st.setInt(1, 100);
    try(ResultSet rs = st.executeQuery()){
      while (rs.next()){
        byte[] bytes = rs.getBytes(1);
        System.out.println("bytes = " + bytes);
      }
    }
  }
} catch (Exception ex) { ex.printStackTrace(); }
```

PostgreSQL limits the BYTEA size to 1 GB. Larger binary objects can be stored as the **object identifier** (**OID**) data type:

```
traverseRS("select * from lobs");
System.out.println();
try (Connection conn = getDbConnection()) {
  conn.setAutoCommit(false);
  LargeObjectManager lobm =
        conn.unwrap(org.postgresql.PGConnection.class)
            .getLargeObjectAPI();
  long lob = lobm.createLO(LargeObjectManager.READ
                           | LargeObjectManager.WRITE);
  LargeObject obj = lobm.open(lob, LargeObjectManager.WRITE);
  File file = new File("src/com/packt/cookbook/ch06_db/image1.png");
  try (FileInputStream fis = new FileInputStream(file)){
    int size = 2048;
    byte[] bytes = new byte[size];
    int len = 0;
    while ((len = fis.read(bytes, 0, size)) > 0) {
      obj.write(bytes, 0, len);
    }
    obj.close();
    String sql = "insert into lobs (id, lob) values(?, ?)";
    try (PreparedStatement st = conn.prepareStatement(sql)) {
      st.setInt(1, 100);
      st.setLong(2, lob);
      st.executeUpdate();
    }
  }
    conn.commit();
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from lobs");
```

The result will be as follows:

```
traverseRS(select * from lobs):


traverseRS(select * from lobs):
id = 100, lob = 304191
```

Note that the `select` statement returns a long value from the `lob` column. This is because the `OID` column does not store the value itself like BYTEA does.

Instead, it stores the reference to the object that is stored somewhere else in the database. Such an arrangement makes deleting the row with the OID type not as straightforward as this:

```
execute("delete from lobs where id = 100");
```

If one does just that, it leaves the actual object an orphan that continues to consume disk space, but, that is not referred to by any of the application tables. To avoid this problem, one has to `unlink` the LOB first by executing the following command:

```
execute("select lo_unlink((select lob from lobs " + " where id=100))");
```

Only after this can you safely execute the `delete from lobs where id = 100` command.

If you forget to `unlink` first, or if you create an orphan LOB accidentally (because of an error in the code or something), there is a way to find orphans in system tables. Again, database documentation should provide you with instructions on how to do this. In the case of PostgreSQL v.9.3 or later, you can check whether you have an orphan LOB by executing the `select count(*) from pg_largeobject` command. If it returns a count that is bigger than 0, then you can delete all the orphans with the following join (assuming that the `lobs` table is the only one that can refer to a LOB):

```
SELECT lo_unlink(pgl.oid) FROM pg_largeobject_metadata pgl
WHERE (NOT EXISTS (SELECT 1 FROM lobs ls" + "WHERE ls.lob = pgl.oid));
```

This is an overhead, though--the price one has to pay for storing a LOB in a database. It's worth noticing that although BYTEA does not require such complexity during the delete operation, it has a different kind of overhead. According to the PostgreSQL documentation, when close to 1 GB, *it would require a huge amount of memory to process such a large value.*

To read LOB data, one can use the following code:

```
try (Connection conn = getDbConnection()) {
  conn.setAutoCommit(false);
  LargeObjectManager lobm =
          conn.unwrap(org.postgresql.PGConnection.class)
              .getLargeObjectAPI();
  String sql = "select lob from lobs where id = ?";
```

```
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    try(ResultSet rs = st.executeQuery()){
      while (rs.next()){
        long lob = rs.getLong(1);
        LargeObject obj = lobm.open(lob, LargeObjectManager.READ);
        byte[] bytes = new byte[obj.size()];
        obj.read(bytes, 0, obj.size());
        System.out.println("bytes = " + bytes);
        obj.close();
      }
    }
  }
  conn.commit();
} catch (Exception ex) { ex.printStackTrace(); }
```

Alternatively, one can also use an even simpler version by getting `Blob` directly from the `ResultSet` object if the LOB is not too big:

```
while (rs.next()){
  Blob blob = rs.getBlob(1);
  byte[] bytes = blob.getBytes(1, (int)blob.length());
  System.out.println("bytes = " + bytes);
}
```

To store `Clob` in PostgreSQL, one can use the same code as the preceding one. While reading from the database, one can convert bytes into a `String` data type or something similar (again, if the LOB is not too big):

```
String str = new String(bytes, Charset.forName("UTF-8"));
System.out.println("bytes = " + str);
```

However, `Clob` in PostgreSQL can be stored directly as data type `TEXT` that is unlimited in size. This code reads the file where this code is written and stores/retrieves it in/from the database:

```
traverseRS("select * from texts");
System.out.println();
try (Connection conn = getDbConnection()) {
  String sql = "insert into texts (id, text) values(?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    File file = new File("src/com/packt/cookbook/ch06_db/"
                         + "Chapter06Database.java");
    try (FileInputStream fis = new FileInputStream(file)) {
      byte[] bytes = fis.readAllBytes();
      st.setString(2, new String(bytes, Charset.forName("UTF-8")));
    }
    int count = st.executeUpdate();
    System.out.println("Update count = " + count);
  }
  sql = "select text from texts where id = ?";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
```

```
    st.setInt(1, 100);
    try(ResultSet rs = st.executeQuery()){
      while (rs.next()) {
        String str = rs.getString(1);
        System.out.println(str);
      }
    }
  }
} catch (Exception ex) { ex.printStackTrace(); }
```

The result will be as follows (we have shown only the first few lines of the output):

```
traverseRS(select * from texts):

Update count = 1
package com.packt.cookbook.ch06_db;

import org.postgresql.ds.PGSimpleDataSource;
import org.postgresql.ds.PGPoolingDataSource;
import org.postgresql.largeobject.LargeObject;
import org.postgresql.largeobject.LargeObjectManager;
```

For bigger objects, streaming methods would be a better (if not the only) choice:

```
traverseRS("select * from texts");
System.out.println();
try (Connection conn = getDbConnection()) {
  String sql = "insert into texts (id, text) values(?, ?)";
  try (PreparedStatement st = conn.prepareStatement(sql)) {
    st.setInt(1, 100);
    File file = new File("src/com/packt/cookbook/ch06_db/"
                         + "Chapter06Database.java");
    //This is not implemented:
    //st.setCharacterStream(2, reader, file.length());
    st.setCharacterStream(2, reader, (int)file.length());

    int count = st.executeUpdate();
    System.out.println("Update count = " + count);
  }
} catch (Exception ex) { ex.printStackTrace(); }
System.out.println();
traverseRS("select * from texts");
```

Note that `setCharacterStream(int, Reader, long)` is not implemented, while `setCharacterStream(int, Reader, int)` works just fine.

We can also read the file from the `texts` table as a character stream and limit it to the first 160 characters:

```
String sql = "select text from texts where id = ?";
try (PreparedStatement st = conn.prepareStatement(sql)) {
```

```
  st.setInt(1, 100);
  try(ResultSet rs = st.executeQuery()){
    while (rs.next()) {
      try(Reader reader = rs.getCharacterStream(1)) {
        char[] chars = new char[160];
        reader.read(chars);
        System.out.println(chars);
      }
    }
  }
}
```

The result will be as follows:

```
package com.packt.cookbook.ch06_db;

import org.postgresql.ds.PGSimpleDataSource;
import org.postgresql.ds.PGPoolingDataSource;

import java.io.File;
import jav
```

There's more...

Here is another recommendation from the PostgreSQL documentation (you can access it at `https://jdbc.postgresql.org/documentation/80/binary-data.html`):

The BYTEA data type is not well suited for storing very large amounts of binary data. While a column of type BYTEA can hold up to 1 GB of binary data, it would require a huge amount of memory to process such a large value.
The Large Object method for storing binary data is better suited to storing very large values, but it has its own limitations. Specifically deleting a row that contains a Large Object reference does not delete the Large Object. Deleting the Large Object is a separate operation that needs to be performed. Large Objects also have some security issues since anyone connected to the database can view and/or modify any Large Object, even if they don't have permissions to view/update the row containing the Large Object reference.

While deciding to store LOBs in a database, one has to remember that the bigger the database, the more difficult it is to maintain it. The speed of access--the main advantage of choosing a database as a storage facility--also slows down, and it is not possible to create indices for LOB types to improve the search. Also, one cannot use LOB columns in a `WHERE` clause, except for a few CLOB cases, or use LOB columns in multiple rows of `INSERT` or `UPDATE` statements.

So, before thinking about a database for a LOB, one should always consider whether storing the name of a file, keywords, and some other content properties in the database would be enough for the solution.

Executing stored procedures

In this recipe, you will learn how to execute a database-stored procedure from a Java program.

Getting ready

Once in a while, a real-life Java programmer encounters the need to manipulate and/or select data in/from several tables and comes up with a set of complex SQL statements. In one scenario, a database administrator looks at the suggested procedure and improves and optimizes it so much that it becomes impossible or at least impractical to implement it in Java. This is when the developed set of SQL statements can be wrapped into a stored procedure that is compiled and stored in the database and then invoked via the JDBC interface. Or, in another twist of fate, a Java programmer might encounter the need for incorporating a call to an existing stored procedure into the program. To accomplish this, the interface `CallableStatement` (which extends interface `PreparedStatement`) can be used, although some databases allow you to call a stored procedure using either interface `Statement` or `PreparedStatement`.

`CallableStatement` can have parameters of three types: `IN` for an input value, `OUT` for the result, and `IN OUT` for either an input or an output value. `OUT` parameters must be registered by the `registerOutParameter()` method of `CallableStatement`. IN parameters are set the same way as the parameters of `PreparedStatement`.

Bear in mind that executing a stored procedure from Java programmatically is one of the least standardized areas. PostgreSQL, for example, does not support stored procedures directly, but they can be invoked as functions, which have been modified for this purpose by interpreting `OUT` parameters as return values. Oracle, on the other hand, allows `OUT` parameters for functions too.

This is why the following difference between database functions and stored procedures can serve only as a general guideline, not as a formal definition:

- A function has a return value, but it does not allow `OUT` parameters (except for some databases) and can be used in an SQL statement.

- A stored procedure does not have a return value (except for some databases); it allows `OUT` parameters (for most databases) and can be executed using the JDBC interface `CallableStatement`.

This is why reading the database documentation in order to learn how to execute a stored procedure is as important as, say, handling LOBs, discussed in the previous recipe.

Because stored procedures are compiled and stored in the database server, the `execute()` method of `CallableStatement` performs better for the same SQL statement than the corresponding method of `Statement` or `PreparedStatement`. This is one of the reasons a lot of Java code is sometimes replaced by one or several stored procedures that include even the business logic. Another reason for such a decision is that one can implement the solution the way one is most familiar with. So, there is no right answer for all cases and situations, and we will refrain from making specific recommendations, except to repeat the familiar mantra about the value of testing and the clarity of the code you are writing.

How to do it...

As in the previous recipe, we will continue using the PostgreSQL database for demonstration purposes. Before writing custom SQL statements, functions, and stored procedures, one should look at the list of already existing functions first. Usually, they provide a wealth of functionality.

Here is an example of calling the `replace(string text, from text, to text)` function that finds all the `from text` substrings in `string text` and replaces them with `to text`:

```
String sql = "{ ? = call replace(?, ?, ? ) }";
try (CallableStatement st = conn.prepareCall(sql)) {
  st.registerOutParameter(1, Types.VARCHAR);
  st.setString(2, "Hello, World! Hello!");
  st.setString(3, "llo");
  st.setString(4, "y");
  st.execute();
  String res = st.getString(1);
  System.out.println(res);
}
```

The result is as follows:

```
Hey, World! Hey!
```

We will incorporate this function into our custom functions and stored procedures in order to show how it can be done.

A stored procedure can be without any parameters at all, with `IN` parameters only, with `OUT` parameters only, or with both. The result may be one or multiple values, or a `ResultSet` object. Here is an example of creating a stored procedure without any parameters in PostgreSQL:

```
execute("create or replace function createTableTexts() "
        + " returns void as "
        + "$$ drop table if exists texts; "
        + "  create table texts (id integer, text text); "
        + "$$ language sql");
```

We use a method we are already familiar with:

```
private static void execute(String sql){
  try (Connection conn = getDbConnection()) {
    try (PreparedStatement st = conn.prepareStatement(sql)) {
      st.execute();
    }
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

This stored procedure (it is always a function in PostgreSQL) creates a `texts` table (drops this if the table exists already). You can find the syntax of the SQL for function creation in the database documentation. The only thing we would like to comment here is that instead of the symbol `$$` that denotes the function body, you can use single quotes. We prefer `$$` because it helps avoid the escaping of single quotes in case we need to include them in the function body.

This procedure can be invoked by `CallableStatement`:

```
String sql = "{ call createTableTexts() }";
try (CallableStatement st = conn.prepareCall(sql)) {
  st.execute();
}
```

Alternatively, it can be invoked with the SQL statement `select createTableTexts()` or `select * from createTableTexts()`. Both statements return a `ResultSet` object (which is `null` in the case of the `createTableTexts()` function), so we can traverse it by our method:

```
private static void traverseRS(String sql){
  System.out.println("traverseRS(" + sql + "):");
  try (Connection conn = getDbConnection()) {
    try (Statement st = conn.createStatement()) {
      try(ResultSet rs = st.executeQuery(sql)){
        int cCount = 0;
        Map<Integer, String> cName = new HashMap<>();
        while (rs.next()) {
          if (cCount == 0) {
            ResultSetMetaData rsmd = rs.getMetaData();
            cCount = rsmd.getColumnCount();
            for (int i = 1; i <= cCount; i++) {
              cName.put(i, rsmd.getColumnLabel(i));
            }
          }
          List<String> l = new ArrayList<>();
          for (int i = 1; i <= cCount; i++) {
```

```
            l.add(cName.get(i) + " = " + rs.getString(i));
          }
          System.out.println(l.stream()
                      .collect(Collectors.joining(", ")));
        }
      }
    }
  } catch (Exception ex) { ex.printStackTrace(); }
}
```

We have already used this method in the previous recipes.

The function can be deleted by the `drop function if exists createTableTexts()` statement.

Now let's put all of this together in Java code, create a function, and invoke it in three different styles:

```
execute("create or replace function createTableTexts() "
        + "returns void as "
        + "$$ drop table if exists texts; "
        + "  create table texts (id integer, text text); "
        + "$$ language sql");
String sql = "{ call createTableTexts() }";
try (Connection conn = getDbConnection()) {
  try (CallableStatement st = conn.prepareCall(sql)) {
    st.execute();
  }
}
traverseRS("select createTableTexts()");
traverseRS("select * from createTableTexts()");
execute("drop function if exists createTableTexts()");
```

The result is as follows:

```
traverseRS(select createTexts()):
createtexts = null
traverseRS(select * from createTexts()):
createtexts = null
```

Note that the name of the function is case-insensitive. We keep it camel case for human readability only.

Now let's create and call a stored procedure (function) with two input parameters:

```
execute("create or replace function insertText(int,varchar)"
        + " returns void "
        + " as $$ insert into texts (id, text) "
```

```
        + "   values($1, replace($2,'XX','ext'));"
        + " $$ language sql");
String sql = "{ call insertText(?, ?) }";
try (Connection conn = getDbConnection()) {
  try (CallableStatement st = conn.prepareCall(sql)) {
    st.setInt(1, 1);
    st.setString(2, "TXX 1");
    st.execute();
  }
}
execute("select insertText(2, 'TXX 2')");
traverseRS("select * from texts");
execute("drop function if exists insertText()");
```

In the function body, the input parameters were referred to by their `$1` and `$2` positions. As mentioned before, we also used the built-in `replace()` function to manipulate the values of the second input parameter before inserting it in the table. We called the newly created stored procedure twice: first via `CallableStatment` and then via the `execute()` method, with different input values. Then we looked inside the table using `traverseRS("select * from texts")` and dropped the newly created function to perform a cleanup (in real code, the function, once created, stays and takes advantage of being there, compiled and ready to run). If we run this code, we'll get the following result:

```
traverseRS(select * from texts):
id = 1, text = Text 1
id = 2, text = Text 2
```

The following code adds two rows to the `texts` table, then looks into it and creates a stored procedure (function) that counts the number of rows in the table and returns the result (note the `bigint` value of the returned value and the matching type for the `OUT` parameter `Types.BIGINT`):

```
execute("insert into texts (id, text) "
         + "values(3,'Text 3'),(4,'Text 4')");
traverseRS("select * from texts");
execute("create or replace function countTexts() "
        + "returns bigint as "
        + "$$ select count(*) from texts; "
        + "$$ language sql");
String sql = "{ ? = call countTexts() }";
try (Connection conn = getDbConnection()) {
  try (CallableStatement st = conn.prepareCall(sql)) {
    st.registerOutParameter(1, Types.BIGINT);
    st.execute();
    System.out.println("Result of countTexts() = " + st.getLong(1));
  }
}
```

```
traverseRS("select countTexts()");
traverseRS("select * from countTexts()");
execute("drop function if exists countTexts()");
```

The newly created stored procedure is executed three times and then deleted. The result is as follows:

```
Result of countTexts() = 4
traverseRS(select * from texts):
id = 1, text = Text 1
id = 2, text = Text 2
id = 3, text = Text 3
id = 4, text = Text 4
traverseRS(select countTexts()):
counttexts = 4
traverseRS(select * from countTexts()):
counttexts = 4
```

An example of a stored procedure with one input parameter (of the type `int`) that returns `ResultSet` will look like this (note the return type defined as `setof texts`, where `texts` is the name of the table):

```
execute("create or replace function selectText(int) "
        + "returns setof texts as
        + "$$ select * from texts where id=$1; "
        + "$$ language sql");
traverseRS("select selectText(1)");
traverseRS("select * from selectText(1)");
execute("drop function if exists selectText(int)");
```

The result will be as follows:

```
traverseRS(select selectText(1)):
selecttext = (1,"Text 1")
traverseRS(select * from selectText(1)):
id = 1, text = Text 1
```

It's worth analyzing the difference in the `ResultSet` content of two different calls to the stored procedure. Without `select *`, it contains the name of the procedure and the returned object (of the `ResultSet` type). But with `select *`, it returns the actual `ResultSet` content from the last SQL statement in the procedure.

Naturally, the question arises why we could not call this stored procedure via `CallableStatement`, like this:

```
String sql = "{ ? = call selectText(?) }";
try (CallableStatement st = conn.prepareCall(sql)) {
  st.registerOutParameter(1, Types.OTHER);
  st.setInt(2, 1);
  st.execute();
  traverseRS((ResultSet)st.getObject(1));
}
```

We tried, but it did not work. Here is what the PostgreSQL documentation has to say about it:

Functions that return data as a set should not be called via the CallableStatement interface, but instead should use the normal Statement or PreparedStatement interfaces.

There is a way around this limitation, though. The same database documentation describes how to retrieve a `refcursor` (a PostgreSQL-specific feature) value that can then be cast to `ResultSet`:

```
execute("create or replace function selectText(int) "
        + "returns refcursor " +
        + "as $$ declare curs refcursor; "
        + " begin "
        + "   open curs for select * from texts where id=$1;"
        + "     return curs; "
        + " end; "
        + "$$ language plpgsql");
String sql = "{ ? = call selectText(?) }";
try (Connection conn = getDbConnection()) {
  conn.setAutoCommit(false);
  try(CallableStatement st = conn.prepareCall(sql)){
    st.registerOutParameter(1, Types.OTHER);
    st.setInt(2, 2);
    st.execute();
    try(ResultSet rs = (ResultSet) st.getObject(1)){
      System.out.println("traverseRS(refcursor()=>rs):");
      traverseRS(rs);
    }
  }
}
traverseRS("select selectText(2)");
traverseRS("select * from selectText(2)");
execute("drop function if exists selectText(int)");
```

A few comments about the preceding code would probably help you understand how it was done:

- Autocommit has to be turned off

- Inside the function, `$1` refers to the first `IN` parameter (not counting the `OUT` parameter)
- The language is set to `plpgsql` in order to access the `refcursor` functionality (PL/pgSQL is a loadable procedural language of the PostgreSQL database)
- To traverse `ResultSet`, we wrote a new method, as follows:

```
        private void traverseRS(ResultSet rs) throws Exception {
          int cCount = 0;
          Map<Integer, String> cName = new HashMap<>();
          while (rs.next()) {
            if (cCount == 0) {
              ResultSetMetaData rsmd = rs.getMetaData();
              cCount = rsmd.getColumnCount();
              for (int i = 1; i <= cCount; i++) {
                cName.put(i, rsmd.getColumnLabel(i));
              }
            }
            List<String> l = new ArrayList<>();
            for (int i = 1; i <= cCount; i++) {
              l.add(cName.get(i) + " = " + rs.getString(i));
            }
            System.out.println(l.stream()
                      .collect(Collectors.joining(", ")));
          }
        }
```

So, our old friend can now be refactored into this:

```
private static void traverseRS(String sql){
  System.out.println("traverseRS(" + sql + "):");
  try (Connection conn = getDbConnection()) {
    try (Statement st = conn.createStatement()) {
      try(ResultSet rs = st.executeQuery(sql)){
        traverseRS(rs);
      }
    }
  } catch (Exception ex) { ex.printStackTrace(); }
}
```

The result will be as follows:

```
traverseRS(refcursor()=>rs):
id = 2, text = Text 2
traverseRS(select selectText(2)):
selecttext = <unnamed portal 1>
traverseRS(select * from selectText(2)):
selecttext = <unnamed portal 1>
```

You can see that the result-traversing methods that do not extract an object and cast it to `ResultSet` don't show the correct data in this case.

There's more...

We covered the most popular cases of calling stored procedures from Java code. The scope of this book did not allow us to present more complex and potentially useful forms of stored procedures in PostgreSQL and other databases. However, we would like to mention them here, so you can have an idea of other possibilities:

- Functions on composite types
- Functions with parameter names
- Functions with variable numbers of arguments
- Functions with default values for arguments
- Functions as table sources
- Functions returning tables
- Polymorphic SQL functions
- Functions with collations

Concurrent and Multithreaded Programming

Concurrent programming has always been a difficult task. It might sound easy, but it is a source of many hard-to-solve problems. In this chapter, we will show you different ways of incorporating concurrency and some best practices, such as immutability, which will help in creating better, concurrent applications. We will also discuss the implementation of some commonly used patterns, such as divide-conquer and publish-subscribe, using the constructs provided by Java. We will cover the following recipes:

- Using the basic element of concurrency - thread
- Different synchronization approaches
- Immutability as a means to achieve concurrency
- Using concurrent collections
- Using the executor service to execute async tasks
- Using fork/join to implement divide-and-conquer
- Using flow to implement the publish-subscribe pattern

Introduction

Concurrency--the ability to execute several procedures in parallel--becomes increasingly important as big data analysis moves into the mainstream of modern applications. Having CPUs or several cores in one CPU helps increase the throughput, but the growth rate of data volume will always outpace hardware advances. Besides, even in a multiple CPU system, one still has to structure the code and think about resource sharing to take advantage of the available computational power.

In the previous chapters, we demonstrated how lambdas with functional interfaces and parallel streams made concurrent processing a part of the toolkit of every Java programmer. One can easily take advantage of this functionality with minimal guidance, if at all.

In this chapter, we will describe some other--old (that existed before Java 9) and new--Java features and APIs that allow more control over concurrency. The high-level concurrency Java API has been around since Java 5. The JDK Enhancement Proposal (JEP) 266, *More Concurrency Updates,* introduced *an interoperable publish-subscribe framework, enhancements to the CompletableFuture API, and various other improvements* to Java 9 in the `java.util.concurrent` package. But before we dive into the details of the latest additions, let's review the basics of concurrent programming in Java and see how to use them.

Java has two units of execution: process and thread. A process usually represents the whole JVM, although an application can create another process using `ProcessBuilder`. But since the multiprocess case is outside the scope of this book, we will focus on the second unit of execution, that is, a thread, which is similar to a process but less isolated from other threads and requires fewer resources for execution.

A process can have many threads running and at least one thread called the main thread. Threads can share resources, including memory and open files,

which allows better efficiency, but comes with a price of higher risk of unintended mutual interference and even blocking of the execution. This is where programming skills and an understanding of the concurrency technique are required. And this is what we are going to discuss in this chapter.

Using the basic element of concurrency - thread

In this chapter, we will look at the `java.lang.Thread` class and see what it can do for concurrency and program performance in general.

Getting ready

A Java application starts as the main thread (not counting system threads that support the process). It can then create other threads and let them run in parallel (sharing the same core via time slicing of having a dedicated CPU for each thread). This can be done using the `java.lang.Thread` class that implements the `Runnable` functional interface with only one `run()` method.

There are two ways of creating a new thread: creating a subclass of `Thread` or implementing the `Runnable` interface and passing the object of the implementing class to the `Thread` constructor. We can invoke the new thread by calling the `start()` method of the `Thread` class (which, in turn, calls the `run()` method that we implemented).

Then, we can either let the new thread run until its completion or pause it and let it continue again. We can also access its properties or intermediate results, if needed.

How to do it...

First, we create a class called `AThread` that extends `Thread` and override its `run()` method:

```
class AThread extends Thread {
  int i1,i2;
  AThread(int i1, int i2){
    this.i1 = i1;
    this.i2 = i2;
  }
  public void run() {
    IntStream.range(i1, i2)
             .peek(Chapter07Concurrency::doSomething)
             .forEach(System.out::println);
  }
}
```

In this example, we wanted the thread to generate a stream of integers in a certain range. Then, we peeked into each emitted integer (the method `peek()` cannot change the stream element) and called the static method `doSomething()` of the main class in order to make the thread busy for some time. Refer to the following code:

```
private static int doSomething(int i){
  IntStream.range(i, 99999).asDoubleStream().map(Math::sqrt).average();
  return i;
}
```

As you can see, this method generates another stream of integers in the range of `i` and `99999`, then converts the stream into a stream of doubles, calculates the square root of each of the stream elements, and finally calculates an average average of the stream. We discard the result and return the passed-in parameter (as a convenience that allows us to keep the fluent style in the stream pipe of the thread, which ends up printing out each element). Using this new class, we can now demonstrate the concurrent execution of the three threads:

```
Thread thr1 = new AThread(1, 4);
thr1.start();

Thread thr2 = new AThread(11, 14);
thr2.start();

IntStream.range(21, 24)
         .peek(Chapter07Concurrency::doSomething)
         .forEach(System.out::println);
```

The first thread generates the integers `1`, `2`, and `3`, the second generates the integers `11`, `12`, and `13`, and the third thread (main one) generates `21`, `22`, and `23`.

As mentioned before, we can rewrite the same program by creating and using a class that could implement the `Runnable` interface:

```
class ARunnable implements Runnable {
  int i1,i2;
  ARunnable(int i1, int i2){
    this.i1 = i1;
    this.i2 = i2;
  }
  public void run() {
    IntStream.range(i1, i2)
             .peek(Chapter07Concurrency::doSomething)
             .forEach(System.out::println);
  }
}
```

So, you can run the same three threads like this:

```
Thread thr1 = new Thread(new ARunnable(1, 4));
thr1.start();

Thread thr2 = new Thread(new ARunnable(11, 14));
thr2.start();

IntStream.range(21, 24)
         .peek(Chapter07Concurrency::doSomething)
         .forEach(System.out::println);
```

We can also take advantage of `Runnable` being a functional interface and avoid creating an intermediate class by passing in a lambda expression instead:

```
Thread thr1 = new Thread(() -> IntStream.range(1, 4)
                  .peek(Chapter07Concurrency::doSomething)
```

```
                    .forEach(System.out::println));
thr1.start();

Thread thr2 = new Thread(() -> IntStream.range(11, 14)
                    .peek(Chapter07Concurrency::doSomething)
                    .forEach(System.out::println));
thr2.start();

IntStream.range(21, 24)
          .peek(Chapter07Concurrency::doSomething)
          .forEach(System.out::println);
```

Which implementation is better depends on your goal and style. Implementing `Runnable` has an advantage (and in some cases, the only possible option) that it allows the implementation to extend from another class. It is particularly helpful when you would like to add thread-like behavior to an existing class. You can even invoke method `run()` directly, without passing the object to the `Thread` constructor.

Using a lambda expression wins over `Runnable` implementation when only method `run()` implementation is needed, no matter how big it is. If it is too big, you can have it isolated in a separate method:

```
public static void main(String arg[]) {
  Thread thr1 = new Thread(() -> runImpl(1, 4));
  thr1.start();

  Thread thr2 = new Thread(() -> runImpl(11, 14));
  thr2.start();

  runImpl(21, 24);
}

private static void runImpl(int i1, int i2){
  IntStream.range(i1, i2)
           .peek(Chapter07Concurrency::doSomething)
           .forEach(System.out::println);
}
```

One would be hard pressed to come up with a shorter implementation of the same functionality.

If we run any of the preceding versions, we will get something like this:

```
1
11
21
12
2
22
13
3
23
```

As you can see, the three threads print out their numbers in parallel, but the sequence depends on the particular JVM implementation and underlying operating system. So, you will probably get different output. Besides, it also changes from run to run.

The `Thread` class has several constructors that allow you to set the thread name and the group it belongs to. Grouping of threads helps manage them in case there are many threads running in parallel. The class also has several methods that provide information about the thread's status and properties and allow you to control its behavior. Add these two lines to the preceding example:

```
System.out.println("Id=" + thr1.getId() + ", " + thr1.getName() + ",
                   priority=" + thr1.getPriority() + ",
                   state=" + thr1.getState());
System.out.println("Id=" + thr2.getId() + ", " + thr2.getName() + ",
                   priority=" + thr2.getPriority() + ",
                   state=" + thr2.getState());
```

You'll get something like this:

```
Id=13, Thread-0, priority=5, state=RUNNABLE
Id=14, Thread-1, priority=5, state=TERMINATED
```

Next, say, you add names to the threads:

```
Thread thr1 = new Thread(() -> runImpl(1, 4), "First Thread");
thr1.start();

Thread thr2 = new Thread(() -> runImpl(11, 14), "Second Thread");
thr2.start();
```

In this case, the output will show the following:

```
Id=13, First Thread, priority=5, state=RUNNABLE
Id=14, Second Thread, priority=5, state=TERMINATED
```

The thread's `id` is generated automatically and cannot be changed, but it can be reused after the thread is terminated. The thread name, on the other hand, can be shared by several threads. Priority can be set programmatically with a value between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`. The smaller the value, the more time the thread is allowed to run (which means it has higher priority). If not set, priority value defaults to `Thread.NORM_PRIORITY`. The state of a thread can have one of the following values:

- `NEW`: When a thread has not yet started
- `RUNNABLE`: When a thread is being executed
- `BLOCKED`: When a thread is blocked and is waiting for a monitor lock
- `WAITING`: When a thread is waiting indefinitely for another thread to perform a particular action
- `TIMED_WAITING`: When a thread is waiting for another thread to perform an action for up to a specified waiting time
- `TERMINATED`: When a thread has exited

We will talk about the `BLOCKED` and `WAITING` states in one of our recipes later.

The `sleep()` method can be used to suspend the thread execution for a specified (in milliseconds) period of time. The complementary method `interrupt()` sends `InterruptedException` to the thread that can be used to wake up the *sleeping* thread. Let's work this out in the code and create a new class:

```
private static class BRunnable implements Runnable {
  int i1, result;
  BRunnable(int i1){ this.i1 = i1; }
  public int getCurrentResult(){ return this.result; }
  public void run() {
    for(int i = i1; i < i1 + 6; i++){
      //Do something useful here
      this.result = i;
      try{ Thread.sleep(1000);
      } catch(InterruptedException ex){}
    }
  }
}
```

The preceding code produces intermediate results, which are stored in the property `result`. Each time a new result is produced, the thread sleeps for one second. In this specific example, written for demonstration purposes only, the code does not do anything particularly useful. It just iterates over a set of values and considers each of them a result. In a real-world code, you would do some calculations here based on the current state of the system or something and assign the calculated value to the property `result`. Now let's use this class:

```
BRunnable r1 = new BRunnable(1);
Thread thr1 = new Thread(r1);
thr1.start();

IntStream.range(21, 29)
         .peek(i -> thr1.interrupt())
         .filter(i ->  {
           int res = r1.getCurrentResult();
           System.out.print(res + " => ");
           return res % 2 == 0;
         })
         .forEach(System.out::println);
```

The preceding program generates a stream of integers: 21, 22, ..., 28. After each integer is generated, the main thread interrupts the `thr1` thread and lets it generate the next result, which is then accessed via the `getCurrentResult()` method and analyzed. If the current result is an even number, the filter allows the generated number flow to be printed out. If not, it is skipped. Here is a possible result:

```
1 => 2 => 22
3 => 3 => 4 => 25
6 => 26
6 => 27
6 => 28
```

It will be different if the program is run on different computers, but you get the idea: this way, one thread can control the output of another thread.

There's more...

There are two other important methods that allow threads to cooperate. First is the `join()` method that allows the current thread to wait until another thread is terminated. Overloaded versions of `join()` accept the parameters that define how long the thread has to wait before it could do something else.

The `setDaemon()` method terminates the thread automatically after all the non-daemon threads are terminated. Usually, it is used for background and supporting processes.

See also

Refer to the following recipes in this chapter:

- Different synchronization approaches
- Immutability as a means to achieve concurrency
- Using concurrent collections
- Using the executor service to execute async tasks
- Using fork/join to implement divide-and-conquer
- Using flow to implement the publish-subscribe pattern

Different synchronization approaches

In this recipe, you will learn about the two most popular and basic methods of managing concurrent access to common resources in Java: a `synchronized` method and a `synchronized` block.

Getting ready

Two or more threads modifying the same value while other threads reading it is the most general description of a problem of concurrent access. Subtler problems include *thread interference* and *memory consistency errors*, both producing unexpected results in seemingly benign fragments of code. We are going to demonstrate such cases and ways to avoid them.

At first glance, it seems quite straightforward: just allow only one thread at a time to modify/access the resource and that's it. But if the access takes a long time, then it creates a bottleneck that might eliminate the advantage of having many threads working in parallel. Or, if one thread blocks access to one resource while waiting for access to another resource and the second thread blocks access to the second resource while waiting for access to the first one, then it creates a problem called a deadlock. These are two very simple examples of the possible challenges a programmer has to tackle while dealing with multiple threads.

How to do it...

First, we'll check out a problem caused by concurrency. Let's create a `Calculator` class that has the `calculate()`: method:

```
class Calculator{
  private double prop;
  public double calculate(int i){
    this.prop = 2.0 * i;
    DoubleStream.generate(new Random()::nextDouble)
                .limit(10);
    return Math.sqrt(this.prop);
  }
}
```

This method assigns an input value to a property and then calculates its square root. We also inserted code that generates a stream of 10 values. We did this in order to keep the method busy for some time. Otherwise, everything is done so quickly that there will be little chance for any concurrency to occur. Also, we wanted the return value to be obviously the same all the time, so we did not complicate it by having calculations and made the method busy by an unrelated activity. Now we are going to use it in the following code:

```
Calculator c = new Calculator();
Thread thr1 = new Thread(() -> System.out.println(IntStream.range(1, 4)
          .peek(x ->DoubleStream.generate(new Random()::nextDouble)
          .limit(10)).mapToDouble(c::calculate).sum()));
thr1.start();
Thread thr2 = new Thread(() -> System.out.println(IntStream.range(1, 4)
          .mapToDouble(c::calculate).sum()));
thr2.start();
```

Even for a novice, it will be obvious that two threads accessing the same object will have a good chance of stepping on each other's toes. As you can see, the `Random` interface implementation prints out the sum of the same three numbers, that is, 1, 2, and 3, after each one of them is processed by the `calculate()` method of the object of `Calculator`. Inside `calculate()`, each number is multiplied by two and then pushed through the square root extraction process. The operation is so simple that we can even calculate it by hand in advance. The result is going to be `5.863703305156273`. And again, to the first

thread, we have added the `peek()` operator with 10 double-generating code to make it run a bit slower to give concurrency a better chance to happen. If you run these examples on your computer and do not see the effect of concurrency, try to increase the number of doubles by replacing `10` with `100`, for example.

Now run the code. Here is one of the possible results:

```
5.414213562373095
5.863703305156273
```

One thread got the correct result, while the other did not. Apparently, in the period between setting the value of the `prop` property and then using it to return the result of the `calculate()` method, the other thread managed to assign its (smaller) value to `prop`. This is the case of thread interference.

There are two ways to protect code from such a problem: using a `synchronized` method or a `synchronized` block; these help include lines of code that are always executed as an atomic operation without any interference from another thread.

Making a `synchronized` method is easy and straightforward:

```
class Calculator{
  private double prop;
  synchronized public double calculate(int i){
    this.prop = 2.0 * i;
    DoubleStream.generate(new Random()::nextDouble).limit(10);
    return Math.sqrt(this.prop);
  }
}
```

We just add the `synchronized` keyword in front of the method definition. Now, no matter how big a stream of doubles is generated, the result of our program is always going to be as follows:

```
5.863703305156273
5.863703305156273
```

This is because another thread cannot enter the `synchronized` method until the current thread (the one that has entered the method already) has exited it.

This approach may cause performance degradation if the method takes a long time to execute, so many threads might be blocked, waiting for their turn to use the method. In such cases, a `synchronized` block can be used to wrap several lines of code in an atomic operation:

```
private static class Calculator{
  private double prop;
  public double calculate(int i){
    synchronized (this) {
      this.prop = 2.0 * i;
      DoubleStream.generate(new Random()::nextDouble).limit(10);
      return Math.sqrt(this.prop);
    }
  }
```

Alternatively, you can do this:

```
private static class Calculator{
  private double prop;
  public double calculate(int i){
    DoubleStream.generate(new Random()::nextDouble).limit(10);
    synchronized (this) {
      this.prop = 2.0 * i;
      return Math.sqrt(this.prop);
    }
  }
```

We can do this because by studying the code, we have discovered that we can rearrange it such that the synchronized portion will be much smaller, thus having fewer chances to become a bottleneck.

A `synchronized` block acquires a lock on an object, any object for that matter. It could be, for example, a dedicated one:

```
private static class Calculator{
  private double prop;
  private Object calculateLock = new Object();
  public double calculate(int i){
    DoubleStream.generate(new Random()::nextDouble).limit(10);
    synchronized (calculateLock) {
      this.prop = 2.0 * i;
      return Math.sqrt(this.prop);
    }
  }
}
```

The advantage of a dedicated lock is that you can be sure that such a lock will be used for accessing a particular block only. Otherwise, the current

object (`this`) might be used to control access to another block; in a huge class, you might not notice this while writing your code. A lock can also be acquired from a class, which is even more susceptible to sharing for unrelated purposes.

We did all these examples just for demonstrating synchronization approaches. If they were to be real code, we would just let each thread create its own `Calculator` object:

```
Thread thr1 = new Thread(() -> System.out.println(IntStream.range(1, 4)
  .peek(x ->DoubleStream.generate(new Random()::nextDouble)
  .limit(10))
  .mapToDouble(x -> {
     Calculator c = new Calculator();
     return c.calculate(x);
  }).sum()));
thr1.start();

Thread thr2 = new Thread(() -> System.out.println(IntStream.range(1, 4)
    .mapToDouble(x -> {
       Calculator c = new Calculator();
       return c.calculate(x);
    }).sum()));
thr2.start();
```

This would be in line with the general idea of making lambda expressions independent of the context in which they are created. This is because in a multithreaded environment, one never knows how the context would look during their execution. The cost of creating a new object every time is negligible unless a large amount of data has to be processed, and testing ensures that the object creation overhead is noticeable. Making the `calculate()` method (and the property) static (which is tempting as it avoids object creation and preserves the fluent style) would not eliminate the concurrency problem because one shared (at the class level this time) value of the property would still remain in place.

Memory consistency errors can have many forms and causes in a multithreaded environment. They are well discussed in the Javadoc of the `java.util.concurrent` package. Here we will mention only the most common case caused by lack of visibility. When one thread changes a property value, the other might not see the change immediately, and you cannot use `synchronized` keyword for a primitive type. In such a situation,

consider using the `volatile` keyword for such a property; it guarantees its read/write visibility between different threads.

There's more...

Different types of locks for different needs and with different behavior are assembled in the `java.util.concurrent.locks` package.

The `java.util.concurrent.atomic` package provides support for lock-free, thread-safe programming on single variables.

The following classes provide synchronization support too:

- `Semaphore`: This restricts the number of threads that can access some resource
- `CountDownLatch`: This allows one or more threads to wait until a set of operations being performed in other threads are completed
- `CyclicBarrier`: This allows a set of threads to wait for each other to reach a common barrier point
- `Phaser`: This provides a more flexible form of barrier that may be used to control phased computation among multiple threads
- `Exchanger`: This allows two threads to exchange objects at a rendezvous point and is useful in several pipeline designs

Each object in Java is inherited from the base object's `wait()`, `notify()`, and `notifyAll()` methods; these can also be used to control threads' behavior and their access to and release from locks.

The `Collections` class has methods that make various collections synchronized. However, this means that only the modifications of the collection could become thread-safe, not the changes to the collection members. Also, while traversing the collection via its iterator, it has to be protected too because an iterator is not thread-safe. Here is a Javadoc example of the correct usage of a synchronized map:

```
Map m = Collections.synchronizedMap(new HashMap());
...
Set s = m.keySet(); // Needn't be in synchronized block
...
```

```
synchronized (m) { // Synchronizing on m, not s!
  Iterator i = s.iterator(); //Must be synchronized block
  while (i.hasNext())
  foo(i.next());
}
```

To add more to your plate as a programmer, you have to realize that the following code is not thread-safe:

```
List<String> l =
       Collections.synchronizedList(new ArrayList<>());
l.add("first");
//... code that adds more elements to the list
int i = l.size();
//... some other code
l.add(i, "last");
```

This is because although `List l` is synchronized, in multithreaded processing, it is quite possible that some other code would add more elements to the list (then the intended last does not reflect the reality) or remove an element (then the code fails with `IndexOutOfBoundException`).

The ones described here are a few of the most often encountered concurrency problems. These problems are not easy to solve. That is why it is not surprising that more and more developers now take a more radical approach. They avoid managing the state. Instead, they prefer writing non-blocking code for asynchronous and parallel processing data in a set of stateless operations. We saw similar code in the chapter about stream pipelines. It seems that Java and many modern languages and computer systems are evolving in this direction.

See also

Refer to the following recipes in this chapter:

- Immutability as a means to achieve concurrency
- Using concurrent collections
- Using the executor service to execute async tasks
- Using fork/join to implement divide-and-conquer
- Using flow to implement the publish-subscribe pattern

Immutability as a means to achieve concurrency

In this recipe, you will learn how to use immutability against problems caused by concurrency.

Getting ready

A concurrency problem most often occurs when different threads modify and read data from the same shared resource. Decreasing the number of modifying operations diminishes the risk of concurrency issues. This is where immutability--the condition of read-only values--enters the stage.

Object immutability means an absence of means to change its state after the object has been created. It does not guarantee thread safety but helps to increase it significantly and provide sufficient protection from concurrency problems in many practical applications.

Creating a new object instead of reusing an existing one (by changing its state via setters and getters) is often perceived as an expensive approach. But with the power of modern computers, there has to be a huge number of object creations done for performance to be affected in any significant way. And even if that is the case, programmers often prefer some performance degradation as the price for getting more reliable results.

How to do it...

Here is a very basic class that produces mutable objects:

```
class MutableClass{
  private int prop;
  public MutableClass(int prop){
    this.prop = prop;
  }
  public int getProp(){
    return this.prop;
  }
  public void setProp(int prop){
    this.prop = prop;
  }
}
```

To make it immutable, we need to remove the setter and add the `final` keyword to its only property and the class itself:

```
final class ImmutableClass{
  final private int prop;
  public ImmutableClass(int prop){
    this.prop = prop;
  }
  public int getProp(){
    return this.prop;
  }
}
```

Adding the `final` keyword to a class prevents it from being extended, so its methods cannot be overridden. Adding `final` to a private property is not as obvious. The motivation is somewhat complex and has to do with the way the compiler reorders the fields during object construction. If the field is declared `final`, it is treated by the compiler as synchronized. That is why adding `final` to a private property is necessary to make the object completely immutable.

The challenge mounts up if the class is composed of other classes, especially mutable ones. When this happens, the injected class might bring in code that would affect the containing class. Also, the inner (mutable) class, which is retrieved by references via the getter, could then be modified and propagate the change inside the containing class. The way to close such

holes is to generate new objects during the composition of the object retrieval. Here is an example of this:

```
final class ImmutableClass{
  private final double prop;
  private final MutableClass mutableClass;
  public ImmutableClass(double prop, MutableClass mc){
    this.prop = prop;
    this.mutableClass = new MutableClass(mc.getProp());
  }
  public double getProp(){
    return this.prop;
  }
  public MutableClass getMutableClass(){
    return new MutableClass(mutableClass.getProp());
  }
}
```

There's more...

In our examples, we used very simple code. If more complexity is added to any of the methods, especially with parameters (and especially when some of the parameters are objects), it is possible you'll get concurrency issues again:

```
int getSomething(AnotherMutableClass amc, String whatever){
  //... code is here that generates a value "whatever"
  amc.setProperty(whatever);
  //...some other code that generates another value "val"
  amc.setAnotherProperty(val);
  return amc.getIntValue();
}
```

Even if this method belongs to `ImmutableClass` and does not affect the state of the `ImmutableClass` object, it is still a subject of the thread's race and has to be analyzed and protected as needed.

The `Collections` class has methods that make various collections unmodifiable. It means that the modification of the collection itself becomes read only, not the collection members.

See also

Refer to the following recipes in this chapter:

- Using concurrent collections
- Using the executor service to execute async tasks
- Using fork/join to implement divide-and-conquer
- Using flow to implement the publish-subscribe pattern

Using concurrent collections

In this recipe, you will learn about the thread-safe collections of the `java.util.concurrent` package.

Getting ready

A collection can be synchronized if you apply one of the `Collections.synchronizeXYZ()` methods to it; here, we have used XYZ as a placeholder that represents either `Set`, `List`, `Map`, or one of the several collection types (see the API of the `Collections` class). We have already mentioned that the synchronization applies to the collection itself, not to its iterator or the collection members.

Such synchronized collections are also called **wrappers** because all of the functionality is still provided by the collections passed as parameters to the `Collections.synchronizeXYZ()` methods, so the wrappers provide only thread-safe access to them. The same effect could be achieved by acquiring a lock on the original collection. Obviously, such a synchronization incurs a performance overhead in a multithreading environment causing each thread to wait for its turn to access the collection.

A well-tuned application for performance implementation of thread-safe collections is provided by the `java.util.concurrent` package.

How to do it...

Each of the concurrent collections of the `java.util.concurrent` package implements (or extends, if it is an interface) one of the four interfaces of the `java.util` package: `List`, `Set`, `Map`, or `Queue`:

1. The `List` interface has only one implementation: the `CopyOnWriteArrayList` class. According to the Javadoc of this class, *all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.... The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException. The iterator will not reflect additions, removals, or changes to the list since the iterator was created. Element-changing operations on iterators themselves (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.* To demonstrate the behavior of the `CopyOnWriteArrayList` class, let's compare it with `java.util.ArrayList` (which is not a thread-safe implementation of `List`). Here is the method that adds an element to the list while iterating on the same list:

```
void demoListAdd(List<String> list) {
  System.out.println("list: " + list);
  try {
    for (String e : list) {
      System.out.println(e);
      if (!list.contains("Four")) {
        System.out.println("Calling list.add(Four)...");
        list.add("Four");
      }
    }
  } catch (Exception ex) {
    System.out.println(ex.getClass().getName());
  }
  System.out.println("list: " + list);
}
```

Consider the following code:

```
System.out.println("***** ArrayList add():");
demoListAdd(new ArrayList<>(Arrays
                  .asList("One", "Two", "Three")));

System.out.println();
System.out.println("***** CopyOnWriteArrayList add():");
demoListAdd(new CopyOnWriteArrayList<>(Arrays.asList("One",
                                  "Two", "Three")));
```

If we execute this code, the result would be as follows:

```
***** ArrayList add():
list: [One, Two, Three]
One
Calling list.add(Four)...
java.util.ConcurrentModificationException
list: [One, Two, Three, Four]

***** CopyOnWriteArrayList add():
list: [One, Two, Three]
One
Calling list.add(Four)...
Two
Three
list: [One, Two, Three, Four]
```

As you can see, `ArrayList` throws `ConcurrentModificationException` when the list is modified while being iterated (we used the same thread for simplicity and because it leads to the same effect, as in the case of another thread modifying the list). The specification, though, does not guarantee that the exception will be thrown or the list modification applied (as in in our case), so a programmer should not base the application logic on such behavior.
The `CopyOnWriteArrayList` class, on the other hand, tolerates the same intervention; however, notice that it does not add a new element to the current list because the iterator was created from a snapshot of the fresh copy of the underlying array.

Now let's try to remove a list element concurrently while traversing the list, using this method:

```
void demoListRemove(List<String> list) {
  System.out.println("list: " + list);
  try {
    for (String e : list) {
      System.out.println(e);
      if (list.contains("Two")) {
        System.out.println("Calling list.remove(Two)...");
        list.remove("Two");
```

```
        }
      }
    } catch (Exception ex) {
      System.out.println(ex.getClass().getName());
    }
    System.out.println("list: " + list);
  }
```

Consider the following code:

```
System.out.println("***** ArrayList remove():");
demoListRemove(new ArrayList<>(Arrays.asList("One",
                                  "Two", "Three")));

System.out.println();
System.out.println("***** CopyOnWriteArrayList remove():");
demoListRemove(new CopyOnWriteArrayList<>(Arrays
                        .asList("One", "Two", "Three")));
```

If we execute this, we will get the following:

```
***** ArrayList remove():
list: [One, Two, Three]
One
Calling list.remove(Two)...
java.util.ConcurrentModificationException
list: [One, Three]

***** CopyOnWriteArrayList remove():
list: [One, Two, Three]
One
Calling list.remove(Two)...
Two
Three
list: [One, Three]
```

The behavior is similar to the previous example. The `CopyOnWriteArrayList` class tolerates the concurrent access to the list but does not modify the current list's copy.

We knew `ArrayList` would not be thread-safe for a long time, so we used a different technique to remove an element from the list while traversing it. Here is how this was done before the Java 8 release:

```
void demoListIterRemove(List<String> list) {
  System.out.println("list: " + list);
  try {
    Iterator iter = list.iterator();
    while (iter.hasNext()) {
      String e = (String) iter.next();
      System.out.println(e);
      if ("Two".equals(e)) {
        System.out.println("Calling iter.remove()...");
```

```
            iter.remove();
          }
        }
      } catch (Exception ex) {
          System.out.println(ex.getClass().getName());
      }
      System.out.println("list: " + list);
    }
```

Let's try this and run the code:

```
System.out.println("***** ArrayList iter.remove():");
demoListIterRemove(new ArrayList<>(Arrays
                   .asList("One", "Two", "Three")));

System.out.println();
System.out.println("*****"
                   + " CopyOnWriteArrayList iter.remove():");
demoListIterRemove(new CopyOnWriteArrayList<>(Arrays
                   .asList("One", "Two", "Three")));
```

The result will be as follows:

```
***** ArrayList iter.remove():
list: [One, Two, Three]
One
Two
Calling iter.remove()...
Three
list: [One, Three]

***** CopyOnWriteArrayList iter.remove():
list: [One, Two, Three]
One
Two
Calling iter.remove()...
java.lang.UnsupportedOperationException
list: [One, Two, Three]
```

This is exactly what Javadoc warned about: "*Element-changing operations on iterators themselves (remove, set, and add) are not supported. These methods throw UnsupportedOperationException."* We should remember this when upgrading an application to make it work in a multithreaded environment: just changing from `ArrayList()` to `CopyOnWriteArrayList` would not be enough if we use an iterator to remove a list element.

Since Java 8, we have a better way to remove an element from a collection using a lambda, which we can and should use from now on (leaving plumbing details to the library):

```
void demoRemoveIf(Collection<String> collection) {
  System.out.println("collection: " + collection);
  System.out.println("Calling list.removeIf(e ->"
                     + " Two.equals(e))...");
  collection.removeIf(e -> "Two".equals(e));
  System.out.println("collection: " + collection);
}
```

So let's do this:

```
System.out.println("***** ArrayList list.removeIf():");
demoRemoveIf(new ArrayList<>(Arrays
                  .asList("One", "Two", "Three")));

System.out.println();
System.out.println("*****"
           + " CopyOnWriteArrayList list.removeIf():");
demoRemoveIf(new CopyOnWriteArrayList<>(Arrays
                  .asList("One", "Two", "Three")));
```

The result of the preceding code is as follows:

```
***** ArrayList list.removeIf():
collection: [One, Two, Three]
Calling list.removeIf(e -> Two.equals(e))...
collection: [One, Three]

***** CopyOnWriteArrayList list.removeIf():
collection: [One, Two, Three]
Calling list.removeIf(e -> Two.equals(e))...
collection: [One, Three]
```

It is short and has no problem with any of the collections and in line with the general trend of having a stateless parallel computation that uses streams with lambdas and functional interfaces.

Also, after we upgrade an application to use the `CopyOnWriteArrayList` class, we can take advantage of a simpler way of adding a new element to the list (without first checking whether it is already there):

```
CopyOnWriteArrayList<String> list =  new CopyOnWriteArrayList<>
                    (Arrays.asList("Five", "Six", "Seven"));
list.addIfAbsent("One");
```

With `CopyOnWriteArrayList`, this can be done as an atomic operation, so one does not need to synchronize this block of code: if-not-present-then-add.

2. Now let's review the concurrent collections of the `java.util.concurrent` package implementing the `Set` interface. There are three such implementations: `ConcurrentHashMap.KeySetView`, `CopyOnWriteArraySet` , and `ConcurrentSkipListSet`.

   The first one is just a view of the keys of `ConcurrentHashMap`. It is backed up by `ConcurrentHashMap` (can be retrieved by the `getMap()` method). We will review the behavior of `ConcurrentHashMap` later.

   The second implementation of `Set` in the `java.util.concurrent` package is the `CopyOnWriteArraySet` class. Its behavior is similar to the `CopyOnWriteArrayList` class. In fact, it uses the `CopyOnWriteArrayList` class's implementation under the hood. The only difference is that it does not allow duplicate elements in the collection.

   The third (and the last) implementation of `Set` in the `java.util.concurrent` package is `ConcurrentSkipListSet`; it implements a subinterface of `Set` called `NavigableSet`. According to the Javadoc of the `ConcurrentSkipListSet` class, insertion, removal, and access operations are safely executed concurrently by multiple threads. There are some limitations described in Javadoc too:

   - It does not permit the use of `null` elements.
   - The size of the set is calculated dynamically by traversing the collection, so it may report inaccurate results if this collection is modified during the operation.
   - The operations `addAll()`, `removeIf()`, or `forEach()` are not guaranteed to be performed atomically. The `forEach()` operation, if concurrent with an `addAll()` operation for example, *might observe only some of the added elements* (as stated in the Javadoc).

   The implementation of class `ConcurrentSkipListSet` is based on the `ConcurrentSkipListMap` class, which we will discuss shortly. To demonstrate the behavior of the `ConcurrentSkipListSet` class, let's compare it with the `java.util.TreeSet` class (non-concurrent implementation of `NavigableSet`). We start with removing an element:

```
void demoNavigableSetRemove(NavigableSet<Integer> set) {
  System.out.println("set: " + set);
  try {
    for (int i : set) {
      System.out.println(i);
      System.out.println("Calling set.remove(2)...");
      set.remove(2);
    }
  } catch (Exception ex) {
    System.out.println(ex.getClass().getName());
  }
  System.out.println("set: " + set);
}
```

Of course, this code is not very efficient; we've removed the same element many times without checking its presence. We have done this just for demo purposes. Besides, since Java 8, the same method `removeIf()` works for `Set` just fine. But we would like to bring up the behavior of the new class `ConcurrentSkipListSet`, so let's execute this code:

```
System.out.println("***** TreeSet set.remove(2):");
demoNavigableSetRemove(new TreeSet<>(Arrays
                            .asList(0, 1, 2, 3)));

System.out.println();
System.out.println("*****"
                   + " ConcurrentSkipListSet set.remove(2):");
demoNavigableSetRemove(new ConcurrentSkipListSet<>(Arrays
                             .asList(0, 1, 2, 3)));
```

The output will be as follows:

```
***** TreeSet set.remove(2):
set: [0, 1, 2, 3]
0
Calling set.remove(2)...
java.util.ConcurrentModificationException
set: [0, 1, 3]

***** ConcurrentSkipListSet set.remove(2):
set: [0, 1, 2, 3]
0
Calling set.remove(2)...
1
Calling set.remove(2)...
3
Calling set.remove(2)...
set: [0, 1, 3]
```

As expected, the `ConcurrentSkipListSet` class handles the concurrency and even removes an element from the current set, which is helpful.

It also removes an element via an iterator without an exception. Consider the following code:

```
void demoNavigableSetIterRemove(NavigableSet<Integer> set){
  System.out.println("set: " + set);
  try {
    Iterator iter = set.iterator();
    while (iter.hasNext()) {
      Integer e = (Integer) iter.next();
      System.out.println(e);
      if (e == 2) {
        System.out.println("Calling iter.remove()...");
        iter.remove();
      }
    }
  } catch (Exception ex) {
    System.out.println(ex.getClass().getName());
  }
  System.out.println("set: " + set);
}
```

Run this for `TreeSet` and `ConcurrentSkipListSet`:

```
System.out.println("***** TreeSet iter.remove():");
demoNavigableSetIterRemove(new TreeSet<>(Arrays
                            .asList(0, 1, 2, 3)));

System.out.println();
System.out.println("*****"
                   + " ConcurrentSkipListSet iter.remove():");
demoNavigableSetIterRemove(new ConcurrentSkipListSet<>(
                         Arrays.asList(0, 1, 2, 3)));
```

We'll not get any exception:

```
***** TreeSet iter.remove():
set: [0, 1, 2, 3]
0
1
2
Calling iter.remove()...
3
set: [0, 1, 3]

***** ConcurrentSkipListSet iter.remove():
set: [0, 1, 2, 3]
0
1
2
Calling iter.remove()...
3
set: [0, 1, 3]
```

This is because, according to the Javadoc, the iterator of `ConcurrentSkipListSet` is weakly consistent, which means the following (according to Javadoc):

- They may proceed concurrently with other operations
- They will never throw `ConcurrentModificationException`
- They are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction (from Javadoc).

This not guaranteed part is somewhat disappointing, but it is better than getting an exception, like with `CopyOnWriteArrayList`.

Adding to a `Set` class is not as problematic as to a `List` class because `Set` does not allow duplicates and handles the necessary checks internally:

```
void demoNavigableSetAdd(NavigableSet<Integer> set) {
  System.out.println("set: " + set);
  try {
    int m = set.stream().max(Comparator.naturalOrder())
                        .get() + 1;
    for (int i : set) {
      System.out.println(i);
      System.out.println("Calling set.add(" + m + ")");
      set.add(m++);
      if (m > 6) {
        break;
      }
    }
  } catch (Exception ex) {
    System.out.println(ex.getClass().getName());
  }
  System.out.println("set: " + set);
}
```

Consider the following code:

```
System.out.println("***** TreeSet set.add():");
demoNavigableSetAdd(new TreeSet<>(Arrays
                              .asList(0, 1, 2, 3)));

System.out.println();
System.out.println("*****"
                   + " ConcurrentSkipListSet set.add():");
```

```
    demoNavigableSetAdd(new ConcurrentSkipListSet<>(Arrays
                                    .asList(0,1,2,3)));
```

If we run this, we'll get the following result:

```
***** TreeSet set.add():
set: [0, 1, 2, 3]
0
Calling set.add(4)
java.util.ConcurrentModificationException
set: [0, 1, 2, 3, 4]

***** ConcurrentSkipListSet set.add():
set: [0, 1, 2, 3]
0
Calling set.add(4)
1
Calling set.add(5)
2
Calling set.add(6)
set: [0, 1, 2, 3, 4, 5, 6]
```

As before, we observe that the concurrent `Set` version handles concurrency better.

3. Now we turn to the `Map` interface that has two implementations in the java.util.concurrent package: `ConcurrentHashMap` and `ConcurrentSkipListMap`.

   The `ConcurrentHashMap` class *supports full concurrency of retrievals and high concurrency for updates* (from Javadoc). It is a thread-safe version of `java.util.HashMap` and is analogous to `java.util.Hashtable` in this respect. In fact, the `ConcurrentHashMap` class meets the requirements of the same functional specification as `java.util.Hashtable`, although its implementation is *somewhat different in synchronization details* (from Javadoc).

   Unlike `java.util.HashMap` and `java.util.Hashtable`, `ConcurrentHashMap` supports, according its JavaDoc, *a set of sequential and parallel bulk operations that, unlike most Stream methods, are designed to be safely, and often sensibly, applied even with maps that are being concurrently updated by other threads*:

   - `forEach()`: This performs a given action on each element

- `search()`: This returns the first available non-null result of applying a given function to each element
- `reduce()`: This accumulates each element (there are five overloaded versions)

These bulk operations accept a `parallelismThreshold` argument that allows deferring parallelization until the map size reaches the specified threshold. Naturally, when the threshold is set to `Long.MAX_VALUE`, there will be no parallelism whatsoever.

There are many other methods in the class API, so refer to its Javadoc for an overview.

Unlike `java.util.HashMap` (and similar to `java.util.Hashtable`), neither `ConcurrentHashMap` nor `ConcurrentSkipListMap` allow null to be used as a key or value.

The second implementation of `Map`--the `ConcurrentSkipListSet` class--is based, as we mentioned before, on the `ConcurrentSkipListMap` class, so all the limitations of the `ConcurrentSkipListSet` class we just described apply to the `ConcurrentSkipListMap` class too.
The `ConcurrentSkipListSet` class is practically a thread-safe version of `java.util.TreeMap`. `SkipList` is a sorted data structure that allows fast search concurrently. All the elements are sorted based on their natural sorting order of keys. The `NavigableSet` functionality we demonstrated for the `ConcurrentSkipListSet` class is present in the `ConcurrentSkipListMap` class too. For many other methods in the class API, refer to its Javadoc.

Now let's demonstrate the difference in the behavior in response to concurrency between the `java.util.HashMap`, `ConcurrentHashMap`, and `ConcurrentSkipListMap` classes. First, we will write the method that generates a test `Map` object:

```
Map createhMap() {
  Map<Integer, String> map = new HashMap<>();
  map.put(0, "Zero");
  map.put(1, "One");
  map.put(2, "Two");
```

```
    map.put(3, "Three");
    return map;
  }
```

Here is the code that adds an element to a `Map` object concurrently:

```
  void demoMapPut(Map<Integer, String> map) {
    System.out.println("map: " + map);
    try {
      Set<Integer> keys = map.keySet();
      for (int i : keys) {
        System.out.println(i);
        System.out.println("Calling map.put(8, Eight)...");
        map.put(8, "Eight");

        System.out.println("map: " + map);
        System.out.println("Calling map.put(8, Eight)...");
        map.put(8, "Eight");

        System.out.println("map: " + map);
        System.out.println("Calling"
                           + " map.putIfAbsent(9, Nine)...");
        map.putIfAbsent(9, "Nine");

        System.out.println("map: " + map);
        System.out.println("Calling"
                           + " map.putIfAbsent(9, Nine)...");
        map.putIfAbsent(9, "Nine");

        System.out.println("keys.size(): " + keys.size());
        System.out.println("map: " + map);
      }
    } catch (Exception ex) {
      System.out.println(ex.getClass().getName());
    }
  }
```

Run this for all three implementations of `Map`:

```
System.out.println("***** HashMap map.put():");
demoMapPut(createhMap());

System.out.println();
System.out.println("***** ConcurrentHashMap map.put():");
demoMapPut(new ConcurrentHashMap(createhMap()));

System.out.println();
System.out.println("*****"
                   + " ConcurrentSkipListMap map.put():");
demoMapPut(new ConcurrentSkipListMap(createhMap()));
```

If we do this, we get an output for `HashMap` for the first key only:

```
***** HashMap map.put():
map: {0=Zero, 1=One, 2=Two, 3=Three}
0
Calling map.put(8, Eight)...
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight}
Calling map.put(8, Eight)...
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight}
Calling map.putIfAbsent(9, Nine)...
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight, 9=Nine}
Calling map.putIfAbsent(9, Nine)...
keys.size(): 6
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight, 9=Nine}
java.util.ConcurrentModificationException
```

We also get an output for `ConcurrentHashMap` and `ConcurrentSkipListMap` for all the keys, including the newly added ones. Here is the last section of the `ConcurrentHashMap` output:

```
keys.size(): 6
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight, 9=Nine}
9
Calling map.put(8, Eight)...
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight, 9=Nine}
Calling map.put(8, Eight)...
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight, 9=Nine}
Calling map.putIfAbsent(9, Nine)...
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight, 9=Nine}
Calling map.putIfAbsent(9, Nine)...
keys.size(): 6
map: {0=Zero, 1=One, 2=Two, 3=Three, 8=Eight, 9=Nine}
```

As mentioned already, the appearance of `ConcurrentModificationException` is not guaranteed. Now we see that the moment it is thrown (if at all) is the moment when the code discovers that the modification has taken place. In the case of our example, it happened on the next iteration. Another point worth noticing is that the current set of keys changes even as we sort of isolate the set in a separate variable:

```
Set<Integer> keys = map.keySet();
```

This reminds us not to dismiss the changes propagated through the objects via their references.

To save the book space and your time, we will not show the code for concurrent removal and just summarize the results. As expected, `HashMap` throws exception `ConcurrentModificationException` when an

element is removed in any of the following ways. Here's the first way:

```
String result = map.remove(2);
```

Here is the second way:

```
boolean success = map.remove(2, "Two");
```

It allows concurrent removal via `Iterator` in two ways. Here's the first way:

```
iter.remove();
```

And here is the second way:

```
boolean result = map.keySet().remove(2);
```

Here's the third way:

```
boolean result = map.keySet().removeIf(e -> e == 2);
```

By contrast, the two concurrent `Map` implementations allow any of the above ways of removal concurrently.

Similar behavior is also exhibited by all the concurrent implementations of the `Queue` interface: `LinkedTransferQueue`, `LinkedBlockingQueue`, `LinkedBlockingDequeue`, `ArrayBlockingQueue`, `PriorityBlockingQueue`, `DelayQueue`, `SynchronousQueue`, `ConcurrentLinkedQueue`, and `ConcurrentLinkedDequeue`, all in the `java.util.concurrent` package. But to demonstrate all of them would require a separate volume, so we leave it up to you to browse the Javadoc and provide an example of `ArrayBlockingQueue` only. The queue will be represented by the `QueueElement` class:

```
class QueueElement {
  private String value;
  public QueueElement(String value){
    this.value = value;
  }
  public String getValue() {
    return value;
```

```
    }
  }
```

The queue producer will be as follows:

```
class QueueProducer implements Runnable {
  int intervalMs, consumersCount;
  private BlockingQueue<QueueElement> queue;
  public QueueProducer(int intervalMs, int consumersCount,
                       BlockingQueue<QueueElement> queue) {
    this.consumersCount = consumersCount;
    this.intervalMs = intervalMs;
    this.queue = queue;
  }
  public void run() {
    List<String> list =
       List.of("One","Two","Three","Four","Five");
    try {
      for (String e : list) {
        Thread.sleep(intervalMs);
        queue.put(new QueueElement(e));
        System.out.println(e + " produced" );
      }
      for(int i = 0; i < consumersCount; i++){
        queue.put(new QueueElement("Stop"));
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
   }
 }
```

The following will be the queue consumer:

```
class QueueConsumer implements Runnable{
  private String name;
  private int intervalMs;
  private BlockingQueue<QueueElement> queue;
  public QueueConsumer(String name, int intervalMs,
                       BlockingQueue<QueueElement> queue){
     this.intervalMs = intervalMs;
     this.queue = queue;
     this.name = name;
  }
  public void run() {
    try {
      while(true){
        String value = queue.take().getValue();
        if("Stop".equals(value)){
          break;
        }
        System.out.println(value + " consumed by " + name);
        Thread.sleep(intervalMs);
      }
    } catch(InterruptedException e) {
      e.printStackTrace();
    }
```

```
    }
}
```

Run the following code:

```
BlockingQueue<QueueElement> queue =
              new ArrayBlockingQueue<>(5);
QueueProducer producer = new QueueProducer(queue);
QueueConsumer consumer = new QueueConsumer(queue);
new Thread(producer).start();
new Thread(consumer).start();
```

Its results may look like this:

```
One produced
One consumed by Second
Two produced
Two consumed by First
Three produced
Three consumed by First
Four produced
Four consumed by Second
Five produced
Five consumed by First
```

How it works...

Before we select which collections to use, read the Javadoc and see whether the limitations of the collection are acceptable for your application.

For example, as per the Javadoc, the `CopyOnWriteArrayList` class *is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don't want to synchronize traversals, yet need to preclude interference among concurrent threads.* Use it when you do not need to add new elements at different positions and do not require sorting. Otherwise, use `ConcurrentSkipListSet`.

The `ConcurrentSkipListSet` and `ConcurrentSkipListMap` classes, as per the Javadoc, *provide expected average log(n) time cost for the contains, add, and remove operations and their variants. Ascending ordered views and their iterators are faster than descending ones.* Use them when you need to iterate quickly through the elements in a certain order and prefer sorting by default.

Use `ConcurrentHashMap` when the concurrency requirements are very demanding and you need to allow locking on the write operation but do not need to lock the element.

`ConcurrentLinkedQueue` and `ConcurrentLinkedDeque` are an appropriate choice when many threads share access to a common collection. `ConcurrentLinkedQueue` employs an efficient non-blocking algorithm.

`PriorityBlockingQueue` is a better choice when natural order is acceptable and you need fast adding of elements to the tail and fast removing of elements from the head of the queue. Blocking means that the queue waits to become non-empty when retrieving an element and waits for space to become available in the queue when storing an element.

`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `LinkedBlockingDeque` have a fixed size (bounded). The other queues are unbounded.

Use these and similar characteristics and recommendations as the guidelines but execute comprehensive testing and performance measuring before and after implementing your functionality.

See also

Refer to the following recipes in this chapter:

- Using the executor service to execute async tasks
- Using fork/join to implement divide-and-conquer
- Using flow to implement the publish-subscribe pattern

Using the executor service to execute async tasks

In this recipe, you will learn how to use `ExecutorService` to implement controllable thread execution.

Getting ready

In an earlier recipe, we demonstrated how to create and execute threads using the `Thread` class directly. It is an acceptable mechanism for a small number of threads that run and produce results predictably quickly. For big-scale applications with longer running threads with complex logic (which might keep them alive for an unpredictably long time) and/or a number of threads growing unpredictably too, a simple create-and-run-until-exit approach might result in an `OutOfMemory` error or require a complex customized system of threads' status maintenance and management. For such cases, `ExecutorService` and related classes of the `java.util.concurrent` package provide an out-of-the-box solution that relieves a programmer of the need to write and maintain a lot of infrastructural code.

At the foundation of the Executor Framework lies an `Executor` interface that has only one `void execute(Runnable command)` method that executes the given command at some time in the future.

Its subinterface `ExecutorService` adds methods that allow you to manage the executor:

- The `invokeAny()`, `invokeAll()`, and `awaitTermination()` methods and `submit()` allow you to define how the threads will be executed and if they are expected to return some values or not
- The `shutdown()` and `shutdownNow()` methods allow you to shut down the executor
- The `isShutdown()` and `isTerminated()` methods provide the status of the executor

The objects of `ExecutorService` can be created with the static factory methods of the `java.util.concurrent.Executors` class:

- `newSingleThreadExecutor()` - This creates an `Executor` method that uses a single worker thread operating off an unbounded queue. It has an

overloaded version with `ThreadFactory` as a parameter.

- `newCachedThreadPool()` - This creates a thread pool that creates new threads as needed, but reuses previously constructed threads when they are available. It has an overloaded version with `ThreadFactory` as a parameter.
- `newFixedThreadPool(int nThreads)` - This creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. It has an overloaded version with `ThreadFactory` as a parameter.

The `ThreadFactory` implementation allows you to override the process of creating new threads, enabling applications to use special thread subclasses, priorities, and so on. A demonstration of its usage is outside the scope of this book.

How to do it...

1. One important aspect of the behavior of the `Executor` interface you need to remember is that once created, it keeps running (waiting for new tasks to execute) until the Java process is stopped. So, if you would like to free memory, the `Executor` interface has to be stopped explicitly. If not shut down, forgotten executors will create a memory leak. Here is one possible way to make sure no executor is left behind:

```
int shutdownDelaySec = 1;
ExecutorService execService =
               Executors.newSingleThreadExecutor();
Runnable runnable =  () -> System.out.println("Worker One did
                                                the job.");
execService.execute(runnable);
runnable =   () -> System.out.println("Worker Two did the
                                       job.");
Future future = execService.submit(runnable);
try {
  execService.shutdown();
  execService.awaitTermination(shutdownDelaySec,
                               TimeUnit.SECONDS);
} catch (Exception ex) {
  System.out.println("Caught around"
          + " execService.awaitTermination(): "
          + ex.getClass().getName());
} finally {
  if (!execService.isTerminated()) {
    if (future != null && !future.isDone()
                       && !future.isCancelled()){
      System.out.println("Cancelling the task...");
      future.cancel(true);
    }
  }
  List<Runnable> l = execService.shutdownNow();
  System.out.println(l.size()
         + " tasks were waiting to be executed."
         + " Service stopped.");
}
```

You can pass a worker (an implementation of either the `Runnable` or `Callable` functional interface) for execution to `ExecutorService` in a variety of ways, which we will see shortly. In this example, we executed two threads: one using the `execute()` method and another using the `submit()` method. Both methods accept `Runnable` or `Callable`, but we used only `Runnable` in this example.

The `submit()` method returns `Future`, which represents the result of an asynchronous computation.

The `shutdown()` method initiates an orderly shutdown of the previously submitted tasks and prevents any new task from being accepted. This method does not wait for the task to complete the execution. The `awaitTermination()` method does that. But after `shutdownDelaySec`, it stops blocking and the code flow gets into `finally` block, where the `isTerminated()` method returns `true` if all the tasks are completed following the shutdown. In this example, we have two tasks executed in two different statements, but note that other methods of `ExecutorService` accept a collection of tasks.

In such a case, when the service is shutting down, we iterate over the collection of `Future` objects. We call each task and cancel it if it is not completed yet, possibly doing something else that had to be done before canceling the task. How much time to wait (value of `shutdownDelaySec`) has to be tested for each application and the possible running tasks.

Finally, the `shutdownNow()` method says this: *attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution* (according to the Javadoc).

2. Now we can collect and assess the results. In a real application, we typically do not want to shut down a service often. We just check the status of the tasks and collect results of those that return true from the `isDone()` method. In the above code example, we just show how to make sure that when we do stop the service, we do it in a controlled manner, without leaving behind any runaway process. If we run that code example, we will get the following:

```
Worker One did the job.
Worker Two did the job.
0 tasks were waiting to be executed. Service stopped.
```

3. Generalize the preceding code and create a method that shuts down a service and the task that has returned `Future`:

```
void shutdownAndCancelTask(ExecutorService execService,
          int shutdownDelaySec, String name, Future future) {
  try {
    execService.shutdown();
    System.out.println("Waiting for " + shutdownDelaySec
                + " sec before shutting down service...");
    execService.awaitTermination(shutdownDelaySec,
                                 TimeUnit.SECONDS);
  } catch (Exception ex) {
    System.out.println("Caught around"
                + " execService.awaitTermination():"
                + ex.getClass().getName());
  } finally {
    if (!execService.isTerminated()) {
      System.out.println("Terminating remaining tasks...");
      if (future != null && !future.isDone()
                         && !future.isCancelled()) {
        System.out.println("Cancelling task "
                           + name + "...");
        future.cancel(true);
      }
    }
    System.out.println("Calling execService.shutdownNow("
                       + name + ")...");
    List<Runnable> l = execService.shutdownNow();
    System.out.println(l.size() + " tasks were waiting"
                + " to be executed. Service stopped.");
  }
}
```

4. Enhance the example by making the `Runnable` (using lambda expression) sleep for some time (simulating useful work to be done):

```
void executeAndSubmit(ExecutorService execService,
            int shutdownDelaySec, int threadSleepsSec) {
  System.out.println("shutdownDelaySec = "
                     + shutdownDelaySec + ", threadSleepsSec = "
                     + threadSleepsSec);
  Runnable runnable = () -> {
    try {
      Thread.sleep(threadSleepsSec * 1000);
      System.out.println("Worker One did the job.");
    } catch (Exception ex) {
      System.out.println("Caught around One Thread.sleep(): "
                         + ex.getClass().getName());
    }
  };
  execService.execute(runnable);
  runnable = () -> {
    try {
      Thread.sleep(threadSleepsSec * 1000);
      System.out.println("Worker Two did the job.");
    } catch (Exception ex) {
```

```
            System.out.println("Caught around Two Thread.sleep(): "
                               + ex.getClass().getName());
          }
        };
        Future future = execService.submit(runnable);
        shutdownAndCancelTask(execService, shutdownDelaySec,
                              "Two", future);
    }
```

Notice the two parameters: `shutdownDelaySec` (that defines how long the service will wait without allowing new tasks to be submitted before moving on and shutting itself down, eventually) and `threadSleepSec` (that defines how long the worker is sleeping, indicating that the simulating process is doing its job).

5. Run the new code for different implementations of `ExecutorService` and values of `shutdownDelaySec` and `threadSleepSec`:

```
System.out.println("Executors.newSingleThreadExecutor():");
ExecutorService execService =
                Executors.newSingleThreadExecutor();
executeAndSubmit(execService, 3, 1);

System.out.println();
System.out.println("Executors.newCachedThreadPool():");
execService = Executors.newCachedThreadPool();
executeAndSubmit(execService, 3, 1);

System.out.println();
int poolSize = 3;
System.out.println("Executors.newFixedThreadPool("
                                        + poolSize + "):");
execService = Executors.newFixedThreadPool(poolSize);
executeAndSubmit(execService, 3, 1);
```

This is how the output may look like (it might be slightly different on your computer, depending on the exact timing of the events controlled by the operating system):

```
Executors.newSingleThreadExecutor():
shutdownDelaySec = 3, threadSleepsSec = 1
Waiting for 3 sec before shutting down service...
Worker One did the job.
Worker Two did the job.
Calling execService.shutdownNow(Two)...
0 tasks were waiting to be executed. Service stopped.
```

6. Analyze the results. In the first example, we find no surprise because of the following line:

```
    execService.awaitTermination(shutdownDelaySec,
                                 TimeUnit.SECONDS);
```

It is blocking for three seconds, whereas each worker works for one second only. So it is enough time for each worker to complete its work even for a single-thread executor.

Let's make the service wait for one second only:

```
Executors.newSingleThreadExecutor():
shutdownDelaySec = 1, threadSleepsSec = 3
Waiting for 1 sec before shutting down service...
Terminating remaining running tasks...
Cancelling task Two...
Calling execService.shutdownNow(Two)...
1 tasks were waiting to be executed. Service stopped.
Caught around One Thread.sleep(): java.lang.InterruptedException
```

When you do this, you will notice that none of the tasks will be completed. In this case, worker `One` was interrupted (see the last line of the output), while task `Two` was canceled.

Let's make the service wait for three seconds:

```
Executors.newSingleThreadExecutor():
shutdownDelaySec = 3, threadSleepsSec = 3
Waiting for 3 sec before shutting down service...
Worker One did the job.
Terminating remaining running tasks...
Cancelling task Two...
Calling execService.shutdownNow(Two)...
0 tasks were waiting to be executed. Service stopped.
Caught around Two Thread.sleep(): java.lang.InterruptedException
```

Now we see that worker `One` was able to complete its task, while worker `Two` was interrupted.

The `ExecutorService` interface produced by `newCachedThreadPool()` or `newFixedThreadPool()` performs similarly on a one-core computer. The only significant difference is that if the `shutdownDelaySec` value is equal to the `threadSleepSec` value, then they both allow you to complete the threads:

```
Executors.newCachedThreadPool():
shutdownDelaySec = 3, threadSleepsSec = 3
Waiting for 3 sec before shutting down service...
Worker One did the job.
Worker Two did the job.
Calling execService.shutdownNow(Two)...
0 tasks were waiting to be executed. Service stopped.
```

This was the result of using `newCachedThreadPool()`. The output of the example using `newFixedThreadPool()` looks exactly the same on a one-core computer.

7. Use the `Future` object as a returned value when you need more control over the task, not just submit one and wait. There is another method called `submit()` in the `ExecutorService` interface that allows you to not only return a `Future` object, but also include the result that is passed to the method as a second parameter in the return object. Let's check out an example of this:

```
Future<Integer> future = execService.submit(() ->
       System.out.println("Worker 42 did the job."), 42);
int result = future.get();
```

The value of `result` is `42`. This method can be helpful when you have submitted many workers (`nWorkers`) and need to know which one is completed:

```
Set<Integer> set = new HashSet<>();
while (set.size() < nWorkers){
  for (Future<Integer> future : futures) {
    if (future.isDone()){
      try {
        String id = future.get(1, TimeUnit.SECONDS);
        if(!set.contains(id)){
          System.out.println("Task " + id + " is done.");
          set.add(id);
        }
      } catch (Exception ex) {
        System.out.println("Caught around future.get(): "
                           + ex.getClass().getName());
      }
    }
  }
}
```

Well, the catch is that `future.get()` is a blocking method. This is why we use a version of the `get()` method that allows us to set the

`delaySec` timeout. Otherwise, `get()` blocks the iteration.

How it works...

Let's move a step closer to real-life code and create a class that implements `Callable` and allows you to return a result from a worker as an object of the `Result` class:

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

An actual numeric result is returned by the `getResult()` method. Here we also included the name of the worker and how long the thread is expected to sleep (to work) just for convenience and to better illustrate the output.

The worker itself is going to be an instance of the `CallableWorkerImpl` class:

```
class CallableWorkerImpl implements CallableWorker<Result>{
  private int sleepSec;
  private String name;
  public CallableWorkerImpl(String name, int sleepSec) {
    this.name = name;
    this.sleepSec = sleepSec;
  }
  public String getName() { return this.name; }
  public int getSleepSec() { return this.sleepSec; }
  public Result call() {
    try {
      Thread.sleep(sleepSec * 1000);
    } catch (Exception ex) {
      System.out.println("Caught in CallableWorker: "
                         + ex.getClass().getName());
    }
    return new Result(name, sleepSec, 42);
  }
}
```

Here, the number `42` is an actual numeric result, which a worker supposedly calculated (while sleeping). The class `CallableWorkerImpl` implemented

interface `CallableWorker` :

```
interface CallableWorker<Result> extends Callable<Result> {
  default String getName() { return "Anonymous"; }
  default int getSleepSec() { return 1; }
}
```

We had to make the methods default and return some data (they will be overridden by the class implementation anyway) to preserve its `functional interface` status. Otherwise, we would not be able to use it in lambda expressions.

We will also create a factory that will generate a list of workers:

```
List<CallableWorker<Result>> createListOfCallables(int nSec){
  return List.of(new CallableWorkerImpl("One", nSec),
                 new CallableWorkerImpl("Two", 2 * nSec),
                 new CallableWorkerImpl("Three", 3 * nSec));
}
```

Now we can use all these new classes and methods to demonstrate the `invokeAll()` method:

```
void invokeAllCallables(ExecutorService execService,
        int shutdownDelaySec, List<CallableWorker<Result>> callables) {
  List<Future<Result>> futures = new ArrayList<>();
  try {
    futures = execService.invokeAll(callables, shutdownDelaySec,
                                     TimeUnit.SECONDS);
  } catch (Exception ex) {
    System.out.println("Caught around execService.invokeAll(): "
                       + ex.getClass().getName());
  }
  try {
    execService.shutdown();
    System.out.println("Waiting for " + shutdownDelaySec
                       + " sec before terminating all tasks...");
    execService.awaitTermination(shutdownDelaySec,
                                 TimeUnit.SECONDS);
  } catch (Exception ex) {
    System.out.println("Caught around awaitTermination(): "
                       + ex.getClass().getName());
  } finally {
    if (!execService.isTerminated()) {
      System.out.println("Terminating remaining tasks...");
      for (Future<Result> future : futures) {
        if (!future.isDone() && !future.isCancelled()) {
          try {
            System.out.println("Cancelling task "
                       + future.get(shutdownDelaySec,
                                TimeUnit.SECONDS).getWorkerName());
            future.cancel(true);
```

```
          } catch (Exception ex) {
            System.out.println("Caught at cancelling task: "
                               + ex.getClass().getName());
          }
        }
      }
    }
    System.out.println("Calling execService.shutdownNow()...");
    execService.shutdownNow();
  }
  printResults(futures, shutdownDelaySec);
}
```

The `printResults()` method outputs the results received from the workers:

```
void printResults(List<Future<Result>> futures, int timeoutSec) {
  System.out.println("Results from futures:");
  if (futures == null || futures.size() == 0) {
    System.out.println("No results. Futures"
                       + (futures == null ? " = null" : ".size()=0"));
  } else {
    for (Future<Result> future : futures) {
      try {
        if (future.isCancelled()) {
          System.out.println("Worker is cancelled.");
        } else {
          Result result = future.get(timeoutSec, TimeUnit.SECONDS);
          System.out.println("Worker "+ result.getWorkerName() +
                             " slept " + result.getSleepSec() +
                             " sec. Result = " + result.getResult());
        }
      } catch (Exception ex) {
        System.out.println("Caught while getting result: "
                           + ex.getClass().getName());
      }
    }
  }
}
```

To get the results, again we use a version of the `get()` method with timeout settings. Run the following code:

```
List<CallableWorker<Result>> callables = createListOfCallables(1);
System.out.println("Executors.newSingleThreadExecutor():");
ExecutorService execService = Executors.newSingleThreadExecutor();
invokeAllCallables(execService, 1, callables);
```

Its output will be as follows:

```
Executors.newSingleThreadExecutor():
Waiting for 1 sec before terminating all tasks...
Calling execService.shutdownNow()...
Results from futures:
Worker is cancelled.
Worker is cancelled.
Worker is cancelled.
```

It's probably worth reminding that the three workers were created with sleep time 1, 2, and 3 seconds correspondingly, while the waiting time before the service shuts down is one second. This is why all the workers were canceled.

Now if we set the waiting time to six seconds, the output of the single-thread executor will be as follows:

```
Executors.newSingleThreadExecutor():
Waiting for 6 sec before terminating all tasks...
Caught in CallableWorkerImpl: java.lang.InterruptedException
Calling execService.shutdownNow()...
Results from futures:
Worker One slept 1 sec. Result = 42
Worker Two slept 2 sec. Result = 42
Worker is cancelled.
```

Naturally, if we increase the waiting time again, all the workers would be able to complete their tasks.

The `ExecutorService` interface produced by `newCachedThreadPool()` or `newFixedThreadPool()` performs much better even on a one-core computer:

```
Executors.newCachedThreadPool():
Waiting for 3 sec before terminating all tasks...
Calling execService.shutdownNow()...
Results from futures:
Worker One slept 1 sec. Result = 42
Worker Two slept 2 sec. Result = 42
Worker Three slept 3 sec. Result = 42
```

As you can see, all the threads were able to complete even with three seconds of waiting time.

As an alternative, instead of setting a timeout during the service shutdown, you can possibly set it on the overloaded version of the `invokeAll()` method:

```
List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                          long timeout, TimeUnit unit)
```

There is one particular aspect of the `invokeAll()` method's behavior that often gets overlooked and causes surprises for first-time users: it returns only after all the tasks are complete (either normally or by throwing an exception). Read the Javadoc and experiment until you recognize that this behavior is acceptable for your application.

By contrast, the `invokeAny()` method blocks only until at least one task is *completed successfully (without throwing an exception), if any do. Upon normal or exceptional return, tasks that have not completed are canceled* (according to Javadoc). Here is an example of the code that does this:

```
void invokeAnyCallables(ExecutorService execService,
        int shutdownDelaySec, List<CallableWorker<Result>> callables) {
  Result result = null;
  try {
    result = execService.invokeAny(callables, shutdownDelaySec,
TimeUnit.SECONDS);
  } catch (Exception ex) {
    System.out.println("Caught around execService.invokeAny(): "
                       + ex.getClass().getName());
  }
  shutdownAndCancelTasks(execService, shutdownDelaySec,
                         new ArrayList<>());
  if (result == null) {
    System.out.println("No result from execService.invokeAny()");
  } else {
    System.out.println("Worker " + result.getWorkerName() +
                       " slept " + result.getSleepSec() +
                       " sec. Result = " + result.getResult());
  }
}
```

You can experiment with it, setting different values for the waiting time (`shutdownDelaySec`) and sleep time for threads until you are comfortable with how this method behaves. As you can see, we have reused the `shutdownAndCancelTasks()` method by passing an empty list of `Future` objects since we do not have them in this case.

There's more...

There are two more static factory methods in the `Executors` class that create instances of `ExecutorService`:

- `newWorkStealingPool()`: This creates a work-stealing thread pool using the number of available processors as its target parallelism level. It has an overloaded version with a parallelism level as a parameter.
- `unconfigurableExecutorService(ExecutorService executor)`: This returns an object that delegates all the defined `ExecutorService` methods to the given executor, except for those methods that might otherwise be accessible using casts.

Also, a subinterface of the `ExecutorService` interface, called `ScheduledExecutorService`, enhances the API with the capability to schedule a thread execution in future and/or their periodic execution.

The objects of `ScheduledExecutorService` can be created using the static factory methods of the `java.util.concurrent.Executors` class too:

- `newSingleThreadScheduledExecutor()`: This creates a single-threaded executor that can schedule commands to run after a given delay or to execute them periodically. It has an overloaded version with `ThreadFactory` as a parameter.
- `newScheduledThreadPool(int corePoolSize)`: This creates a thread pool that can schedule commands to run after a given delay or to execute them periodically. It has an overloaded version with `ThreadFactory` as a parameter.
- `unconfigurableScheduledExecutorService( ScheduledExecutorService executor )`: This returns an object that delegates all the defined `ScheduledExecutorService` methods to the given executor, but not any other methods that might otherwise be accessible using casts.

The `Executors` class also has several overloaded methods that accept, execute, and return `Callable` (which, by contrast with `Runnable`, contains the result).

The `java.util.concurrent` package also includes classes that implement `ExecutorService`:

- **The ThreadPoolExecutor class**: This executes each submitted task using one of the several pooled threads, normally configured using the `Executors` factory methods.
- **The ScheduledThreadPoolExecutor class**: This extends the `ThreadPoolExecutor` class and implements the `ScheduledExecutorService` interface.
- **The ForkJoinPool class**: This manages the execution of workers (`ForkJoinTask` processes) using a work-stealing algorithm. We will discuss it in the next recipe.

Instances of these classes can be created via class constructors that accept more parameters, including the queue that holds the results, for providing more refined thread pool management.

See also

Refer to the following recipes in this chapter:

- Using fork/join to implement divide-and-conquer
- Using flow to implement the publish-subscribe pattern

Using fork/join to implement divide-and-conquer

In this recipe, you will learn how to use the fork/join framework for the divide-and-conquer computation pattern.

Getting ready

As mentioned in the previous recipe, the `ForkJoinPool` class is an implementation of the `ExecutorService` interface that manages the execution of workers--`ForkJoinTask` processes--using the work-stealing algorithm. It takes advantage of multiple processors, if available, and works best on tasks that can be broken down into smaller tasks recursively, which is also called a *divide-and-conquer* strategy.

Each thread in the pool has a dedicated double-ended queue (deque) that stores tasks, and the thread picks up the next task (from the head of the queue) as soon as the current task is completed. When another thread finishes executing all the tasks in its queue, it can take a task (steal it) from the tail of a non-empty queue of another thread.

As with any `ExecutorService` implementation, the fork/join framework distributes tasks to worker threads in a thread pool. This framework is distinct because it uses a work-stealing algorithm. Worker threads that run out of tasks can steal tasks from other threads that are still busy.

Such a design balances the load and allows an efficient use of the resources.

For demonstration purposes, we are going to use the API created in `Chapter 3`, *Modular Programming*: the `TrafficUnit`, `SpeedModel`, and `Vehicle` interfaces and the `TrafficUnitWrapper`, `FactoryTraffic`, `FactoryVehicle`, and `FactorySpeedModel` classes. We will also rely on the streams and stream pipelines described in `Chapter 3`, *Modular Programming*.

Just to refresh your memory, here is the `TrafficUnitWrapper` class:

```
class TrafficUnitWrapper {
  private double speed;
  private Vehicle vehicle;
```

```
  private TrafficUnit trafficUnit;
  public TrafficUnitWrapper(TrafficUnit trafficUnit){
    this.trafficUnit = trafficUnit;
    this.vehicle = FactoryVehicle.build(trafficUnit);
  }
  public TrafficUnitWrapper setSpeedModel(SpeedModel speedModel) {
    this.vehicle.setSpeedModel(speedModel);
    return this;
  }
  TrafficUnit getTrafficUnit(){ return this.trafficUnit;}
  public double getSpeed() { return speed; }

  public TrafficUnitWrapper calcSpeed(double timeSec) {
    double speed = this.vehicle.getSpeedMph(timeSec);
    this.speed = Math.round(speed * this.trafficUnit.getTraction());
    return this;
  }
}
```

We will also slightly modify the existing API interface and make it a bit more compact by introducing a new `DateLocation` class:

```
class DateLocation {
  private int hour;
  private Month month;
  private DayOfWeek dayOfWeek;
  private String country, city, trafficLight;

  public DateLocation(Month month, DayOfWeek dayOfWeek,
                      int hour, String country, String city,
                      String trafficLight) {
    this.hour = hour;
    this.month = month;
    this.dayOfWeek = dayOfWeek;
    this.country = country;
    this.city = city;
    this.trafficLight = trafficLight;
  }
  public int getHour() { return hour; }
  public Month getMonth() { return month; }
  public DayOfWeek getDayOfWeek() { return dayOfWeek; }
  public String getCountry() { return country; }
  public String getCity() { return city; }
  public String getTrafficLight() { return trafficLight;}
}
```

It will also allow you to hide the details and help you see the important aspects of this recipe.

How to do it...

All computations are encapsulated inside a subclass of one of the two subclasses (`RecursiveAction` or `RecursiveTask<T>`) of the abstract `ForkJoinTask` class. You can extend either `RecursiveAction` (and implement the `void compute()` method) or `RecursiveTask<T>` (and implement the `T compute()` method). As you may have probably noticed, you can choose to extend the `RecursiveAction` class for tasks that do not return any value, and extend `RecursiveTask<T>` when you need your tasks to return a value. In our demo, we are going to use the latter because it is slightly more complex.

Let's say we would like to calculate the average speed of traffic in a certain location on a certain date and time and driving conditions (all these parameters are defined by the property `DateLocation` object). Other parameters will be as follows:

- `timeSec`: The number of seconds during which the vehicles have a chance to accelerate after stopping at the traffic light
- `trafficUnitsNumber`: The number of vehicles to include in the average speed calculation

Naturally, the more vehicles included in the calculations, the better the prediction. But as this number increases, the number of calculations increases too. This gives rise to the need to break down the number of vehicles into smaller groups and compute the average speed of each group in parallel with the others. Yet, there is a certain minimal number of calculations that is not worth splitting between two threads. Here's what Javadoc has to say about it: *As a very rough rule of thumb, a task should perform more than 100 and less than 10000 basic computational steps, and should avoid indefinite looping. If tasks are too big, then parallelism cannot improve throughput. If too small, then memory and internal task maintenance overhead may overwhelm processing.* Yet, as always, the final answer about the best minimal number of calculations without splitting will come from testing. This is why we recommend to pass it as a parameter. We

will call this parameter `threshold`. Notice that it also serves as a criterium for exiting from the recursion.

We will call our class (task) `AverageSpeed` and extend `RecursiveTask<Double>` because we would like to have as a result of the average speed value of the `double` type:

```
class AverageSpeed extends RecursiveTask<Double> {
  private double timeSec;
  private DateLocation dateLocation;
  private int threshold, trafficUnitsNumber;
  public AverageSpeed(DateLocation dateLocation,
                      double timeSec, int trafficUnitsNumber,
                      int threshold) {
    this.timeSec = timeSec;
    this.threshold = threshold;
    this.dateLocation = dateLocation;
    this.trafficUnitsNumber = trafficUnitsNumber;
  }
  protected Double compute() {
    if (trafficUnitsNumber < threshold) {
      //... write the code here that calculates
      //... average speed trafficUnitsNumber vehicles
      return averageSpeed;
    } else{
      int tun = trafficUnitsNumber / 2;
      //write the code that creates two tasks, each
      //for calculating average speed of tun vehicles
      //then calculates an average of the two results
      double avrgSpeed1 = ...;
      double avrgSpeed2 = ...;
      return (double) Math.round((avrgSpeed1 + avrgSpeed2) / 2);
    }
  }
}
```

Before we finish writing the code for the `compute()` method, let's write the code that will execute this task. There are several ways to do this. We can use `fork()` and `join()`, for example:

```
void demo1_ForkJoin_fork_join() {
  AverageSpeed averageSpeed = createTask();
  averageSpeed.fork();
  double result = averageSpeed.join();
  System.out.println("result = " + result);
}
```

This technique provided the name for the framework. The `fork()` method, according to Javadoc, a*rranges to asynchronously execute this task in the pool the current task is running in, if applicable, or using the*

ForkJoinPool.commonPool() if not in ForkJoinPool(). In our case, we did not use any pool yet, so `fork()` is going to use `ForkJoinPool.commonPool()` by default. It places the task in the queue of a thread in the pool.
The `join()` method returns the result of the computation when it is done.

The `createTask()` method contains the following:

```
AverageSpeed createTask() {
  DateLocation dateLocation = new DateLocation(Month.APRIL,
        DayOfWeek.FRIDAY, 17, "USA", "Denver", "Main103S");
  double timeSec = 10d;
  int trafficUnitsNumber = 1001;
  int threshold = 100;
  return new AverageSpeed(dateLocation, timeSec,
                          trafficUnitsNumber, threshold);
}
```

Notice the values of the `trafficUnitsNumber` and `threshold` parameters. This will be important for analyzing the results.

Another way to accomplish this is to use either the `execute()` or `submit()` method--each providing the same functionality--for the execution of the task. The result of the execution can be retrieved by the `join()` method (the same as in the previous example):

```
void demo2_ForkJoin_execute_join() {
  AverageSpeed averageSpeed = createTask();
  ForkJoinPool commonPool = ForkJoinPool.commonPool();
  commonPool.execute(averageSpeed);
  double result = averageSpeed.join();
  System.out.println("result = " + result);
}
```

The last method we are going to review is `invoke()`, which is equivalent to calling the `fork()` method followed by the `join()` method:

```
void demo3_ForkJoin_invoke() {
  AverageSpeed averageSpeed = createTask();
  ForkJoinPool commonPool = ForkJoinPool.commonPool();
  double result = commonPool.invoke(averageSpeed);
  System.out.println("result = " + result);
}
```

Naturally, this is the most popular way to start the divide-and-conquer process.

Now let's get back to the `compute()` method and see how it can be implemented. First, let's implement the `if` block (that calculates the average speed of less than `threshold` vehicles). We will use the technique and code we described in Chapter 3, *Modular Programming*:

```
double speed = FactoryTraffic.getTrafficUnitStream(dateLocation,
                                    trafficUnitsNumber)
        .map(TrafficUnitWrapper::new)
        .map(tuw -> tuw.setSpeedModel(FactorySpeedModel.
                        generateSpeedModel(tuw.getTrafficUnit())))
        .map(tuw -> tuw.calcSpeed(timeSec))
        .mapToDouble(TrafficUnitWrapper::getSpeed)
        .average()
        .getAsDouble();
System.out.println("speed(" + trafficUnitsNumber + ") = " + speed);
return (double) Math.round(speed);
```

We get the `trafficUnitsNumber` of the vehicles from `FactoryTraffic`, and we create an object of `TrafficUnitWrapper` for each emitted element and call the `setSpeedModel()` method on it (by passing in the newly generated `SpeedModel` object, based on the emitted `TrafficUnit` object). Then we calculate the speed, get an average of all the speeds in the stream, and get the result as `double` from the `Optional` object (the return type of the `average()` operation). We then print out the result and round to get a more presentable format.

It is also possible to achieve the same result using a traditional `for` loop. But, as mentioned before, it seems that Java follows the general trend of more fluent and stream-like style, geared towards processing a large amount of data. So, we recommend you get used to it.

In Chapter 15, *Testing*, you will see another version of the same functionality that allows better unit testing of each step in isolation, which again supports the view that unit testing, along with writing code, helps you make your code more testable and decreases the need for rewriting it later.

Now, let's review the options of the `else` block implementation. The first few lines are always going to be the same:

```
int tun = trafficUnitsNumber / 2;
System.out.println("tun = " + tun);
AverageSpeed as1 = new AverageSpeed(dateLocation, timeSec, tun,
                                    threshold);
```

```
AverageSpeed as2 = new AverageSpeed(dateLocation, timeSec, tun,
                                    threshold);
```

We divide the `trafficUnitsNumber` number by two (we do not worry about possible loss of one unit in the case of an average across a big set) and create two tasks.

The following--the actual task execution code--can be written in several different ways. Here is the first possible solution, which is familiar to us already, that comes to mind:

```
as1.fork();                //add to the queue
double res1 = as1.join();  //wait until completed
as2.fork();
double res2 = as2.join();
return (double) Math.round((res1 + res2) / 2);
```

Run the following code:

```
demo1_ForkJoin_fork_join();
demo2_ForkJoin_execute_join();
demo3_ForkJoin_invoke();
```

If we do this, we will see the same output (but with different speed values) three times:

```
tun = 500
tun = 250
tun = 125
tun = 62
speed (62) = 18.548387096774192
speed (62) = 33.483870967741936
tun = 62
speed (62) = 28.532258064516128
speed (62) = 23.64516129032258
tun = 125
tun = 62
speed (62) = 29.306451612903224
speed (62) = 23.112903225806452
tun = 62
speed (62) = 24.919354838709676
speed (62) = 27.322580645161292
tun = 250
tun = 125
tun = 62
speed (62) = 29.112903225806452
speed (62) = 18.903225806451612
tun = 62
speed (62) = 23.193548387096776
speed (62) = 31.85483870967742
tun = 125
tun = 62
speed (62) = 29.451612903225808
speed (62) = 25.580645161290324
tun = 62
speed (62) = 27.14516129032258
speed (62) = 23.532258064516128
result = 27.0
```

You see how the original task of calculating average speed over 1,001 units (vehicles) was first divided by two several times until the number of one group (62) fell under the threshold of 100. Then, an average speed of the last two groups was calculated and combined (joined) with the results of other groups.

Another way to implement an `else` block of the `compute()` method could be as follows:

```
as1.fork();                    //add to the queue
double res1 = as2.compute();  //get the result recursively
double res2 = as1.join();     //wait until the queued task ends
return (double) Math.round((res1 + res2) / 2);
```

Here's how the result will look:

```
tun = 500
tun = 250
tun = 125
tun = 62
tun = 250
tun = 125
tun = 125
tun = 62
tun = 125
tun = 62
tun = 62
tun = 62
tun = 62
speed (62) = 22.774193548387096
speed (62) = 31.822580645161292
speed (62) = 27.758064516129032
tun = 62
speed (62) = 27.112903225806452
speed (62) = 28.887096774193548
speed (62) = 28.306451612903224
speed (62) = 26.35483870967742
tun = 62
speed (62) = 25.887096774193548
speed (62) = 29.596774193548388
speed (62) = 32.274193548387096
speed (62) = 32.37096774193548
speed (62) = 27.548387096774192
speed (62) = 22.0
speed (62) = 25.661290322580644
speed (62) = 21.161290322580644
speed (62) = 25.177419354838708
result = 28.0
```

You can see how, in this case, the `compute()` method (of the second task) was called recursively many times until it reached the threshold by the number of elements, then its results were joined with the results of the call to the `fork()` and `join()` methods of the first task.

As mentioned before, all this complexity can be replaced by a call to the `invoke()` method:

```
double res1 = as1.invoke();
double res2 = as2.invoke();
return (double) Math.round((res1 + res2) / 2);
```

It produces a result similar to the one produced by calling `fork()` and `join()` on each of the tasks:

```
tun = 500
tun = 250
tun = 125
tun = 62
speed (62) = 30.467741935483872
speed (62) = 17.14516129032258
tun = 62
speed (62) = 29.93548387096774
speed (62) = 27.70967741935484
tun = 125
tun = 62
speed (62) = 28.85483870967742
speed (62) = 33.45161290322581
tun = 62
speed (62) = 22.419354838709676
speed (62) = 35.645161290322584
tun = 250
tun = 125
tun = 62
speed (62) = 20.338709677419356
speed (62) = 34.064516129032256
tun = 62
speed (62) = 38.854838709677416
speed (62) = 22.387096774193548
tun = 125
tun = 62
speed (62) = 28.14516129032258
speed (62) = 35.725806451612904
tun = 62
speed (62) = 22.161290322580644
speed (62) = 23.20967741935484
result = 29.0
```

Yet, there is an even better way to implement an `else` block of the `compute()` method:

```
return ForkJoinTask.invokeAll(List.of(as1, as2))
        .stream()
        .mapToDouble(ForkJoinTask::join)
        .map(Math::round)
        .average()
        .getAsDouble();
```

If this looks complex to you, just notice that it is just a stream-like way to iterate over the results of `invokeAll()`:

```
<T extends ForkJoinTask> Collection<T> invokeAll(Collection<T> tasks)
```

It is also to iterate over the results of calling `join()` on each of the returned tasks (and combining the results into average). The advantage is that we yield to the framework to decide how to optimize the load distribution. The result is as follows:

```
tun = 500
tun = 250
tun = 250
tun = 125
tun = 125
tun = 62
tun = 62
tun = 125
tun = 62
tun = 62
speed (62) = 24.306451612903224
speed (62) = 19.161290322580644
tun = 125
tun = 62
speed (62) = 32.17741935483871
tun = 62
speed (62) = 21.661290322580644
speed (62) = 38.096774193548384
speed (62) = 28.14516129032258
speed (62) = 24.258064516129032
speed (62) = 40.88709677419355
speed (62) = 35.564516129032256
speed (62) = 33.03225806451613
tun = 62
speed (62) = 19.548387096774192
tun = 62
speed (62) = 30.387096774193548
speed (62) = 32.03225806451613
speed (62) = 25.5
speed (62) = 33.91935483870968
speed (62) = 17.20967741935484
result = 29.0
```

You can see it differs from any of the preceding results and can change depending on the availability and load of the CPUs on your computer.

Using flow to implement the publish-subscribe pattern

In this recipe, you will learn about the new publish-subscribe capability introduced in Java 9.

Getting ready

Among many other features, Java 9 introduced these four interfaces in the `java.util.concurrent.Flow` class:

```
Flow.Publisher<T> - producer of items (messages) of type T
Flow.Subscriber<T> - receiver of messages of type T
Flow.Subscription - links producer and receiver
Flow.Processor<T,R> - acts as both producer and receiver
```

With this, Java stepped into the world of reactive programming--programming with the asynchronous processing of data streams.

We discussed streams in `Chapter 3`, *Modular Programming* and pointed out that they are not data structures, as they do not keep data in memory. The stream pipeline does nothing until an element is emitted. Such a model allows minimal resource allocation and uses resources only as needed. The application behaves *in response* to the appearance of the data it reacts to, thus the name.

In a publish-subscribe pattern, the main two actors are a `Publisher` and a `Subscriber`. `Publisher` streams data (publishes), and `Subscriber` listens to data (subscribes).

The `Flow.Publisher<T>` interface is a functional interface. It only has one abstract method:

```
void subscribe(Flow.Subscriber<? super T> subscriber)
```

According to the Javadoc, this method *adds the given Flow.Subscriber<T> if possible. If already subscribed, or the attempt to subscribe fails, the onError() method of theFlow.Subscriber<T> is invoked with an IllegalStateException. Otherwise, the onSubscribe() method of theFlow.Subscriber<T> is invoked with a new Flow.Subscription. Subscribers may enable receiving items by invoking the request() method of thisFlow.Subscription and may unsubscribe by invoking its cancel() method.*

The `Flow.Subscriber<T>` interface has four methods; some of them were mentioned just now:

- `void onSubscribe(Flow.Subscription subscription)` is invoked prior to invoking any other `Subscriber` methods for the given `Subscription`
- `void onError(Throwable throwable)` is invoked upon an unrecoverable error encountered by a `Publisher` or `Subscription`, after which no other `Subscriber` methods are invoked by the `Subscription`
- `void onNext(T item)` is invoked with the `Subscription`'s next item
- `void onComplete()`: This method is invoked when it is known that no additional `Subscriber` method invocations will occur for a `Subscription`

The `Flow.Subscription` interface has two methods:

- `void cancel()`: This method causes the `Subscriber` to (eventually) stop receiving messages
- `void request(long n)`: This method adds the given n number of items to the current unfulfilled demand for this subscription

The `Flow.Processor<T,R>` interface is outside the scope of this book.

How to do it...

To save some time and space, instead of creating our own implementation of the `Flow.Publisher<T>` interface, we can use the `SubmissionPublisher<T>` class from the `java.util.concurrent` package. But, we will create our own implementation of the `Flow.Subscriber<T>` interface:

```
class DemoSubscriber<T> implements Flow.Subscriber<T> {
  private String name;
  private Flow.Subscription subscription;
  public DemoSubscriber(String name){ this.name = name; }
  public void onSubscribe(Flow.Subscription subscription) {
    this.subscription = subscription;
    this.subscription.request(0);
  }
  public void onNext(T item) {
    System.out.println(name + " received: " + item);
    this.subscription.request(1);
  }
  public void onError(Throwable ex){ ex.printStackTrace();}
  public void onComplete() { System.out.println("Completed"); }
}
```

We will also implement the `Flow.Subscription` interface:

```
class DemoSubscription<T> implements Flow.Subscription {
  private final Flow.Subscriber<T> subscriber;
  private final ExecutorService executor;
  private Future<?> future;
  private T item;
  public DemoSubscription(Flow.Subscriber subscriber,
                          ExecutorService executor) {
    this.subscriber = subscriber;
    this.executor = executor;
  }
  public void request(long n) {
    future = executor.submit(() -> {
      this.subscriber.onNext(item );
    });
  }
  public synchronized void cancel() {
    if (future != null && !future.isCancelled()) {
      this.future.cancel(true);
    }
  }
}
```

As you can see, we just followed Javadoc recommendations and expect the `onSubscribe()` method of a subscriber to be called when the subscriber is added to a publisher.

Another detail to notice is that the `SubmissionPublisher<T>` class has the `submit(T item)` method that, according to Javadoc, *publishes the given item to each current subscriber by asynchronously invoking its onNext() method, blocking uninterruptibly while resources for any subscriber are unavailable*. This way, the `SubmissionPublisher<T>` class submits items to the current subscribers until it is closed. This allows item generators to act as reactive-streams publishers.

To demonstrate this, let's create several subscribers and subscriptions using the `demoSubscribe()` method:

```
void demoSubscribe(SubmissionPublisher<Integer> publisher,
        ExecutorService execService, String subscriberName){
  DemoSubscriber<Integer> subscriber =
                     new DemoSubscriber<>(subscriberName);
  DemoSubscription subscription =
           new DemoSubscription(subscriber, execService);
  subscriber.onSubscribe(subscription);
  publisher.subscribe(subscriber);
}
```

Then use them in the following code:

```
ExecutorService execService =  ForkJoinPool.commonPool();
try (SubmissionPublisher<Integer> publisher =
                           new SubmissionPublisher<>()){
  demoSubscribe(publisher, execService, "One");
  demoSubscribe(publisher, execService, "Two");
  demoSubscribe(publisher, execService, "Three");
  IntStream.range(1, 5).forEach(publisher::submit);
} finally {
  //...make sure that execService is shut down
}
```

The preceding code creates three subscribers, connected to the same publisher with a dedicated subscription. The last line generates a stream of numbers 1, 2, 3, and 4 and submits each of them to the publisher. We expect that every subscriber will get each of the generated numbers as the parameter of the `onNext()` method.

In the `finally` block, we included the code you are already familiar with from the previous recipe:

```
try {
  execService.shutdown();
  int shutdownDelaySec = 1;
  System.out.println("Waiting for " + shutdownDelaySec
                     + " sec before shutting down service...");
  execService.awaitTermination(shutdownDelaySec, TimeUnit.SECONDS);
} catch (Exception ex) {
  System.out.println("Caught around execService.awaitTermination(): "
                     + ex.getClass().getName());
} finally {
  System.out.println("Calling execService.shutdownNow()...");
  List<Runnable> l = execService.shutdownNow();
  System.out.println(l.size()
              +" tasks were waiting to be executed. Service stopped.");
}
```

If we run the preceding code, the output may look like the following:

```
Waiting for 1 sec before shutting down service...
Three received: null
Two received: null
One received: null
Three received: 1
One received: 1
Two received: 1
One received: 2
Three received: 2
One received: 3
Two received: 2
One received: 4
Three received: 3
Completed
Two received: 3
Three received: 4
Two received: 4
Completed
Completed
Calling execService.shutdownNow()...
0 tasks were waiting to be executed. Service stopped.
```

As you can see, because of asynchronous processing, the control gets to the `finally` block very quickly and waits for 1 sec before shutting down the service. This period of waiting is enough for the items to be generated and passed to the subscribers. We also confirmed that every generated item was sent to each of the subscribers. The three `null` values were generated every time the `onSubscribe()` method of each of the subscribers is called.

It is reasonable to expect that in future Java releases, there will be more support added for reactive (asynchronous and non-blocking) functionality.

Better Management of the OS Process

In this chapter, we will cover the following recipes:

- Spawning a new process
- Redirecting the process output and error streams to file
- Changing the working directory of a subprocess
- Setting the environment variable for a subprocess
- Running shell scripts
- Obtaining the process information of the current JVM
- Obtaining the process information of the spawned process
- Managing the spawned process
- Enumerating live processes in the system
- Connecting multiple processes using pipe
- Managing subprocesses

Introduction

How often have you ended up writing code that spawns a new process? Not often. However, there would be situations that ask for writing such code. In such cases, you resort to using a third-party API such as Apache Commons Exec (`https://commons.apache.org/proper/commons-exec/`) and the like. Why was this? Wasn't the Java API sufficient? No, it wasn't, at least until Java 9. Now with Java 9, we have quite a few more features added to the process API.

Until Java 7, redirecting the input, output, and error streams was not trivial. With Java 7, there were new APIs introduced, which allowed the redirecting of the input, output, and error to other processes (pipe), to a file, or to a standard input/output. Then in Java 8, there were a few more APIs introduced. In Java 9, there are new APIs for the following areas:

- Get the process information, such as **process ID** (**PID**), the user who launched the process, the time it has been running for, and so on
- Enumerate the processes running in the system
- Manage the subprocesses and get access to the process tree by navigating up the process hierarchy

In this chapter, we will look at a few recipes that will help you explore the new things in the process API, and you will also get to know the changes that have been introduced from the times of `Runtime.getRuntime().exec()`. And you all know that using that was a crime.



All the recipes can only be executed on the Linux platform because we will be using Linux-specific commands while spawning a new process from Java code. There are two ways to execute the script `run.sh` on Linux:

- `sh run.sh`
- `chmod +x run.sh && ./run.sh`

Spawning a new process

In this recipe, we will see how to spawn a new process using `ProcessBuilder`. We will also see how to make use of the input, output, and error streams. This should be a very straightforward and common recipe. However, the aim of introducing this is to make this chapter a bit more complete and not just to focus on Java 9 features.

Getting ready

There is a command in Linux called `free`, which shows the amount of RAM free and used by the system. It accepts an option, `-m`, to show the output in megabytes. So, just running free `-m` gives us the following output:

```
root@ubuntu-512mb-lon1-01:~/samples/java9-samples/chp8/2_redirect_to_file# free -m
              total        used        free      shared  buff/cache   available
Mem:            488         208          17           5         261         243
Swap:             0           0           0
```

We will be running the preceding code from within the Java program.

How to do it...

1. Create an instance of `ProcessBuilder` by providing the required command and its options:

```
ProcessBuilder pBuilder = new ProcessBuilder("free", "-m");
```

   An alternate way to specify the command and options is as follows:

```
pBuilder.command("free", "-m");
```

2. Set up the input and output streams for the process builder and other properties, such as the directory of execution and environment variables. After that, invoke `start()` on the `ProcessBuilder` instance to spawn the process and get a reference to the `Process` object:

```
Process p = pBuilder.inheritIO().start();
```

   The `inheritIO()` function sets the standard I/O of the spawned subprocess to be the same as that of the current Java process.

3. We then wait for the completion of the process, or for 1 second, whichever is sooner, as shown in the following code:

```
if(p.waitFor(1, TimeUnit.SECONDS)){
  System.out.println("process completed successfully");
}else{
  System.out.println("waiting time elapsed, process did
                      not complete");
  System.out.println("destroying process forcibly");
  p.destroyForcibly();
}
```

   If it doesn't complete in the time specified, then we kill the process by invoking the `destroyForcibly()` method.

4. Compile and run the code by using the following commands:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java -p mods -m process/com.packt.process.NewProcessDemo
```

5. The output we get is as follows:

```
              total        used        free      shared  buff/cache   available
Mem:            487          96          30           5         361         370
Swap:             0           0           0
process completed successfully
```

The code for this recipe can be found at `chp8/1_spawn_new_process`.

How it works...

There are two ways to let `ProcessBuilder` know which command to run:

1. By passing the command and its options to the constructor while creating the `ProcessBuilder` object.
2. By passing the command and its options as parameters to the `command()` method of the `ProcessBuilder` object.

Before spawning the process, we can do the following:

- We can change the directory of execution by using the `directory()` method
- We can redirect the input stream, output stream, and error streams to file or to another process
- We can provide the required environment variables for the subprocess

We will see all these activities in their respective recipes in this chapter.

A new process is spawned when the `start()` method is invoked and the caller gets a reference to this subprocess in the form of an instance of the `Process` class. Using this `Process` object, we can do a lot of things, such as the following:

- Get information about the process, including its PID
- Get the output and error streams
- Check for the completion of the process
- Destroy the process
- Associate the tasks to be performed once the process completes
- Check for the subprocesses spawned by the process
- Find the parent process of the process if it exists

In our recipe, we `waitFor` 1 second or the completion of the process (whichever occurs first). If the process has completed, then `waitFor` returns

`true`; else, it returns `false`. If the process doesn't complete, we can kill the process by invoking the `destroyForcibly()` method on the `Process` object.

Redirecting the process output and error streams to file

In this recipe, we will see how to deal with the output and error streams of a process spawned from the Java code. We will write the output or error produced by the spawned process to a file.

Getting ready

In this recipe, we will make use of the `iostat` command. This command is used for reporting the CPU and I/O statistics for different devices and partitions. Let's run the command and see what it reports:

```
$ iostat
```



In some Linux distributions, such as Ubuntu, `iostat` is not installed by default. You can install the utility by running `sudo apt-get install sysstat`.

The output of the preceding command is as follows:

```
Linux 4.8.0-26-generic (ubuntu-512mb-blr1-01)   12/22/2016      _x86_64_        (1 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          0.17    0.00    0.06    0.01    0.00   99.76

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
vda               0.32         1.79         3.95    7844239   17323645
```

How to do it...

1. Create a new `ProcessBuilder` object by specifying the command to be executed:

```
ProcessBuilder pb = new ProcessBuilder("iostat");
```

2. Redirect the output and error streams to the file's output and error, respectively:

```
pb.redirectError(new File("error"))
  .redirectOutput(new File("output"));
```

3. Start the process, and wait for it to complete:

```
Process p = pb.start();
int exitValue = p.waitFor();
```

4. Read the content of the output file:

```
Files.lines(Paths.get("output"))
                .forEach(l -> System.out.println(l));
```

5. Read the content of the error file. This is created only if there is some error in the command:

```
Files.lines(Paths.get("error"))
                .forEach(l -> System.out.println(l));
```



Step 4 and 5 are for our reference. This has nothing to do with `ProcessBuilder` or the process spawned. Using these two lines of code, we can inspect what was written to the output and error files by the process.

The complete code can be found at `chp8/2_redirect_to_file`.

6. Compile the code by using the following command:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
```

7. Run the code by using the following command:

```
$ java -p mods -m process/com.packt.process.RedirectFileDemo
```

We will get the following output:

```
Output
Linux 4.8.0-26-generic (ubuntu-512mb-blr1-01)   12/22/2016      _x86_64_        (1 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          0.17    0.00    0.06    0.01    0.00   99.76

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
vda               0.32         1.79         3.95    7874939   17335285

Error
```

We can see that as the command executed successfully; there is nothing in the error file.

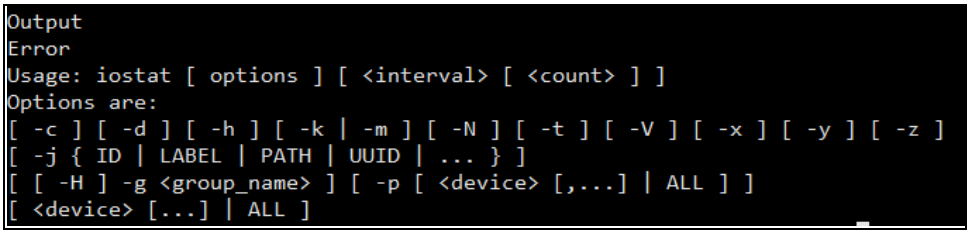
There is more...

You can provide an erroneous command to `ProcessBuilder` and then see the error get written to the error file and nothing in the output file. You could do this by changing the `ProcessBuilder` instance creation as follows:

```
ProcessBuilder pb = new ProcessBuilder("iostat", "-Z");
```

Compile and run by using the commands given earlier in the *How to do it...* section.

You will see that there is an error reported in the error file but nothing in the output file:



Changing the working directory of a subprocess

Often, you would want a process to be executed in the context of some path, such as listing the files in some directory. In order to do so, we will have to tell `ProcessBuilder` to launch the process in the context of a given location. We can achieve this by using the `directory()` method. This method serves two purposes:

1. Returns the current directory of execution when we don't pass any parameter.
2. Sets the current directory of execution to the passed value when we pass a parameter.

In this recipe, we will see how to execute the `tree` command to recursively traverse all the directories from the current directory and print it in the form of a tree.

Getting ready

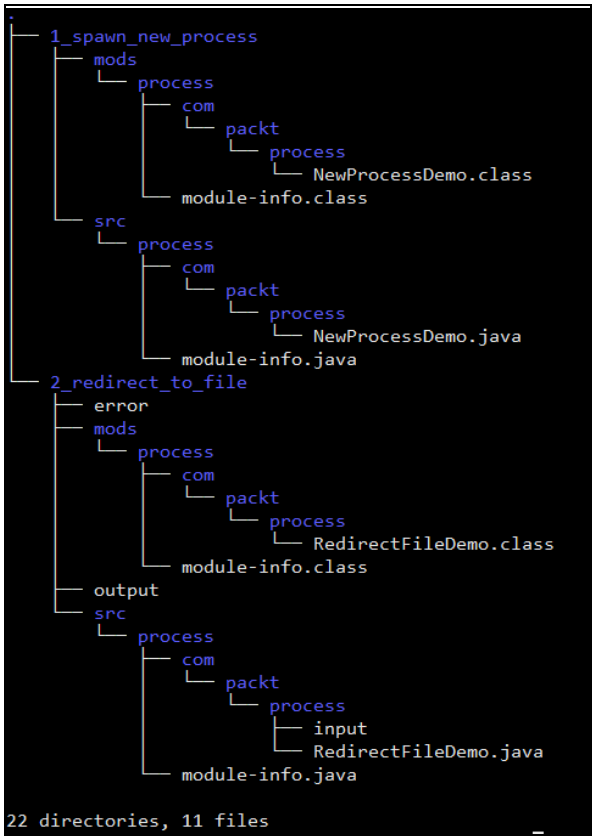
Generally, the `tree` command doesn't come preinstalled. So, you will have to install the package that contains the command. To install on an Ubuntu/Debian-based system, run the following command:

```
sudo apt-get install tree
```

To install on Linux, which supports the `yum` package manager, run the following command:

```
yum install tree
```

To verify your installation, just run the `tree` command, and you should be able to see the current directory structure printed. For me, it's something like this:



There are multiple options supported by the `tree` command. It's for you to explore.

How to do it...

1. Create a new `ProcessBuilder` object:

```
ProcessBuilder pb = new ProcessBuilder();
```

2. Set the command to `tree` and the output and error to the same as that of the current Java process:

```
pb.command("tree").inheritIO();
```

3. Set the directory to whatever directory you want. I set it as the root folder:

```
pb.directory(new File("/root"));
```

4. Start the process and wait for it to exit:

```
Process p = pb.start();
int exitValue = p.waitFor();
```

5. Compile and run using the following commands:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java -p mods -m process/com.packt.process.ChangeWorkDirectoryDemo
```

6. The output will be the recursive contents of the directory, specified in the `directory()` method of the `ProcessBuilder` object, printed in a tree like format.

The complete code can be found at `chp8/2_redirect_to_file`.

How it works...

The `directory()` method accepts the path of the working directory for `Process`. The path is specified as an instance of `File`.

Setting the environment variable for a subprocess

Environment variables are just like any other variables that we have in our programming languages. They have a name and hold some value, which can be varied. These are used by the Linux/Windows commands or the shell/batch scripts to perform different operations. These are called environment variables because they are present in the environment of the process/command/script getting executed. Generally, the process inherits the environment variables from the parent process.

They are accessed in different ways in different operating systems. In Windows, they are accessed as `%ENVIRONMENT_VARIABLE_NAME%`, and in Unix-based operating systems, they are accessed as `$ENVIRONMENT_VARIABLE_NAME`.

In Unix-based systems, you can use the `printenv` command to print all the environment variables available for the process, and in Windows-based systems, you can use the `SET` command.

In this recipe, we will pass some environment variables to our subprocess and make use of the `printenv` command to print all the environment variables available.

How to do it...

1. Create an instance of `ProcessBuilder`:

```
ProcessBuilder pb = new ProcessBuilder();
```

2. Set the command to `printenv` and the output and error streams to the same as that of the current Java process:

```
pb.command("printenv").inheritIO();
```

3. Provide the environment variables, `COOKBOOK_VAR1` with the value, `First variable`, `COOKBOOK_VAR2` with the value, `Second variable`, and `COOKBOOK_VAR3` with the value, `Third variable`:

```
Map<String, String> environment = pb.environment();
environment.put("COOKBOOK_VAR1", "First variable");
environment.put("COOKBOOK_VAR2", "Second variable");
environment.put("COOKBOOK_VAR3", "Third variable");
```

4. Start the process and wait for it to complete:

```
Process p = pb.start();
int exitValue = p.waitFor();
```

   The complete code for this recipe can be found at `chp8/4_environment_variables`.

5. Compile and run the code by using the following commands:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java -p mods -m process/com.packt.process.EnvironmentVariableDemo
```

The output you get is as follows:

```
XDG_SESSION_ID=2412
MAIL=/var/mail/root
COOKBOOK_VAR1=First variable
COOKBOOK_VAR2=Second variable
LOGNAME=root
COOKBOOK_VAR3=Third variable
PWD=/root/java9-samples/chp8/4_environment_variables
```

You can see the three variables printed among other variables.

How it works...

When you invoke the `environment()` method on the instance of `ProcessBuilder`, it copies the environment variables of the current process, populates them in an instance of `HashMap`, and returns it to the caller code.



All the work of loading the environment variables is done by a package private final class, `ProcessEnvironment`, which actually extends `HashMap`.

We then make use of this map to populate our own environment variables, but we need not set the map back to `ProcessBuilder` because we will have a reference to the map object and not a copy. Any changes made to the map object will reflect in the actual map object held by the `ProcessBuilder` instance.

Running shell scripts

We generally collect a set of commands used in performing an operation in a file, called a shell script in the Unix world and batch file in Windows. The commands present in these files are executed sequentially, with the exceptions being when you have conditional blocks or loops in the scripts.

These shell scripts are evaluated by the shell in which they get executed. Different types of shells available are `bash`, `csh`, `ksh`, and so on. The `bash` shell is the most commonly used shell.

In this recipe, we will write a simple shell script and then invoke the same from the Java code using the `ProcessBuilder` and `Process` objects.

Getting ready

First, let's write our shell script. This script does the following:

1. Print the value of the environment variable, `MY_VARIABLE`.
2. Execute the `tree` command.
3. Execute the `iostat` command.

Let's create a shell script file by the name, `script.sh`, with the following commands in it:

```
echo $MY_VARIABLE;
echo "Running tree command";
tree;
echo "Running iostat command"
iostat;
```

You can place the `script.sh` in your home folder that is, in the `/home/<username>`. Now let's see how we can execute this from Java.

How to do it...

1. Create a new instance of `ProcessBuilder`:

```
ProcessBuilder pb = new ProcessBuilder();
```

2. Set the directory of execution to point to the directory of the shell script file:

```
pb.directory(new File("/root"));
```



Note that the above path passed while creating the `File` object will depend on where you have place your script `script.sh`. In our case we had it placed in `/root`. You might have copied the script in `/home/yourname` and accordingly the `File` object will be created as `new File("/home/yourname")`.

3. Set an environment variable that would be used by the shell script:

```
Map<String, String> environment = pb.environment();
environment.put("MY_VARIABLE", "From your parent Java process");
```

4. Set the command to be executed and also the arguments to be passed to the command. Also, set the output and error streams for the process to same as that of the current Java process:

```
pb.command("/bin/bash", "script.sh").inheritIO();
```

5. Start the process, and wait for it to execute completely:

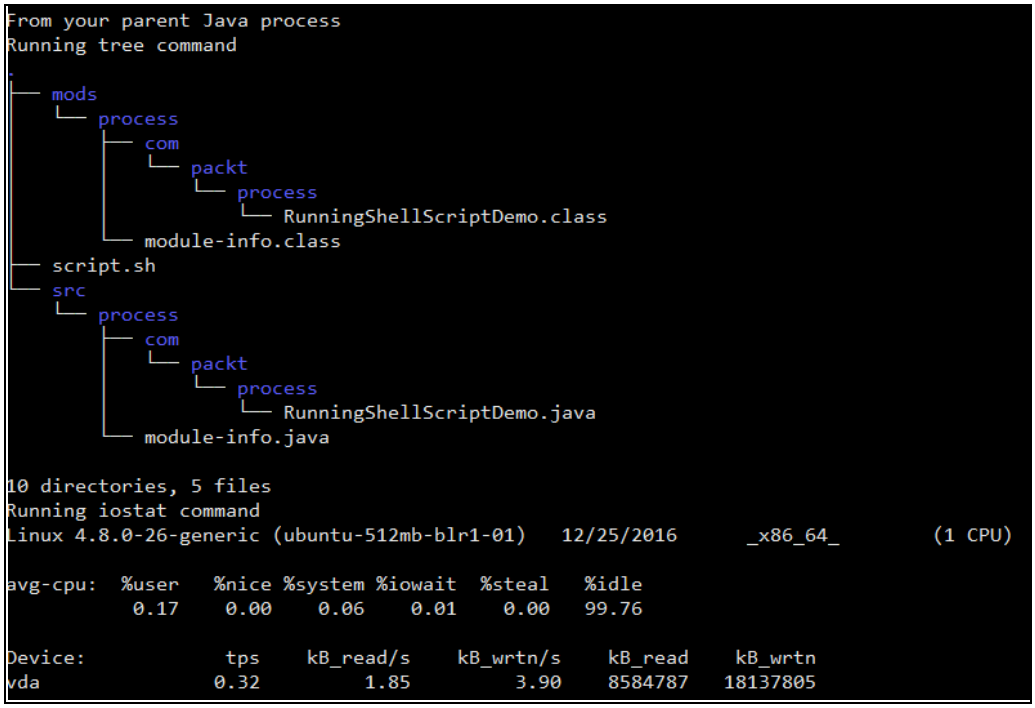
```
Process p = pb.start();
int exitValue = p.waitFor();
```

You can get the complete code from `chp8/5_running_shell_script`.

You can compile and run the code by using the following commands:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java -p mods -m process/com.packt.process.RunningShellScriptDemo
```

The output we get is as follows:



How it works...

You must make note of two things in this recipe:

1. Change the working directory of the process to the location of the shell script.
2. Use `/bin/bash` to execute the shell script.

If you don't do step 1, then you'll have to use the absolute path for the shell script file. However, in the recipe, we did do step 1, and hence, we just use the shell script name for the `/bin/bash` command.

Step 2 is basically how you would want to execute the shell script. The way to do so is to pass the shell script to the interpreter, which will interpret and execute the script. That is what the following line of code does:

```
pb.command("/bin/bash", "script.sh")
```

Obtaining the process information of the current JVM

A running process has a set of attributes associated with it, such as the following:

- **PID**: This uniquely identifies the process
- **Owner**: This is the name of the user who launched the process
- **Command**: This is the command that runs under the process
- **CPU time**: This indicates the time for which the process has been active
- **Start time**: This indicates the time when the process was launched

These are a few attributes that we are generally interested in. Perhaps, we would also be interested in CPU usage or memory usage. Now, getting this information from within Java was not possible prior to Java 9. However, in Java 9, a new set of APIs have been introduced, which enables us to get the basic information about the process.

In this recipe, we will see how to get the process information for the current Java process, that is, the process that is executing your code.

How to do it...

1. Create a simple class and use `ProcessHandle.current()` to get `ProcessHandle` for the current Java process:

```
ProcessHandle handle = ProcessHandle.current();
```

2. We have added some code, which will add some running time to the code:

```
for ( int i = 0 ; i < 100; i++){
  Thread.sleep(1000);
}
```

3. Use the `info()` method on the instance of `ProcessHandle` to get an instance of `ProcessHandle.Info`:

```
ProcessHandle.Info info = handle.info();
```

4. Use the instance of `ProcessHandle.Info` to get all the information made available by the interface:

```
System.out.println("Command line: " + info.commandLine().get());
System.out.println("Command: " + info.command().get());
System.out.println("Arguments: " +
            String.join(" ", info.arguments().get()));
System.out.println("User: " + info.user().get());
System.out.println("Start: " + info.startInstant().get());
System.out.println("Total CPU Duration: " +
            info.totalCpuDuration().get().toMillis() +"ms");
```

5. Use the `pid()` method of `ProcessHandle` to get the process ID of the current Java process:

```
System.out.println("PID: " + handle.pid());
```

6. We will also print the end time using the time at which the code is about to end. This will give us an idea of the execution time of the process:

```
Instant end = Instant.now();
System.out.println("End: " + end);
```

You can get the complete code from `chp8/6_current_process_info`.

Compile and run the code by using the following commands:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java -p mods -m process/com.packt.process.CurrentProcessInfoDemo
```

The output you see will be something like this:

```
Started main...
Command line: /usr/lib/jdk-9-b146/jdk-9/bin/java -p mods -m process/com.packt.process.CurrentProcessInfoDemo
Command: /usr/lib/jdk-9-b146/jdk-9/bin/java
Arguments: -p mods -m process/com.packt.process.CurrentProcessInfoDemo
User: root
Start: 2016-12-28T21:05:20.490Z
Total CPU Duration: 350ms
PID: 13487
End: 2016-12-28T21:07:00.995953Z
```



It will take some time till the program completes execution. One observation to be made is that even if the program ran for around 2 minutes, the total CPU duration was 350ms. This is the time period during which the CPU was busy.

How it works...

To give more control to the native processes and get its information, a new interface called `ProcessHandle` has been added to the Java API. Using `ProcessHandle`, you can control the process execution as well as get some information about the process. The interface has another inner interface called `ProcessHandle.Info`. This interface provides APIs to get information about the process.

There are multiple ways to get hold of the `ProcessHandle` object for a process. Some of the ways are as follows:

- `ProcessHandle.current()`: This is used to get the `ProcessHandle` instance for the current Java process
- `Process.toHandle()`: This is used to get `ProcessHandle` for a given `Process` object
- `ProcessHandle.of(pid)`: This is used to get `ProcessHandle` for a process identified by the given PID

In our recipe, we make use of the first approach, that is, we use `ProcessHandle.current()`. This gives us a handle on the current Java process. Invoking the `info()` method on the `ProcessHandle` instance will give us an instance of the implementation of the `ProcessHandle.Info` interface, which we can make use of to get the process information, as shown in the recipe code.

> *`ProcessHandle` and `ProcessHandle.Info` are interfaces. The JDK provider, that is, Oracle JDK or Open JDK, will provide implementations for these interfaces. Oracle JDK has a class called `ProcessHandleImpl`, which implements `ProcessHandle` and another inner class within `ProcessHandleImpl` called `Info`, which implements the `ProcessHandle.Info` interface. So, whenever you call one of the aforementioned methods to get a `ProcessHandle` object, an instance of `ProcessHandleImpl` is returned.*



The same goes with the `Process` class as well. It is an abstract class and Oracle JDK provides an implementation called `ProcessImpl`, which implements the abstract methods in the `Process` class.

In all the recipes in this chapter, any mention of the instance of `ProcessHandle` or the `ProcessHandle` object will refer to the instance or object of `ProcessHandleImpl` or any other implementation class provided by the JDK you are using.

Also, any mention of the instance of `ProcessHandle.Info` or the `ProcessHandle.Info` object will refer to the instance or object of `ProcessHandleImpl.Info` or any other implementation class provided by the JDK you are using.

Obtaining the process information of the spawned process

In our previous recipe, we saw how to get the process information for the current Java process. In this recipe, we will look at how to get the process information for a process spawned by the Java code, that is, by the current Java process. The APIs used will be the same as we saw in the previous recipe, except for the way the instance of `ProcessHandle` is implemented.

Getting ready

In this recipe, we will make use of a Unix command, `sleep`, which is used to pause the execution for a period of time in seconds.

How to do it...

1. Spawn a new process from the Java code, which runs the `sleep` command:

```
ProcessBuilder pBuilder = new ProcessBuilder("sleep", "20");
Process p = pBuilder.inheritIO().start();
```

2. Get the `ProcessHandle` instance for this spawned process:

```
ProcessHandle handle = p.toHandle();
```

3. Wait for the spawned process to complete execution:

```
int exitValue = p.waitFor();
```

4. Use `ProcessHandle` to get the `ProcessHandle.Info` instance and use its APIs to get the required information. Alternatively, we can even use the `Process` object directly to get `ProcessHandle.Info` by using the `info()` method in the `Process` class:

```
ProcessHandle.Info info = handle.info();

System.out.println("Command line: " + info.commandLine().get());
System.out.println("Command: " + info.command().get());
System.out.println("Arguments: " + String.join(" ",
                   info.arguments().get()));
System.out.println("User: " + info.user().get());
System.out.println("Start: " + info.startInstant().get());
System.out.println("Total CPU time(ms): " +
                   info.totalCpuDuration().get().toMillis());
System.out.println("PID: " + handle.pid());
```

You can get the complete code from `chp8/7_spawned_process_info`.

Compile and run the code by using the following commands:

```
$ javac -d mods --module-source-path src $(find src -name *.java)
$ java -p mods -m process/com.packt.process.SpawnedProcessInfoDemo
```

Alternatively, there is a `run.sh` script in `chp8\7_spawned_process_info`, which you can run from any Unix-based system as `/bin/bash run.sh`.

The output you see will be something like this:

```
Started main
Command line: /bin/sleep 20
Command: /bin/sleep
Arguments: 20
User: root
Start: 2016-12-28T21:18:17.130Z
Total CPU time(ms): 0
PID: 13589
```

Managing the spawned process

There are a few methods, such as `destroy()`, `destroyForcibly()` (added in Java 8), `isAlive()` (added in Java 8), and `supportsNormalTermination()` (added in Java 9), which can be used to control the process spawned. These methods are available on the `Process` object as well as on the `ProcessHandle` object. Here, controlling would be just to check if the process is alive and if yes, then destroy the process.

In this recipe, we will spawn a long running process and do the following:

- Check for its liveliness
- Check if it can be stopped normally, that is, depending on the platform, the process has to be stopped by just using destroy or by using force destroy
- Stop the process

How to do it...

1. Spawn a new process from the Java code, which runs the `sleep` command for say 1 minute, that is, 60 seconds:

```
ProcessBuilder pBuilder = new ProcessBuilder("sleep", "60");
Process p = pBuilder.inheritIO().start();
```

2. Wait for, say, 10 seconds:

```
p.waitFor(10, TimeUnit.SECONDS);
```

3. Check whether the process is alive:

```
boolean isAlive = p.isAlive();
System.out.println("Process alive? " + isAlive);
```

4. Check whether the process can be stopped normally:

```
boolean normalTermination = p.supportsNormalTermination();
System.out.println("Normal Termination? " + normalTermination);
```

5. Stop the process and check for its liveliness:

```
p.destroy();
isAlive = p.isAlive();
System.out.println("Process alive? " + isAlive);
```

You can get the complete code from `chp8\8_manage_spawned_process`.

We have provided a utility script called `run.sh`, which you can use to compile and run the code: `sh run.sh`.

The output we get is as follows:

```
Started main
Process alive? true
Normal Termination? true
Process alive? false
```



If we are running the program on Windows, `supportsNormalTermination()` *returns* `false`*, but on Unix* `supportsNormalTermination()` *returns* `true` *(as seen in the preceding output as well).*

Enumerating live processes in the system

In Windows, you open up the Windows Task Manger to view the processes currently active, and in Linux, you use the `ps` command with its varied options to view the processes along with other details, such as user, time spent, command, and so on.

In Java 9, a new API has been added, called `ProcessHandle`, which deals with controlling and getting information about the processes. One of the methods of the API is `allProcesses()`, which returns a snapshot of all the processes visible to the current process. In this recipe, we will look at how the method works and what information we can extract from the API.

How to do it...

1. Use the `allProcesses()` method on the `ProcessHandle` interface to get a stream of the currently active processes:

```
Stream<ProcessHandle> liveProcesses =
                ProcessHandle.allProcesses();
```

2. Iterate over the stream using `forEach()` and pass a lambda expression to print the details available:

```
liveProcesses.forEach(ph -> {
  ProcessHandle.Info phInfo = ph.info();
  System.out.println(phInfo.command().orElse("") +" " +
                     phInfo.user().orElse(""));
});
```

You can get the complete code from `chp8/9_enumerate_all_processes`.

We have provided a utility script called `run.sh`, which you can use to compile and run the code: `sh run.sh`.

The output we get is as follows:

```
/lib/systemd/systemd-timesyncd systemd-timesync
/lib/systemd/systemd-logind root
/usr/sbin/cron root
/lib/systemd/systemd-resolved systemd-resolve
/usr/sbin/rsyslogd syslog
/usr/bin/dbus-daemon (deleted) messagebus
/usr/sbin/atd daemon
/usr/lib/snapd/snapd root
/usr/bin/lxcfs root
/usr/sbin/acpid root
/usr/lib/accountsservice/accounts-daemon root
/sbin/iscsid root
/sbin/iscsid root
/sbin/agetty root
/sbin/agetty root
/usr/lib/policykit-1/polkitd root
/usr/sbin/sshd root
```

In the preceding output, we are printing the command name as well as the user of the process. We have shown a small part of the output.

Connecting multiple processes using pipe

In Unix, it's common to pipe a set of commands together using the `|` symbol to create a pipeline of activities, where the input for the command is the output from the previous command. This way, we can process the input to get the desired output.

A common scenario is when you want to search for something or some pattern in the log files or for an occurrence of some text in the log file. In such scenarios, you can create a pipeline, wherein you pass the required log file data via a series of commands, namely: `cat`, `grep`, `wc -l`, and so on.

In this recipe, we will make use of the Iris dataset from the UCI machine learning repository available at `https://archive.ics.uci.edu/ml/datasets/Iris` to create a pipeline, wherein we will count the number of occurrences of each type of flower.

Getting ready

We have already downloaded the Iris flower dataset, which can be found at `chp8/10_connecting_process_pipe/iris.data` of the code download for this book.

If you happen to look at the `Iris` data, you will see there are 150 rows of the following format:

```
4.7,3.2,1.3,0.2,Iris-setosa
```

Here, there are multiple attributes separated by a comma (`,`) and the attributes are as follows:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm
- Class:
    - Iris Setosa
    - Iris Versicolour
    - Iris Virginica

In this recipe, we will find the total number of flowers in each class, namely Setosa, Versicolour, and Virginica.

We will make use of a pipeline with the following commands (using a Unix-based operating system):

```
$ cat iris.data.txt | cut -d',' -f5 | uniq -c
```

The output we get is as follows:

```
50 Iris-setosa
50 Iris-versicolor
50 Iris-virginica
1
```

The 1 at the end is for the new line available at the end of the file. So, there are 50 flowers of each class. Let us dissect the above shell command pipeline and understand the function of each of them:

- `cat`: This command reads the file given as the argument
- `cut`: This splits each line by using the character given in the `-d` option and returns the value in the column identified by the `-f` option.
- `uniq`: This returns a unique list from the given values, and when the `-c` option is used, it returns how many times each unique value is present in the list

How to do it...

1. Create a list of `ProcessBuilder` objects, which will hold the `ProcessBuilder` instances participating in our pipeline. Also, redirect the output of the last process in the pipeline to the standard output of the current Java process:

```
List<ProcessBuilder> pipeline = List.of(
  new ProcessBuilder("cat", "iris.data.txt"),
  new ProcessBuilder("cut", "-d", ",", "-f", "5"),
  new ProcessBuilder("uniq", "-c")
      .redirectOutput(ProcessBuilder.Redirect.INHERIT)
);
```

2. Use the `startPipeline()` method of `ProcessBuilder` and pass the list of `ProcessBuilder` objects to start the pipeline. It will return a list of `Process` objects, each representing a `ProcessBuilder` object in the list:

```
List<Process> processes = ProcessBuilder.startPipeline(pipeline);
```

3. Get the last process in the list and `waitFor` it to complete:

```
int exitValue = processes.get(processes.size() - 1).waitFor();
```

You can get the complete code from `chp8/10_connecting_process_pipe`.

We have provided a utility script called `run.sh`, which you can use to compile and run the code: `sh run.sh`.

The output we get is as follows:

```
     50 Iris-setosa
     50 Iris-versicolor
     50 Iris-virginica
      1
```

How it works...

The `startPipeline()` method starts a `Process` for each `ProcessBuilder` object in the list. Except for the first and the last processes, it redirects the output of one process to the input of another process by using `ProcessBuilder.Redirect.PIPE`. If you have provided `redirectOutput` for any intermediate process as something other than `ProcessBuilder.Redirect.PIPE`, then there will be an error thrown, something similar to the following:

```
Exception in thread "main" java.lang.IllegalArgumentException: builder
redirectOutput() must be PIPE except for the last builder: INHERIT.
```

It states that any builder except for the last should redirect its output to the next process. The same is applicable for `redirectInput`.

Managing subprocesses

When a process launches another process, the launched process becomes the subprocess of the launching process. The launched process, in turn, can launch another process and this chain can continue. This results in a process tree. Often, we would have to deal with a buggy subprocess and might want to kill that subprocess, or we might want to know the subprocesses that are launched and might want to get some information about it.

In Java 9, two new APIs in the `Process` class have been added: `children()` and `descendants()`. The `children()` API allows you to get a list of the snapshot of processes that are the immediate children of the current process, and the `descendants()` API provides a snapshot of processes that are recursively `children()` of the current process, that is, they are invoking `children()` recursively on each child process.

In this recipe, we will look at both the `children()` and `descendants()` APIs and see what information we can gather from the snapshot of the process.

Getting ready

Let's create a simple shell script, which we will be using in the recipe. This script can be found at `chp8/11_managing_sub_process/script.sh`:

```
echo "Running tree command";
tree;
sleep 60;
echo "Running iostat command";
iostat;
```

In the preceding script, we are running the commands, `tree` and `iostat`, separated by a sleep time of 1 minute. If you want to know about these commands, please refer to the *Running shell scripts* recipe of this chapter. The sleep command, when executed from within the bash shell, creates a new subprocess each time it is invoked.

We will create, say, 10 instances of `ProcessBuilder` to run the preceding shell script and launch them simultaneously.

How to do it...

1. We will create 10 instances of `ProcessBuilder` to run our shell script (available at `chp8/11_managing_sub_process/script.sh`). We are not concerned with its output, so let's discard the output of the commands by redirecting the output to a predefined redirect called `ProcessHandle.Redirect.DISCARD`:

```
for ( int i = 0; i < 10; i++){
  new ProcessBuilder("/bin/bash", "script.sh")
      .redirectOutput(ProcessBuilder.Redirect.DISCARD)
      .start();
}
```

2. Get the handle for the current process:

```
ProcessHandle currentProcess = ProcessHandle.current();
```

3. Use the current process to get its children using the `children()` API and iterate over each of its children to print their information. Once we have an instance of `ProcessHandle`, we can do multiple things, such as destroy the process, get its process information, and so on:

```
System.out.println("Obtaining children");
currentProcess.children().forEach(pHandle -> {
  System.out.println(pHandle.info());
});
```

4. Use the current process to get all the subprocesses that are its descendants using the `descendants()` API and iterate over each of them to print their information:

```
currentProcess.descendants().forEach(pHandle -> {
  System.out.println(pHandle.info());
});
```

You can get the complete code from `chp8/11_managing_sub_process`.

We have provided a utility script called `run.sh`, which you can use to compile and run the code: `sh run.sh`.

The output we get is as follows:

```
Obtaining children
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.800Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.850Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.850Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.850Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
Obtaining descendants
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.800Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.850Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.850Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.850Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/bash, args: [script.sh], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.820Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.850Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.860Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.870Z], totalTime: Optional[PT0S]]
[user: Optional[root], cmd: /bin/sleep, args: [5], startTime: Optional[2017-01-07T08:05:15.880Z], totalTime: Optional[PT0S]]
```

How it works...

The APIs, `children()`, and `descendants()`, return `Stream` of `ProcessHandler` for each of the processes, which are either direct children or descendants of the current process. Using the instance of `ProcessHandler`, we can perform the following operations:

1. Get the process information.
2. Check the status of the process.
3. Stop the process.

GUI Programming Using JavaFX

In this chapter we will cover the following recipes:

- Creating a GUI using JavaFX controls
- Using the FXML markup to create a GUI
- Using CSS to style elements in JavaFX
- Creating a bar chart
- Creating a pie chart
- Creating a line chart
- Creating an area chart
- Creating a bubble chart
- Creating a scatter chart
- Embedding HTML in an application
- Embedding media in an application
- Adding effects to controls
- Using the new TIFF I/O API to read TIFF images

Introduction

GUI programming has been in Java since JDK 1.0, via the API called **Abstract Window Toolkit** (**AWT**). This was a remarkable thing during those times, but it had its own limitations, a few of which are as follows:

- It had a limited set of components
- You couldn't create custom reusable components because AWT was using native components
- The look and feel of the components couldn't be controlled and they took the look and feel of the host OS

Then, in Java 1.2, a new API for GUI development called **Swing** was introduced, which worked on the deficiencies of AWT by providing the following:

- A richer components library.
- Support for creating custom components.
- Native look and feel and support for plugging in a different look and feel. Some of the know Java look and feel themes are Nimbus, Metal, Motif, and the system default.

A lot of desktop applications that make use of Swing have been built, and a lot of them are still being used. However, with time, technology has to evolve; otherwise, it will eventually be outdated and seldom used. In 2008, Adobe's Flex started gaining attention. It was a framework for building **Rich Internet applications** (**RIAs**). The desktop applications were always rich component-based UIs but the web applications were not that amazing to use. Adobe introduced a framework called Flex, which enabled web developers to create rich, immersive UIs on the web. So the web applications were no longer boring.

Adobe also introduced a rich internet application runtime environment for the desktop called Adobe AIR, which allowed running Flex applications on

the desktop. This was a major blow to the age-old Swing API. In order to come back to the market, in 2009, Sun Microsystems introduced something called JavaFX. This framework was inspired by Flex (which used XML for defining the UI) and introduced its own scripting language called JavaFX script, somewhat closer to JSON and JavaScript. One could invoke Java APIs from the JavaFX script. There was a new architecture introduced, which had a new Windowing toolkit and a new graphics engine. It was a much better alternative to Swing but had a drawback--developers had to learn JavaFX script to develop JavaFX-based applications. In addition to Sun Microsystems not being able to invest more on JavaFX and the Java platform, in general, JavaFX never took off as envisioned.

Oracle (after acquiring Sun Microsystems) announced a new JavaFX version 2.0, which was an entire rewrite of JavaFX, thereby eliminating the scripting language and making JavaFX an API within the Java platform. This has made using the JavaFX API similar to using Swing APIs. Also, one can embed JavaFX components within Swing, thereby making Swing-based applications more functional. Since then, there has been no looking back for JavaFX.

In this chapter, we will focus entirely on the recipes around JavaFX. We will try to cover as many recipes as possible to give you all a good experience of using JavaFX.

Creating a GUI using JavaFX controls

In this recipe, we will look at creating a simple GUI application using JavaFX controls. We will build an app that will help you compute your age by providing your date of birth. Optionally, you can even enter your name and the app will greet you and display your age. It is a pretty simple example, which tries to show how you can create a GUI by using layouts, components, and event handling.

Getting ready

The JDK you have installed comes with the JavaFX modules, so there isn't anything needed to be done to start using JavaFX. Various modules that contain the JavaFX classes are as follows:

- `javafx.base`
- `javafx.controls`
- `javafx.fxml`
- `javafx.graphics`
- `javafx.media`
- `javafx.swing`
- `javafx.web`

In our recipe, we will be using a few modules as and when required from the preceding list.

How to do it...

1. Create a class, which extends the `javafx.application.Application`. The `Application` class manages the lifecycle of the JavaFX application. The `Application` class has an abstract method, `start(Stage stage)`, which one has to implement. This would be our starting point for our JavaFX UI:

```
public class CreateGuiDemo extends Application{
  public void start(Stage stage){
    //to implement in new steps
  }
}
```

   The class can also be the starting point for the application by providing a `public static void main(String [] args) {}` method:

```
public class CreateGuiDemo extends Application{
  public void start(Stage stage){
    //to implement in new steps
  }
  public static void main(String[] args){
    //launch the JavaFX application
  }
}
```

   The code for the subsequent steps has to be written within the `start(Stage stage)` method.

2. Let's create a container layout to properly align the components that we will be adding. In this case, we will use `javafx.scene.layout.GridPane` to lay the components in the form of a grid of rows and columns:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

   Along with creating the `GridPane` instance, we are setting its layout properties, such as the alignment of `GridPane`, the horizontal and vertical spaces between the rows and columns, and the padding within each cell of the grid.

3. Create a new label, which will show the name of our application, specifically, `Age calculator`, and add it to `gridPane`, which we created in the preceding step:

```
Text appTitle = new Text("Age calculator");
appTitle.setFont(Font.font("Arial", FontWeight.NORMAL, 15));
gridPane.add(appTitle, 0, 0, 2, 1);
```

4. Create a label and a text input combination, which will be used for accepting the user's name. Then add these two components to `gridPane`:

```
Label nameLbl = new Label("Name");
TextField nameField = new TextField();
gridPane.add(nameLbl, 0, 1);
gridPane.add(nameField, 1, 1);
```

5. Create a label and a date picker combination, which will be used for accepting the user's date of birth:

```
Label dobLbl = new Label("Date of birth");
gridPane.add(dobLbl, 0, 2);
DatePicker dateOfBirthPicker = new DatePicker();
gridPane.add(dateOfBirthPicker, 1, 2);
```

6. Create a button, which will be used by the user to trigger the age calculation, and add it to `gridPane`:

```
Button ageCalculator = new Button("Calculate");
gridPane.add(ageCalculator, 1, 3);
```

7. Create a component to hold the result of the computed age:

```
Text resultTxt = new Text();
resultTxt.setFont(Font.font("Arial", FontWeight.NORMAL, 15));
gridPane.add(resultTxt, 0, 5, 2, 1);
```

8. Now, we need to bind an action to the button created in step 6. The action would be to get the name entered in the name field and the date of birth entered in the date picker field. If the date of birth is provided, then use the Java time APIs to compute the period between now and the date of birth. If there is a name provided, then prepend a greeting, `Hello, <name>`, to the result:

```
ageCalculator.setOnAction((event) -> {
  String name = nameField.getText();
```

```
        LocalDate dob = dateOfBirthPicker.getValue();
        if ( dob != null ){
          LocalDate now = LocalDate.now();
          Period period = Period.between(dob, now);
          StringBuilder resultBuilder = new StringBuilder();
          if ( name != null && name.length() > 0 ){
            resultBuilder.append("Hello, ")
                         .append(name)
                         .append("n");
          }
          resultBuilder.append(String.format(
            "Your age is %d years %d months %d days",
            period.getYears(),
            period.getMonths(),
            period.getDays())
          );
          resultTxt.setText(resultBuilder.toString());
        }
      });
```

9. Create an instance of the `Scene` class by providing the `gridPane` object we created in step 2 and the dimensions, the width and height of the scene:

```
      Scene scene = new Scene(gridPane, 300, 250);
```

   An instance of `Scene` holds the graph of the UI components, called scene graph.

10. We have seen that the `start()` method provides us with a reference to a `Stage` object. The `Stage` object is the top-level container in JavaFX, something like a JFrame. We set the `Scene` object to the `Stage` object and use its `show()` method to render the UI:

```
      stage.setTitle("Age calculator");
      stage.setScene(scene);
      stage.show();
```

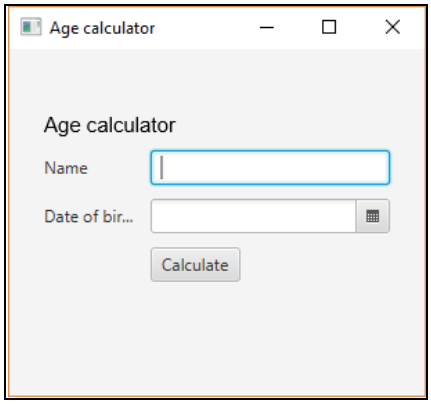
11. Now, we need to launch this JavaFX UI from the main method. We use the `launch(String[] args)` method of the `Application` class to launch the JavaFX UI:

```
      public static void main(String[] args) {
        Application.launch(args);
      }
```

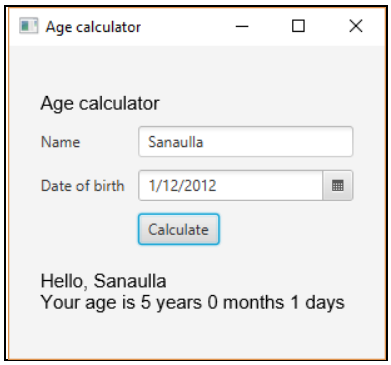
The complete code can be found at `chp9/1_create_javafx_gui`.

We have provided two scripts, `run.bat` and `run.sh`, in `chp9/1_create_javafx_gui`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh`, and you will see the GUI as shown in the following screenshot:

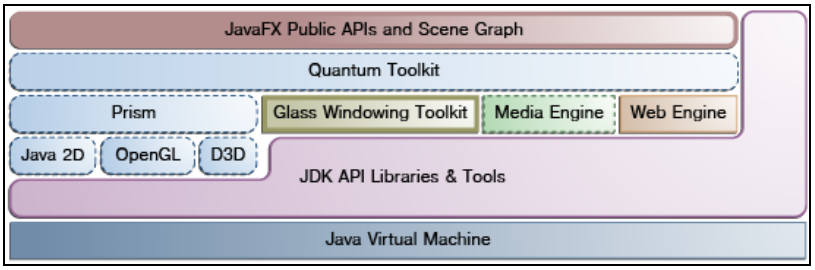


Enter the name and date of birth and click on `Calculate` to view the age:



How it works...

Before going into the other details, let's give you a brief overview of the JavaFX architecture. We have taken the following image describing the architecture stack from the JavaFX documentation (http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm#JFXST788):



Let's start from the top of the stack:

- **The JavaFX APIs and Scene graph**: This is the starting point of the application and most of our focus will be around this part. This provides APIs for different components, layout, and other utilities to facilitate developing a JavaFX-based UI. The scene graph holds the visual elements of the application.
- **Prism, quantum toolkit, and the other stuff in blue**: These components manage the rendering of the UI and provide a bridge between the underlying operating system and JavaFX. This layer provides software rendering in the cases where the graphics hardware is unable to provide hardware-accelerated rendering of rich UI and 3D elements.
- **The Glass Windowing Toolkit**: This is the windowing toolkit, just like the AWT used by Swing.
- **The media engine**: This supports media in JavaFX.
- **The web engine**: This supports the web component, which allows complete HTML rendering.
- **The JDK APIs and JVM**: These integrate with the Java API and compile the code down to byte code to run on the JVM.

Let's get back to explaining the recipe. The `javafx.application.Application` class is the entry point for launching the JavaFX applications. It has the following methods that map to the lifecycle of the application (in their invocation order):

- `init()`: This method is invoked immediately after the instantiation of `javafx.application.Application`. One can override this method to do some initialization before the start of the application. By default, this method does nothing.
- `start(javafx.stage.Stage)`: This method is called immediately after `init()` and after the system has done the required initialization to run the application. This method is passed with a `javafx.stage.Stage` instance, which is the primary stage on which the components are rendered. One can create other `javafx.stage.Stage` objects but the one provided by the application is the primary stage.
- `stop()`: This method is called when the application should stop. One can do the necessary exit-related operations.



A stage is a top-level JavaFX container. The primary stage passed as an argument to the `start()` method is created by the platform, and the application can create other `Stage` containers as and when required.

The other important method related to `javafx.application.Application` is the `launch()` method. There are two variants of this:

- `launch(Class<? extends Application> appClass, String... args)`
- `launch(String... args)`

This method is called from the main method and should be called only once. The first variant takes the name of the class that extends the `javafx.application.Application` class along with the arguments passed to the main method, and the second variant doesn't take the name of the class and, instead, should be invoked from within the class that extends the `javafx.application.Application` class. In our recipe, we have made use of the second variant.

We have created a class, `CreateGuiDemo`, which extends `javafx.application.Application`. This will be the entry point for JavaFX UI and we also added a main method to the class, making it an entry point for our application.

A layout construct determines how your components are laid out. There are multiple layouts supported by JavaFX, as follows:

- `javafx.scene.layout.HBox` and `javafx.scene.layout.VBox`: These are used to align the components horizontally and vertically
- `javafx.scene.layout.BorderPane`: This allows placing the components in the top, right, bottom, left, and center positions
- `javafx.scene.layout.FlowPane`: This layout allows placing the components in a flow, that is, beside each other, wrapping at the flow pane's boundary
- `javafx.scene.layout.GridPane`: This layout allows placing the components in a grid of rows and columns
- `javafx.scene.layout.StackPane`: This layout places the components in a back-to-front stack
- `javafx.scene.layout.TilePane`: This layout places the components in a grid of uniformly sized tiles

In our recipe, we have made use of `GridPane` and configured the layout so that we can achieve the following:

- The grid is placed at the center (`gridPane.setAlignment(Pos.CENTER);`)
- Set the gap between the columns to 10 (`gridPane.setHgap(10);`)
- Set the gap between the rows to 10 (`gridPane.setVgap(10);`)
- Set the padding within the cell of the grid (`gridPane.setPadding(new Insets(25, 25, 25, 25));`)

A `javafx.scene.text.Text` component's font can be set using the `javafx.scene.text.Font` object as shown here: `appTitle.setFont(Font.font("Arial", FontWeight.NORMAL, 15));`

While adding the component to `javafx.scene.layout.GridPane`, we have to mention the column number, row number, and column span, that is, how

many columns the component occupies, and the row span, that is, how many rows the component occupies in that order. The column span and row span are optional. In our recipe, we have placed `appTitle` in the first row and column, and it occupies two column spaces and one row space, as shown in the code here: `appTitle.setFont(Font.font("Arial", FontWeight.NORMAL, 15));`

The other important part in this recipe is the setting of the event for the `ageCalculator` button. We make use of the `setOnAction()` method of the `javafx.scene.control.Button` class to set the action performed when the button is clicked. This accepts an implementation of the `javafx.event.EventHandler<ActionEvent>` interface. As `javafx.event.EventHandler` is a functional interface, its implementation can be written in the form a lambda expression, as follows:

```
ageCalculator.setOnAction((event) -> {
  //event handling code here
});
```

The preceding syntax looks similar to your anonymous inner classes widely used during the times of Swing. You can learn more about functional interfaces and lambda expressions in the recipes in Chapter 4, *Going Functional*.

In our event handling code, we get the values from `nameField` and `dateOfBirthPicker` by using the methods, `getText()` and `getValue()`, respectively. `DatePicker` returns the date selected as an instance of `java.time.LocalDate`. This is one of the new date-time APIs added to Java 8. It represents a date, that is, year, month, and day, without any timezone-related information. We then make use of the `java.time.Period` class to find the duration between the current date and the selected date, as follows:

```
LocalDate now = LocalDate.now();
Period period = Period.between(dob, now);
```

`Period` represents the date-based duration in terms of years, months, and days, that is, for example, 3 years, 2 months, and 3 days. This is exactly what we are trying to extract with this line of code: `String.format("Your age is %d years %d months %d days",  period.getYears(), period.getMonths(), period.getDays()).`

We have already mentioned that the UI components in JavaFX are represented in the form of a scene graph and this scene graph is then rendered on to a container, called `Stage`. The way to create a scene graph is by using the `javafx.scene.Scene` class. We create a `javafx.scene.Scene` instance by passing the root of the scene graph and also by providing the dimensions of the container in which the scene graph is going to be rendered.

We make use of the container provided to the `start()` method, which is nothing but an instance of `javafx.stage.Stage`. Setting the scene for the `Stage` object and then calling its `show()` methods makes the complete scene graph rendered on the display:

```
stage.setScene(scene);
stage.show();
```

Using the FXML markup to create a GUI

In our first recipe, we looked at using Java APIs to build a UI. It often happens that the person who is adept at Java might not be a good UI designer, that is, he may be poor at identifying the best user experience for their app. In the world of web development, we have developers working on the frontend, based on the designs given by the UX designer and the other set of developers working on the backend to build services that are consumed by the frontend.

Both the developer parties agree on a set of APIs and a common data interchange model. The frontend developers work by using some mock data based on the data interchange model and also integrate the UI with the required APIs. On the other hand, the backend developers work on implementing the APIs so that they return the data in the interchange model agreed upon. So, both the parties work simultaneously using the expertise in their work areas.

It would be amazing if the same could be replicated (at least to some extent) on desktop applications. A step in this direction was the introduction of an XML-based language, called FXML. This enables a declarative method of UI development, where the developer can independently develop the UI using the same JavaFX components but available as XML tags. The different properties of the JavaFX components are available as attributes of the XML tags. Event handlers can be declared and defined in the Java code and then referred from FXML.

In this recipe, we will guide you through building the UI using FXML and then integrating FXML with the Java code for binding the action and for launching the UI defined in the FXML.

Getting ready

There is nothing much required if you have been able to implement the previous recipe. If you are coming directly to this recipe, there is nothing much to do either. The JavaFX APIs come with OpenJDK, which you must already have installed on your systems.

We will develop a simple age calculator app. This app will ask for the user's name (which is optional) and the date of birth, and calculate the age from the given date of birth and display it to the user.

How to do it...

1. All the FXML files should end with the `.fxml` extension. Let's create an empty `fxml_age_calc_gui.xml` file in the location, `src/gui/com/packt`. In the subsequent steps, we will update this file with the XML tags for the JavaFX components.
2. Create a `GridPane` layout, which will hold all the components in a grid of rows and columns. We will also provide the required spacing between the rows and the columns using the `vgap` and `hgap` attributes. Also, we will provide `GridPane`, which is our root component, with the reference to the Java class where we will add the required event handling. This Java class will be like the controller for the UI:

```
<GridPane alignment="CENTER" hgap="10.0" vgap="10.0"
  xmlns:fx="http://javafx.com/fxml"
  fx:controller="com.packt.FxmlController">
</GridPane>
```

3. We will provide the padding within each cell of the grid by defining a `padding` tag with `Insets` within `GridPane`:

```
<padding>
  <Insets bottom="25.0" left="25.0" right="25.0" top="25.0" />
</padding>
```

4. Next is to add a `Text` tag, which displays the title of the application: `Age Calculator`. We provide the required style information in the `style` attribute and the placement of the `Text` component within `GridPane` using the `GridPane.columnIndex` and `GridPane.rowIndex` attributes. The cell occupancy information can be provided using the `GridPane.columnSpan` and `GridPane.rowSpan` attributes:

```
<Text style="-fx-font: NORMAL 15 Arial;" text="Age calculator"
  GridPane.columnIndex="0" GridPane.rowIndex="0"
  GridPane.columnSpan="2" GridPane.rowSpan="1">
</Text>
```

5. We then add the `Label` and `TextField` components for accepting the name. Note the use of the `fx:id` attribute in `TextField`. This helps in binding this

component in the Java controller by creating a field with the same name as that of the `fx:id` value:

```
<Label text="Name" GridPane.columnIndex="0"
  GridPane.rowIndex="1">
</Label>
<TextField fx:id="nameField" GridPane.columnIndex="1"
  GridPane.rowIndex="1">
</TextField>
```

6. We add the `Label` and `DatePicker` components for accepting the date of birth:

```
<Label text="Date of Birth" GridPane.columnIndex="0"
  GridPane.rowIndex="2">
</Label>
<DatePicker fx:id="dateOfBirthPicker" GridPane.columnIndex="1"
  GridPane.rowIndex="2">
</DatePicker>
```

7. Then, we add a `Button` object and set its `onAction` attribute to the name of the method in the Java controller that handles the click event of this button:

```
<Button onAction="#calculateAge" text="Calculate"
  GridPane.columnIndex="1" GridPane.rowIndex="3">
</Button>
```

8. Finally we add a `Text` component to display the calculated age:

```
<Text fx:id="resultTxt" style="-fx-font: NORMAL 15 Arial;"
  GridPane.columnIndex="0" GridPane.rowIndex="5"
  GridPane.columnSpan="2" GridPane.rowSpan="1"
</Text>
```

9. The next step is to implement the Java class, which is directly related to the XML-based UI components created in the preceding steps. Create a class named `FxmlController`. This will contain the code that is relevant to the FXML UI; that is, it will contain the references to the components created in the FXML action handlers for the components created in the FXML:

```
public class FxmlController {
  //to implement in next few steps
}
```

10. We need references to the `nameField`, `dateOfBirthPicker`, and `resultText` components . We use the first two to get the entered name and date of birth, respectively, and the third to display the result of age calculation:

```
@FXML private Text resultTxt;
@FXML private DatePicker dateOfBirthPicker;
@FXML private TextField nameField;
```

11. The next step is to implement the `calculateAge` method, which is registered as the action event handler for the `Calculate` button. The implementation is similar to the one in the previous recipe. The only difference is that it is a method, unlike the previous recipe, where it was a lambda expression:

```
@FXML
public void calculateAge(ActionEvent event){
  String name = nameField.getText();
  LocalDate dob = dateOfBirthPicker.getValue();
  if ( dob != null ){
    LocalDate now = LocalDate.now();
    Period period = Period.between(dob, now);
    StringBuilder resultBuilder = new StringBuilder();
    if ( name != null && name.length() > 0 ){
      resultBuilder.append("Hello, ")
                   .append(name)
                   .append("n");
    }
    resultBuilder.append(String.format(
      "Your age is %d years %d months %d days",
      period.getYears(),
      period.getMonths(),
      period.getDays())
    );
    resultTxt.setText(resultBuilder.toString());
  }
}
```

12. In both steps 10 and 11, we have used an annotation, `@FXML`. This annotation indicates that the class or the member is accessible to the FXML-based UI.
13. Next, we'll create another Java class, `FxmlGuiDemo`, which is responsible for rendering the FXML-based UI and which would also be the entry point for launching the application:

```
public class FxmlGuiDemo extends Application{
  //code to launch the UI + provide main() method
}
```

14. Now we need to create a scene graph from the FXML UI definition by overriding the `start(Stage stage)` method of the `javafx.application.Application` class and then render the scene graph within the passed `javafx.stage.Stage` object:

```
@Override
public void start(Stage stage) throws IOException{
  FXMLLoader loader = new FXMLLoader();
  Pane pane = (Pane)loader.load(getClass()
      .getModule()
      .getResourceAsStream("com/packt/fxml_age_calc_gui.fxml")
  );

  Scene scene = new Scene(pane,300, 250);
  stage.setTitle("Age calculator");
  stage.setScene(scene);
  stage.show();
}
```

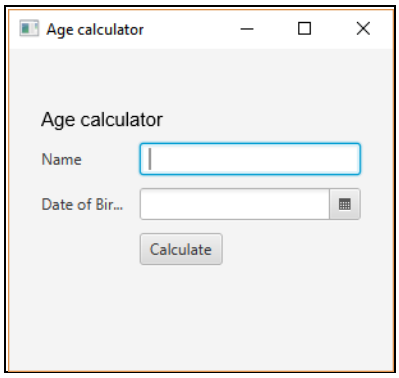
15. Finally, we provide the `main()` method implementation:

```
public static void main(String[] args) {
  Application.launch(args);
}
```

The complete code can be found at the location, `chp9/2_fxml_gui`.

We have provided two run scripts, `run.bat` and `run.sh`, in `chp9/2_fxml_gui`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh`, and you will see the GUI as shown in the following screenshot:



Enter the name and date of birth and click on `Calculate` to view the age:



How it works...

There is no XSD defining the schema for the FXML document. So, to know the tags to be used, they follow a simple naming convention. The Java class name of the component is also the name of the XML tag. For example, the XML tag for the `javafx.scene.layout.GridPane` layout is `<GridPane>`, for `javafx.scene.control.TextField`, it is `<TextField>`, and for `javafx.scene.control.DatePicker`, it is `<DatePicker>`:

```
Pane pane = (Pane)loader.load(getClass()
    .getModule()
    .getResourceAsStream("com/packt/fxml_age_calc_gui.fxml")
 );
```

The preceding line of code makes use of an instance of `javafx.fxml.FXMLLoader` to read the FXML file and get the Java representation of the UI components. `FXMLLoader` uses an event-based SAX parser to parse the FXML file. Instances of the respective Java classes for the XML tags are created via reflection and the values of attributes of the XML tags are populated into the respective properties of the Java classes.

As the root of our FXML is `javafx.scene.layout.GridPane`, which extends `javafx.scene.layout.Pane`, we can cast the return value from `FXMLoader.load()` to `javafx.scene.layout.Pane`.

The other interesting thing in this recipe is the `FxmlController` class. This class acts as an interface to FXML. We indicate the same in FXML by using the `fx:controller` attribute to the `<GridPane>` tag. We can get hold of the UI components defined in FXML by using the `@FXML` annotation against the member fields of the `FxmlController` class, as we did in this recipe:

```
@FXML private Text resultTxt;
@FXML private DatePicker dateOfBirthPicker;
@FXML private TextField nameField;
```

The name of the member is the same as that of the `fx:id` attribute value in FXML, and the type of the member is the same as that of the tag in FXML.

For example, the first member is bound to the following:

```
<Text fx:id="resultTxt" style="-fx-font: NORMAL 15 Arial;"
  GridPane.columnIndex="0" GridPane.rowIndex="5"
  GridPane.columnSpan="2" GridPane.rowSpan="1">
</Text>
```

On similar lines, we created an event handler in `FxmlController` and annotated it with `@FXML`, and the same has been referenced in FXML with the `onAction` attribute of `<Button>`. Note that we have added `#` to the beginning of the method name in the `onAction` attribute value.

See also

Refer to the following recipe of this chapter:

- Creating a GUI using JavaFX controls

Using CSS to style elements in JavaFX

Those from a web development background will be able to appreciate the usefulness of the **Cascading Style Sheets** (**CSS**), and for those who are not, we will provide an overview of what they are and how they are useful, before diving into CSS application in JavaFX.

The elements or the components that you see on web pages are often styled according to the theme of the website. This styling is possible by using a language called CSS. CSS consists of a group of name:value pairs, separated by semi colons. These name:value pairs, when associated with an HTML element, say, `<button>`, give it the required styling.

There are multiple ways to associate these name:value pairs to the element, the simplest being when you put this name:value pair within the style attribute of your HTML element. For example, to give the button a blue background, we can do the following:

```
<button style="background-color: blue;"></button>
```

There are predefined names for different styling properties, and these take a specific set of values; that is, the property, `background-color`, will only take valid color values.

The other approach is to define these groups of name:value pairs in a different file with a `.css` extension. Let's call this group of name:value pairs as CSS properties. We can associate these CSS properties with different selectors, that is, selectors for choosing the elements on the HTML page to apply the CSS properties to. There are three different ways of providing the selectors:

1. By directly giving the name of the HTML element, that is, whether it is an anchor tag (`<a>`), button, or input. In such cases, the CSS

properties are applied to all the types of HTML elements in the page.

2. By using the `id` attribute of the HTML element. Suppose, we have a button with `id="btn1"`, then we can define a selector, `#btn1`, against which we provide the CSS properties. Take a look at the following example:

```
#btn1 { background-color: blue; }
```

3. By using the class attribute of the HTML element. Suppose we have a button with `class="blue-btn"`, then we can define a selector, `.blue-btn`, against which, we provide the CSS properties. Check out the following example:

```
.blue-btn { background-color: blue; }
```

The advantage of using a different CSS file is that we can independently evolve the appearance of the web pages without getting tightly coupled to the location of the elements. Also, this encourages the reuse of CSS properties across different pages, thereby giving them a uniform look across all the pages.

When we apply a similar approach to JavaFX, we can leverage the CSS knowledge already available with our web designers to build CSS for JavaFX components, and this helps in styling the components more easily than with the use of Java APIs. When this CSS is mixed with FXML, then it becomes a known domain for web developers.

In this recipe, we will look at styling a few JavaFX components using an external CSS file.

Getting ready

There is a small difference in defining the CSS properties for the JavaFX components. All the properties must be prefixed with `-fx-`, that is, `background-color` becomes `-fx-background-color`. The selectors, that is, `#id` and `.class`-name still remain the same in the JavaFX world as well. We can even provide multiple classes to the JavaFX components, thereby applying all these CSS properties to the components.

The CSS that I have used in this recipe is based on a popular CSS framework called Bootstrap (`http://getbootstrap.com/css/`).

How to do it...

1. Let's create `GridPane`, which will hold the components in a grid of rows and columns:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

2. First, we will create a button and add two classes, `btn` and `btn-primary`, to it. In the next step, we will define these selectors with the required CSS properties:

```
Button primaryBtn = new Button("Primary");
primaryBtn.getStyleClass().add("btn");
primaryBtn.getStyleClass().add("btn-primary");
gridPane.add(primaryBtn, 0, 1);
```

3. Now, let's provide the required CSS properties for the classes, `btn` and `btn-primary`. The selector for the classes are of the form `.<class-name>`:

```
.btn{
  -fx-border-radius: 4px;
  -fx-border: 2px;
  -fx-font-size: 18px;
  -fx-font-weight: normal;
  -fx-text-align: center;
}
.btn-primary {
  -fx-text-fill: #fff;
  -fx-background-color: #337ab7;
  -fx-border-color: #2e6da4;
}
```

4. Let's create another button with a different CSS class:

```
Button successBtn = new Button("Sucess");
successBtn.getStyleClass().add("btn");
successBtn.getStyleClass().add("btn-success");
gridPane.add(successBtn, 1, 1);
```

5. Now we define the CSS properties for the `.btn-success` selector as follows:

```
.btn-success {
  -fx-text-fill: #fff;
  -fx-background-color: #5cb85c;
  -fx-border-color: #4cae4c;
}
```

6. Let's create yet another button with a different CSS class:

```
Button dangerBtn = new Button("Danger");
dangerBtn.getStyleClass().add("btn");
dangerBtn.getStyleClass().add("btn-danger");
gridPane.add(dangerBtn, 2, 1);
```

7. We will define the CSS properties for the selector `.btn-danger`:

```
.btn-danger {
  -fx-text-fill: #fff;
  -fx-background-color: #d9534f;
  -fx-border-color: #d43f3a;
}
```

8. Now, let's add some labels with different selectors, namely `badge`, `badge-info`:

```
Label label = new Label("Default Label");
label.getStyleClass().add("badge");
gridPane.add(label, 0, 2);

Label infoLabel = new Label("Info Label");
infoLabel.getStyleClass().add("badge");
infoLabel.getStyleClass().add("badge-info");
gridPane.add(infoLabel, 1, 2);
```

9. The CSS properties for the previous selectors are:

```
.badge{
  -fx-label-padding: 6,7,6,7;
  -fx-font-size: 12px;
  -fx-font-weight: 700;
  -fx-text-fill: #fff;
  -fx-text-alignment: center;
  -fx-background-color: #777;
  -fx-border-radius: 4;
}

.badge-info{
  -fx-background-color: #3a87ad;
}
.badge-warning {
  -fx-background-color: #f89406;
}
```

10. Let's add `TextField` with a `big-input` class:

```
TextField bigTextField = new TextField();
bigTextField.getStyleClass().add("big-input");
gridPane.add(bigTextField, 0, 3, 3, 1);
```

11. We define CSS properties so that the content of the text box is large in size and red in color:

```
.big-input{
  -fx-text-fill: red;
  -fx-font-size: 18px;
  -fx-font-style: italic;
  -fx-font-weight: bold;
}
```

12. Let's add some radio buttons:

```
ToggleGroup group = new ToggleGroup();
RadioButton bigRadioOne = new RadioButton("First");
bigRadioOne.getStyleClass().add("big-radio");
bigRadioOne.setToggleGroup(group);
bigRadioOne.setSelected(true);
gridPane.add(bigRadioOne, 0, 4);
RadioButton bigRadioTwo = new RadioButton("Second");
bigRadioTwo.setToggleGroup(group);
bigRadioTwo.getStyleClass().add("big-radio");
gridPane.add(bigRadioTwo, 1, 4);
```

13. We define CSS properties so that the labels of the radio button are large in size and green in color:

```
.big-radio{
  -fx-text-fill: green;
  -fx-font-size: 18px;
  -fx-font-weight: bold;
  -fx-background-color: yellow;
  -fx-padding: 5;
}
```

14. Finally, we add `javafx.scene.layout.GridPane` to the scene graph and render the scene graph on `javafx.stage.Stage`. We also need to associate the `stylesheet.css` with the `Scene`:

```
Scene scene = new Scene(gridPane, 600, 500);
scene.getStylesheets().add("com/packt/stylesheet.css");
stage.setTitle("Age calculator");
stage.setScene(scene);
stage.show();
```

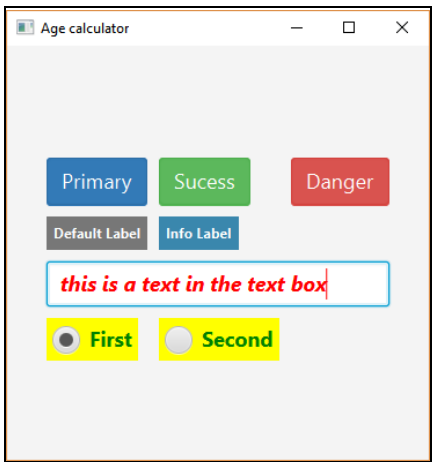
15. Add a `main()` method to launch the GUI:

```
    public static void main(String[] args) {
      Application.launch(args);
    }
```

The complete code can be found at the location, `chp9/3_css_javafx`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/3_css_javafx`. The `run.bat` will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh`, and you will see the following GUI:



How it works...

In this recipe, we make use of class names and their corresponding CSS selectors to associate components with different styling properties. JavaFX supports a subset of CSS properties and there are different properties applicable to different types of JavaFX components. The JavaFX CSS reference guide (`http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html`) will help you identify the supported CSS properties.

All the scene graph nodes extend from an abstract class, `javax.scene.Node`. This abstract class provides an API, `getStyleClass()`, which returns a list of class names (which are plain `String`) added to the node or the JavaFX component. As this is a simple list of class names, we can even add more class names to it by using `getStyleClass().add("new-class-name")`.

The advantage of using class names is that it allows us to group similar components by a common class name. This technique is widely used in the web development world. Suppose I have a list of buttons on the HTML page and I want a similar action to be performed on the click of each button. To achieve this, I will assign each of the buttons the same class, say, `my-button`, and then use `document.getElementsByClassName('my-button')` to get an array of these buttons. Now we can loop through the array of buttons obtained and add the required action handlers.

After assigning a class to the component, we need to write the CSS properties for the given class name. These properties then get applied to all the components with the same class name.

Let's pick one of the components from our recipe and see how we went about styling the same. Consider the following component with two classes, `btn` and `btn-primary`:

```
primaryBtn.getStyleClass().add("btn");
primaryBtn.getStyleClass().add("btn-primary");
```

We have used the selectors, `.btn` and `.btn-primary`, and we have grouped all the CSS properties under these selectors, as follows:

```
.btn{
  -fx-border-radius: 4px;
  -fx-border: 2px;
  -fx-font-size: 18px;
  -fx-font-weight: normal;
  -fx-text-align: center;
}
```

```
.btn-primary {
  -fx-text-fill: #fff;
  -fx-background-color: #337ab7;
  -fx-border-color: #2e6da4;
}
```

Note that in CSS, we have a `color` property, and its equivalent in JavaFX is `-fx-text-fill`. The rest of the CSS properties, namely `border-radius`, `border`, `font-size`, `font-weight`, `text-align`, `background-color`, and `border-color`, are prefixed with `-fx-`.

The important part is how you associate the stylesheet with the Scene component. The `scene.getStylesheets().add("com/packt/stylesheet.css");` line of code associates stylesheets with the Scene component. As `getStylesheets()` returns a list of strings, we can add multiple strings to it, which means that we can associate multiple stylesheets to a Scene.

The documentation of `getStylesheets()` states the following:

The URL is a hierarchical URI of the form [scheme:][//authority][path]. If the URL does not have a [scheme:] component, the URL is considered to be the [path] component only. Any leading '/' character of the [path] is ignored and the [path] is treated as a path relative to the root of the application's classpath.

In our recipe, we are using the `path` component only, and, hence, it looks for the file in the classpath. This is the reason why we have added the stylesheet in the same package as that of the scene. This is an easier way of making it available on the classpath.

Creating a bar chart

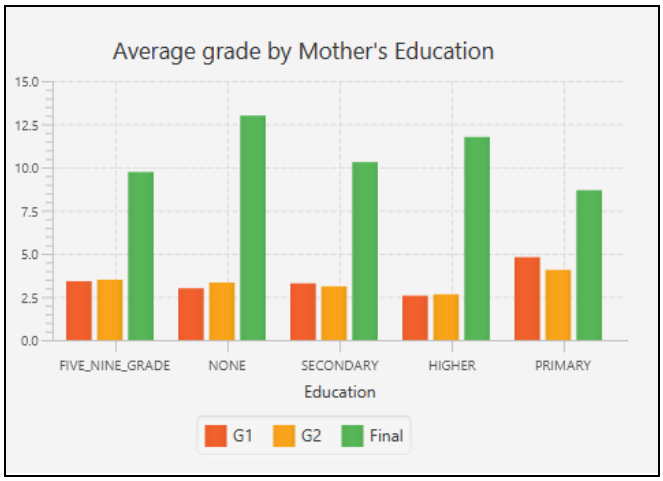
Data, when represented in the form of tables, is very hard to understand, but the same, when represented graphically by using charts, is comfortable for the eyes and easy to understand. We have seen a lot of charting libraries for web applications. However, the same support was lacking on the desktop application front. Swing didn't have native support for creating charts and we had to rely on third-party applications such as **JFreeChart** (http://www.jfree.org/jfreechart/). With JavaFX though, we have native support for creating charts, and we are going to show you how to represent the data in the form of charts using the JavaFX chart components.

JavaFX supports the following chart types:

- Bar chart
- Line chart
- Pie chart
- Scatter chart
- Area chart
- Bubble chart

In the next few recipes, we will cover the construction of each chart type. This segregation of each chart type into a recipe of its own will help us in explaining the recipes in a simpler way and will aid better understanding.

This recipe will be all about bar charts. A sample bar chart looks something like this:



Bar charts can have a single bar or multiple bars (like the preceding image) for each value on the *x*-axis. Multiple bars help us in comparing multiple value points for each value on the *x*-axis.

Getting ready

We will make use of a subset of data from the student performance machine learning repository (https://archive.ics.uci.edu/ml/datasets/Student+Performance). The dataset consists of student performance in two subjects, Mathematics and Portuguese, along with their social background information, such as their parents' occupation and education, among other information. There are quite a lot of attributes in the dataset, but we will pick the following:

- Student's gender
- Student's age
- Father's education
- Father's occupation
- Mother's education
- Mother's occupation
- Whether the student has taken extra classes
- First term grades
- Second term grades
- Final grades

As we mentioned earlier, there are a lot of attributes captured in the data, but we should be good with a few important attributes that will help us in plotting some useful charts. Due to this, we have extracted the information from the dataset available in the machine learning repository into a separate file, which can be found at the location, `chp9/4_bar_charts/src/gui/com/packt/students`, in the code download for the book. An excerpt from the students file is as follows:

```
"F";18;4;4;"at_home";"teacher";"no";"5";"6";6
"F";17;1;1;"at_home";"other";"no";"5";"5";6
"F";15;1;1;"at_home";"other";"yes";"7";"8";10
"F";15;4;2;"health";"services";"yes";"15";"14";15
"F";16;3;3;"other";"other";"yes";"6";"10";10
"M";16;4;3;"services";"other";"yes";"15";"15";15
```

The entries are semicolon (`;`) separated. Each entry has been explained for what it represents. The education information (fields 3 and 4) is a numeric value, where each number represents the level of education, as follows:

- 0: None
- 1: Primary education (fourth grade)
- 2: Fifth to ninth grade
- 3: Secondary education
- 4: Higher education

We have created a module for processing the student file. The module name is `student.processor` and its code can be found at `chp9/101_student_data_processor`. So, if you want to change any code there, you can rebuild the JAR by running the `build-jar.bat` or `build-jar.sh` file. This will create a modular JAR, `student.processor.jar`, in the `mlib` directory. Then, you have to replace this modular JAR with the one present in the `mlib` directory of this recipe, that is, `chp9/4_bar_charts/mlib`.



We recommend you to build the student.processor modular jar from the source available in `chp9/101_student_data_processor`. We have provided `build-jar.bat` and `build-jar.sh` scripts to help you with building the jar. You just have to run the script relevant to your platform. And then copy the jar build in `101_student_data_processor/mlib` to `4_bar_charts/mlib`.

This way, we can reuse this module across all the recipes on charts.

How to do it...

1. First, create `GridPane` and configure it to place the charts that we will be creating:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

2. Use the `StudentDataProcessor` class from the `student.processor` module to parse the student file and load the data into `List` of `Student`:

```
StudentDataProcessor sdp = new StudentDataProcessor();
List<Student> students = sdp.loadStudent();
```

3. The raw data, that is, the list of `Student` objects is not useful for plotting a chart, so we need to process the students' grades by grouping the students according to their mothers' and fathers' education and computing the average of those students' grades (all three terms). For this, we will write a simple method, which accepts `List<Student>`, a grouping function, that is, the value on which the students need to be grouped, and a mapping function, that is, the value that has to be used to compute the average:

```
private Map<ParentEducation, IntSummaryStatistics> summarize(
  List<Student> students,
  Function<Student, ParentEducation> classifier,
  ToIntFunction<Student> mapper
){
  Map<ParentEducation, IntSummaryStatistics> statistics =
    students.stream().collect(
      Collectors.groupingBy(
        classifier,
        Collectors.summarizingInt(mapper)
      )
  );
  return statistics;
}
```

The preceding method uses the new Stream-based APIs. These APIs are so powerful that they group the students by using

`Collectors.groupingBy()` and then compute the statistics of their grades by using `Collectors.summarizingInt()`.

4. The data for the bar chart is provided as an instance of `XYChart.Series`. Each series results in one `Y` value for a given `X` value, which is one bar for a given `X` value. We will have multiple series, one for each term, that is, first term grades, second term grades, and the final grades. Let's create a method that takes in the statistics of each term grades and the `seriesName` and returns a `series` object:

```
private XYChart.Series<String,Number> getSeries(
    String seriesName,
    Map<ParentEducation, IntSummaryStatistics> statistics
){
  XYChart.Series<String,Number> series = new XYChart.Series<>();
  series.setName(seriesName);
  statistics.forEach((k, v) -> {
    series.getData().add(
      new XYChart.Data<String, Number>(
        k.toString(),v.getAverage()
      )
    );
  });
  return series;
}
```

5. We will create two bar charts: one for the average grade by mother's education and the other for the average grade by father's education. For this, we will create a method that will take `List<Student>` and a classifier, that is, a function that will return the value to be used to group the students. This method will do the necessary computations and return us a `BarChart` object:

```
private BarChart<String, Number> getAvgGradeByEducationBarChart(
  List<Student> students,
  Function<Student, ParentEducation> classifier
){
  final CategoryAxis xAxis = new CategoryAxis();
  final NumberAxis yAxis = new NumberAxis();
  final BarChart<String,Number> bc =
        new BarChart<>(xAxis,yAxis);
  xAxis.setLabel("Education");
  yAxis.setLabel("Grade");
  bc.getData().add(getSeries(
    "G1",
    summarize(students, classifier, Student::getFirstTermGrade)
  ));
  bc.getData().add(getSeries(
    "G2",
```

```
            summarize(students, classifier, Student::getSecondTermGrade)
          ));
          bc.getData().add(getSeries(
            "Final",
            summarize(students, classifier, Student::getFinalGrade)
          ));
          return bc;
        }
```

6. Create `BarChart` for the average grades by mother's education, and add it to `gridPane`:

```
        BarChart<String, Number> avgGradeByMotherEdu =
                        getAvgGradeByEducationBarChart(
          students,
          Student::getMotherEducation
        );
        avgGradeByMotherEdu.setTitle("Average grade by Mother's
                                      Education");
        gridPane.add(avgGradeByMotherEdu, 1,1);
```

7. Create `BarChart` for average grades by father's education and add it to `gridPane`:

```
        BarChart<String, Number> avgGradeByFatherEdu =
                      getAvgGradeByEducationBarChart(
          students,
          Student::getFatherEducation
        );
        avgGradeByFatherEdu.setTitle("Average grade by Father's
                                      Education");
        gridPane.add(avgGradeByFatherEdu, 2,1);
```

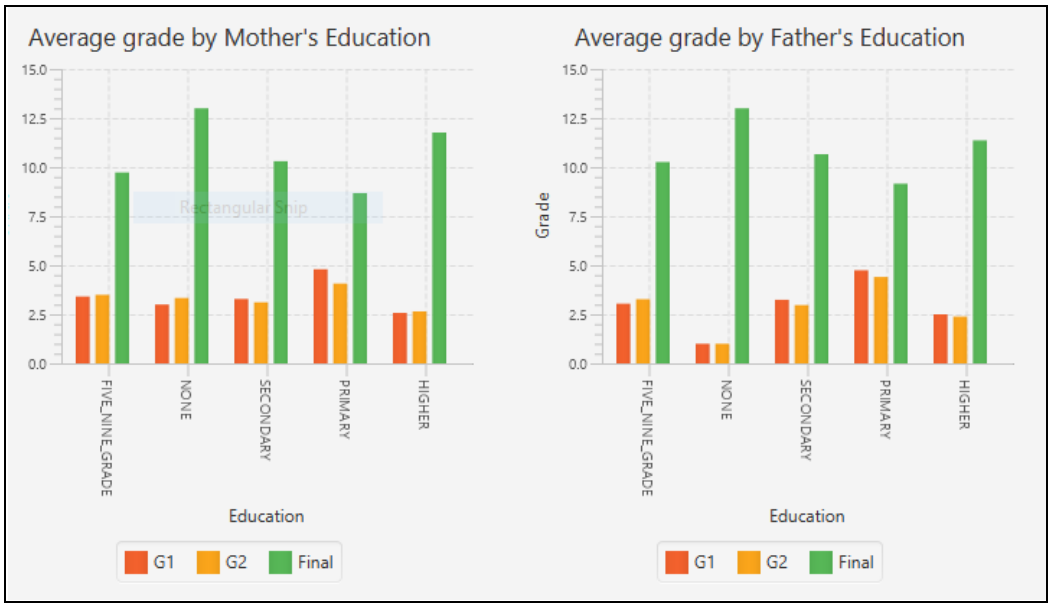
8. Create a scene graph using `gridPane` and set it to `Stage`:

```
        Scene scene = new Scene(gridPane, 800, 600);
        stage.setTitle("Bar Charts");
        stage.setScene(scene);
        stage.show();
```

The complete code can be found at `chp9/4_bar_charts`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/4_bar_charts`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh`, and you will see the following GUI:


Average grade by Mother's Education
15.0
12.5
10.0
7.5
5.0
2.5
0.0
FIVE_NINE_GRADE
NONE
SECONDARY
PRIMARY
HIGHER
Education
G1
G2
Final
Average grade by Father's Education
Grade
15.0
12.5
10.0
7.5
5.0
2.5
0.0
FIVE_NINE_GRADE
NONE
SECONDARY
PRIMARY
HIGHER
Education
G1
G2
Final

How it works...

Let's first see what it takes to create `BarChart`. `BarChart` is a two axes-based chart, where the data is plotted on two axes, namely the *x* axis (horizontal axis) and the *y* axis (vertical axis). The other two axes-based charts are area chart, bubble chart, and line chart.

In JavaFX, there are two types of axes supported:

- `javafx.scene.chart.CategoryAxis`: This supports string values on the axes
- `javafx.scene.chart.NumberAxis`: This supports numeric values on the axes

In our recipe, we created `BarChart` with `CategoryAxis` as the *x* axis, where we plot the education, and `NumberAxis` as the *y* axis, where we plot the grade, as follows:

```
final CategoryAxis xAxis = new CategoryAxis();
final NumberAxis yAxis = new NumberAxis();
final BarChart<String,Number> bc = new BarChart<>(xAxis,yAxis);
xAxis.setLabel("Education");
yAxis.setLabel("Grade");
```

In the next few paragraphs, we show you how the plotting of `BarChart` works.

The data to be plotted on `BarChart` should be a pair of values, where each pair represents *(x, y)* values, that is, a point on the *x* axis and a point on the *y* axis. This pair of values is represented by `javafx.scene.chart.XYChart.Data`. `Data` is a nested class within `XYChart`, which represents a single data item for a two axes-based chart. An `XYChart.Data` object can be created quite simply, as follows:

```
XYChart.Data item = new XYChart.Data("Cat1", "12");
```

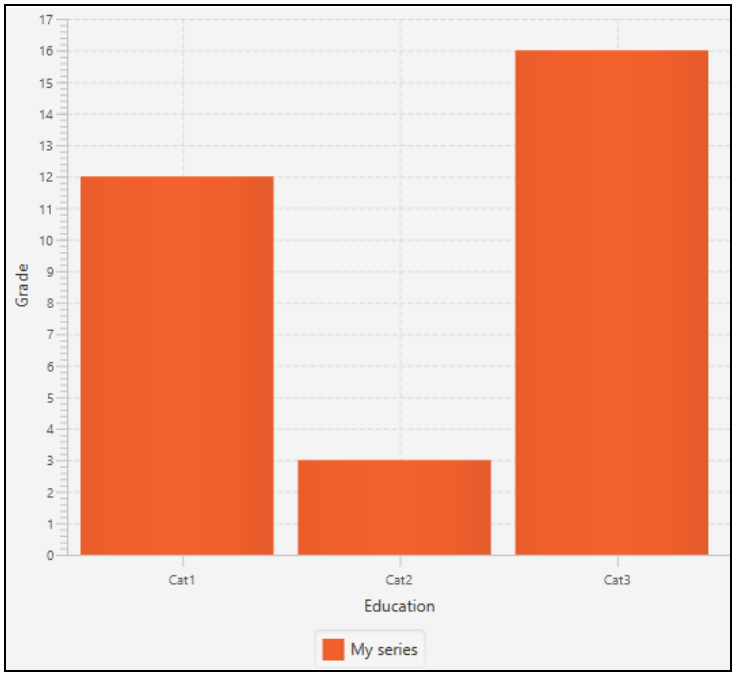
This is just one-data item. A chart can have multiple data items, that is, a series of data items. To represent a series of data items, JavaFX provides a class called `javafx.scene.chart.XYChart.Series`. This `XYChart.Series` object is a named series of `XYChart.Data` items. Let's create a simple series, as follows:

```
XYChart.Series<String,Number> series = new XYChart.Series<>();
series.setName("My series");
series.getData().add(
  new XYChart.Data<String, Number>("Cat1", 12)
);
series.getData().add(
  new XYChart.Data<String, Number>("Cat2", 3)
);
series.getData().add(
  new XYChart.Data<String, Number>("Cat3", 16)
);
```

`BarChart` can have multiple series of data items. If we provide it with multiple series, then there will be multiple bars for each data point on the *x* axis. For our demonstration of how this works, we will stick with one series. But the `BarChart` class in our recipe uses multiple series. Let's add the series to the `BarChart` and then render it onto the screen:

```
bc.getData().add(series);
Scene scene = new Scene(bc, 800, 600);
stage.setTitle("Bar Charts");
stage.setScene(scene);
stage.show();
```

This results in the following chart:



The other interesting part of this recipe is the grouping of students based on the education of mother and father and then computing the average of their

first term, second term, and final grades. The line of code that does the grouping and average computation is as follows:

```
Map<ParentEducation, IntSummaryStatistics> statistics =
        students.stream().collect(
  Collectors.groupingBy(
    classifier,
    Collectors.summarizingInt(mapper)
  )
);
```

The preceding line of code does the following:

- It creates a stream from `List<Student>`.
- It reduces this stream to the required grouping by using the `collect()` method.
- One of the overloaded versions of `collect()` takes two parameters. The first one is the function that returns the value on which the students need to be grouped. The second parameter is an additional mapping function, which maps the grouped student object into the required format. In our case, the required format is to get `IntSummaryStatistics` for the group of students on any of their grade values.

The preceding two pieces (setting up the data for a bar chart and creating the required objects to populate a `BarChart` instance) are important parts of the recipe; understanding them will give you a clearer picture of the recipe.

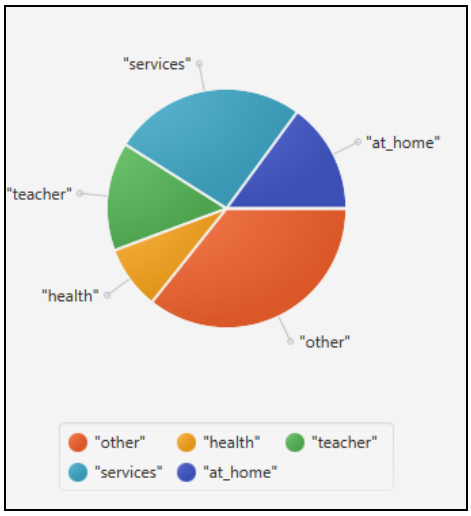
See also

Refer to the following recipes of this chapter:

- Creating an area chart
- Creating a bubble chart
- Creating a line chart
- Creating a scatter chart

Creating a pie chart

Pie charts, as the name suggests, are circular charts with slices (either joined or separated out), where each slice and its size indicates the magnitude of the item that the slice represents. Pie charts are used to compare the magnitudes of different classes, categories, products, and the like. This is how a sample pie chart looks:



Getting ready

We will make use of the same student data (taken from the machine learning repository and processed at our end) that we had discussed in the recipe, *Creating a bar chart*. For this, we have created a module, `student.processor`, which will read the student data and provide us with a list of `Student` objects. The source code for the module can be found at `chp9/101_student_data_processor`. We have provided the modular jar for the `student.processor` module at `chp9/5_pie_charts/mlib` of this recipe's code.



We recommend you to build the `student.processor` modular jar from the source available in `chp9/101_student_data_processor`. We have provided `build-jar.bat` and `build-jar.sh` scripts to help you with building the jar. You just have to run the script relevant to your platform. And then copy the jar build in `101_student_data_processor/mlib` to `4_bar_charts/mlib`.

How to do it...

1. Let's first create and configure `GridPane` to hold our pie charts:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

2. Create an instance of `StudentDataProcessor` (which comes from the `student.processor` module) and use it to load `List` of `Student`:

```
StudentDataProcessor sdp = new StudentDataProcessor();
List<Student> students = sdp.loadStudent();
```

3. Now, we need to get the count of students by their mother's and father's profession. We will write a method, which will take a list of students and a classifier, that is, the function that returns the value on which the students need to be grouped. The method returns an instance of `PieChart`:

```
private PieChart getStudentCountByOccupation(
    List<Student> students,
    Function<Student, String> classifier
){
  Map<String, Long> occupationBreakUp =
          students.stream().collect(
    Collectors.groupingBy(
      classifier,
      Collectors.counting()
    )
  );
  List<PieChart.Data> pieChartData = new ArrayList<>();
  occupationBreakUp.forEach((k, v) -> {
    pieChartData.add(new PieChart.Data(k.toString(), v));
  });
  PieChart chart = new PieChart(
    FXCollections.observableList(pieChartData)
  );
  return chart;
}
```

4. We will invoke the preceding method twice - one with the mother's occupation as the classifier and the other with the father's occupation

as the classifier. We then add the returned `PieChart` instance to `gridPane`. This should be done from within the `start()` method:

```
PieChart motherOccupationBreakUp = getStudentCountByOccupation(
  students, Student::getMotherJob
);
motherOccupationBreakUp.setTitle("Mother's Occupation");
gridPane.add(motherOccupationBreakUp, 1,1);

PieChart fatherOccupationBreakUp = getStudentCountByOccupation(
  students, Student::getFatherJob
);
fatherOccupationBreakUp.setTitle("Father's Occupation");
gridPane.add(fatherOccupationBreakUp, 2,1);
```

5. Next is to create the scene graph using `gridPane` and add it to `Stage`:

```
Scene scene = new Scene(gridPane, 800, 600);
stage.setTitle("Pie Charts");
stage.setScene(scene);
stage.show();
```

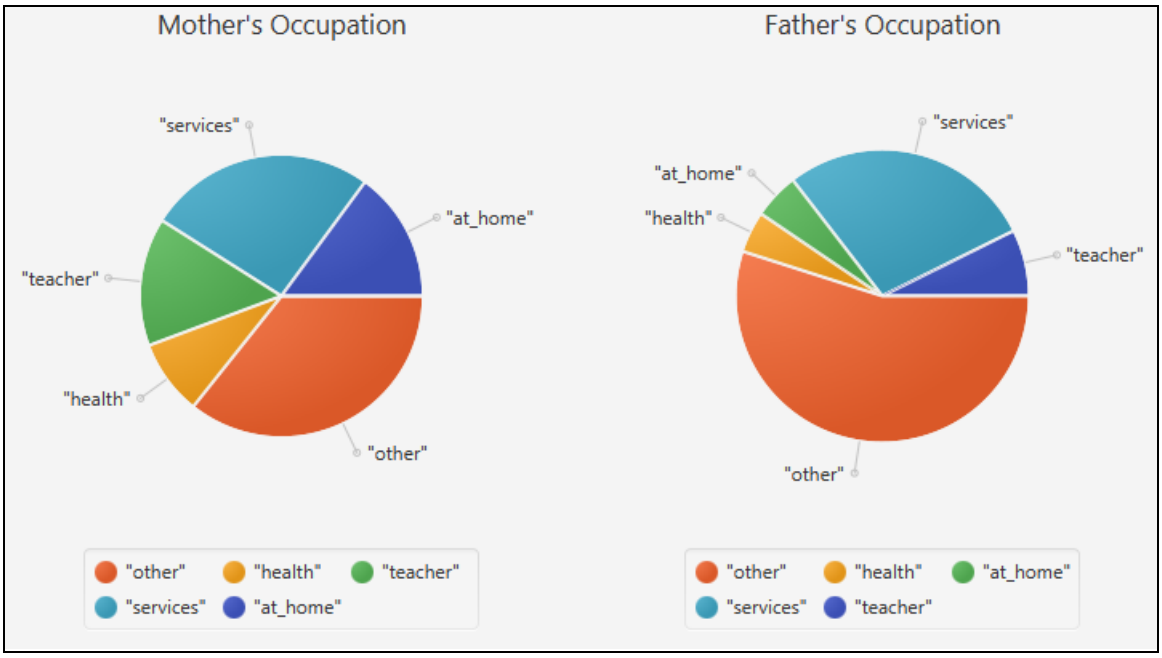
6. The UI can be launched from the main method by invoking the `Application.launch` method:

```
public static void main(String[] args) {
  Application.launch(args);
}
```

The complete code can be found at `chp9/5_pie_charts`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/5_pie_charts`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh` and you will see the following GUI:


Mother's Occupation
"services"
"at_home"
"teacher"
"health"
"other"
"other"
"health"
"teacher"
"services"
"at_home"
Father's Occupation
"services"
"at_home"
"health"
"teacher"
"other"
"other"
"health"
"at_home"
"services"
"teacher"

How it works...

The most important method that does all the work in this recipe is the `getStudentCountByOccupation()`. It does the following:

1. Group the number of students by profession. This can be done in a single line of code using the power of the new streaming APIs (added as part of Java 8):

```
Map<String, Long> occupationBreakUp =
            students.stream().collect(
  Collectors.groupingBy(
    classifier,
    Collectors.counting()
  )
);
```

2. Build data required for `PieChart`. The `PieChart` instance's data is `ObservableList` of `PieChart.Data`. We first make use of `Map` obtained in the preceding step to create `ArrayList` of `PieChart.Data`. Then, we use the `FXCollections.observableList()` API to obtain `ObservableList<PieChart.Data>` from `List<PieChart.Data>`:

```
List<PieChart.Data> pieChartData = new ArrayList<>();
occupationBreakUp.forEach((k, v) -> {
  pieChartData.add(new PieChart.Data(k.toString(), v));
});
PieChart chart = new PieChart(
  FXCollections.observableList(pieChartData)
);
```

The other important thing in the recipe is the classifiers we use: `Student::getMotherJob` and `Student::getFatherJob`. These are the two method references that invoke the `getMotherJob` and `getFatherJob` methods on the different instances of `Student` in the list of `Student`.

Once we get the `PieChart` instances, we add them to `GridPane` and then construct the scene graph using `GridPane`. The scene graph has to be associated with `Stage` for it to be rendered on the screen.

The main method launches the UI by invoking the `Application.launch(args);` method.

Creating a line chart

A line chart is a two-axis chart similar to the bar chart; instead of having bars, the data is plotted on the X-Y plane using points, and the points are joined together to depict the change of data. Line charts are used to get an understanding of how a certain variable is performing and when combined with multiple variables, that is, by using multiple series, we can see how each variable is performing when compared to other variables.

Getting ready

In this recipe, we will make use of crude oil and brent oil price variation over the last three years. This data can be found at the locations, `chp9/6_line_charts/src/gui/com/packt/crude-oil` and `chp9/6_line_charts/src/gui/com/packt/brent-oil`.

How to do it...

1. We will create a **Plain Old Java Object** (**POJO**) to represent the oil price in a given month and year:

```
public class OilPrice{
  public String period;
  public Double value;
}
```

2. Next is to write a `getOilData(String oilType)` method, which will read the data from the given file and construct `List<OilPrice>`:

```
private List<OilPrice> getOilData(String oilType)
        throws IOException{
  Scanner reader = new Scanner(getClass()
              .getModule()
              .getResourceAsStream("com/packt/"+oilType)
  );
  List<OilPrice> data = new LinkedList<>();
  while(reader.hasNext()){
    String line = reader.nextLine();
    String[] elements = line.split("t");
    OilPrice op = new OilPrice();
    op.period = elements[0];
    op.value = Double.parseDouble(elements[1]);
    data.add(op);
  }
  Collections.reverse(data);
  return data;
}
```

3. Next, we will write a method, which will take the name of the series and the data to be populated in the series:

```
private XYChart.Series<String,Number> getSeries(
  String seriesName, List<OilPrice> data
) throws IOException{
  XYChart.Series<String,Number> series = new XYChart.Series<>();
```

```
    series.setName(seriesName);
    data.forEach(d -> {
      series.getData().add(new XYChart.Data<String, Number>(
        d.period, d.value
      ));
    });
    return series;
  }
```

4. Create an empty `start(Stage stage)` method, which we will override in the next few steps:

```
@Override
public void start(Stage stage) throws IOException {
  //code to be added here from the next few steps
}
```

5. Create and configure `GridPane`:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

6. Create `CategoryAxis` as the *x* axis and `NumberAxis` as the *y* axis, and label them accordingly:

```
final CategoryAxis xAxis = new CategoryAxis();
final NumberAxis yAxis = new NumberAxis();
xAxis.setLabel("Month");
yAxis.setLabel("Price");
```

7. Initialize a `LineChart` instance with the axes created in the preceding step:

```
final LineChart<String,Number> lineChart =
new LineChart<>(xAxis,yAxis);
```

8. Load the crude oil data into `List<OilPrice>`:

```
List<OilPrice> crudeOil = getOilData("crude-oil");
```

9. Load Brent oil data into `List<OilPrice>`:

```
List<OilPrice> brentOil = getOilData("brent-oil");
```

10. Create a series, each for crude oil and Brent oil, and add it to `lineChart`:

```
        lineChart.getData().add(getSeries("Crude Oil", crudeOil));
        lineChart.getData().add(getSeries("Brent Oil", brentOil));
```

11. Add `LineChart` to `gridPane`:

```
        gridPane.add(lineChart, 1, 1);
```

12. Create a scene graph with `GridPane` as the root and set the size as required:

```
        Scene scene = new Scene(gridPane, 800, 600);
```

13. Set the properties for the `Stage` object passed into the `start(Stage stage)` method and associate the scene graph created in the preceding step:

```
        stage.setTitle("Line Charts");
        stage.setScene(scene);
        stage.show();
```

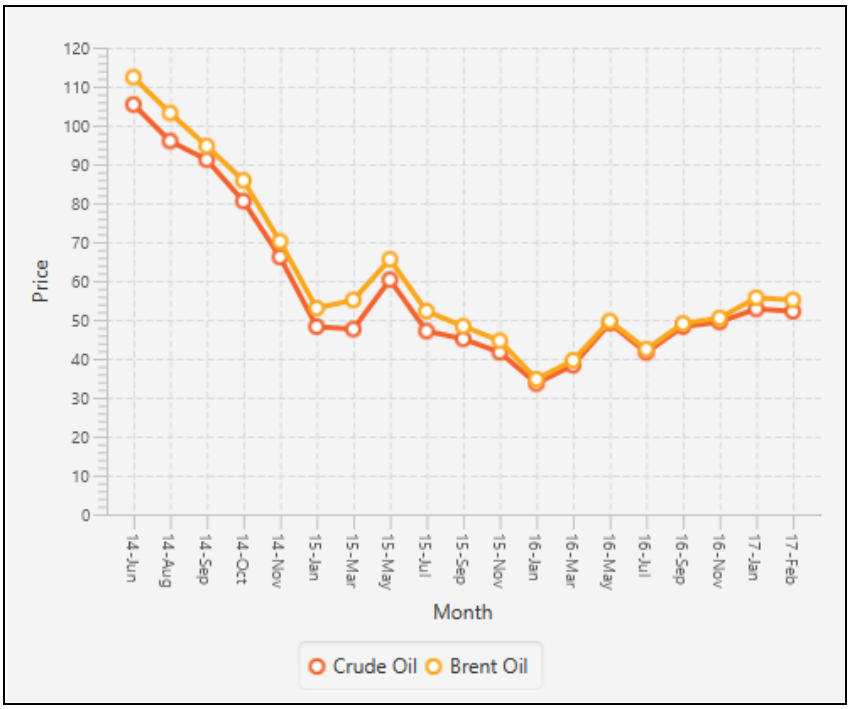
14. Launch the UI by invoking the `javafx.application.Application.launch()` method from the `main` method:

```
        public static void main(String[] args) {
          Application.launch(args);
        }
```

The complete code can be found at `chp9/6_line_charts`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/6_line_charts`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh`, and you will see the following GUI:


120
110
100
90
80
70
60
50
40
30
20
10
0
Price
14-Jun
14-Aug
14-Sep
14-Oct
14-Nov
15-Jan
15-Mar
15-May
15-Jul
15-Sep
15-Nov
16-Jan
16-Mar
16-May
16-Jul
16-Sep
16-Nov
17-Jan
17-Feb
Month
Crude Oil
Brent Oil

How it works...

Like any other two-axis chart, the line chart has the *x* axis and *y* axis. These axes can be of string types or numeric. String values are represented by `javafx.scene.chart.CategoryAxis` and numeric values by `javafx.scene.chart.NumberAxis`.

A new `LineChart` is created by providing the *x* axis and *y* axis objects as parameters to its constructor:

```
final CategoryAxis xAxis = new CategoryAxis();
final NumberAxis yAxis = new NumberAxis();
xAxis.setLabel("Category");
yAxis.setLabel("Price");
final LineChart<String,Number> lineChart = new LineChart<>(xAxis,yAxis);
```

Data to `LineChart` is provided in the form of an instance of `XYChart.Series`. So, if `LineChart` uses `String` on the *x* axis and `Number` on the *y* axis, then we create an instance of `XYChart.Series<String, Number>`, as follows:

```
XYChart.Series<String,Number> series =  new XYChart.Series<>();
series.setName("Series 1");
```

`XYChart.Series` contains data of the `XYChart.Data` type, so `XYChart.Series<String, Number>` will contain data of the `XYChart.Data<String, Number>` type. Let's add some data to the series created in the preceding step:

```
series.getData().add(new XYChart.Data<String, Number>("Cat 1", 10));
series.getData().add(new XYChart.Data<String, Number>("Cat 2", 20));
series.getData().add(new XYChart.Data<String, Number>("Cat 3", 30));
series.getData().add(new XYChart.Data<String, Number>("Cat 4", 40));
```

We then add the series to the `lineChart`'s data:

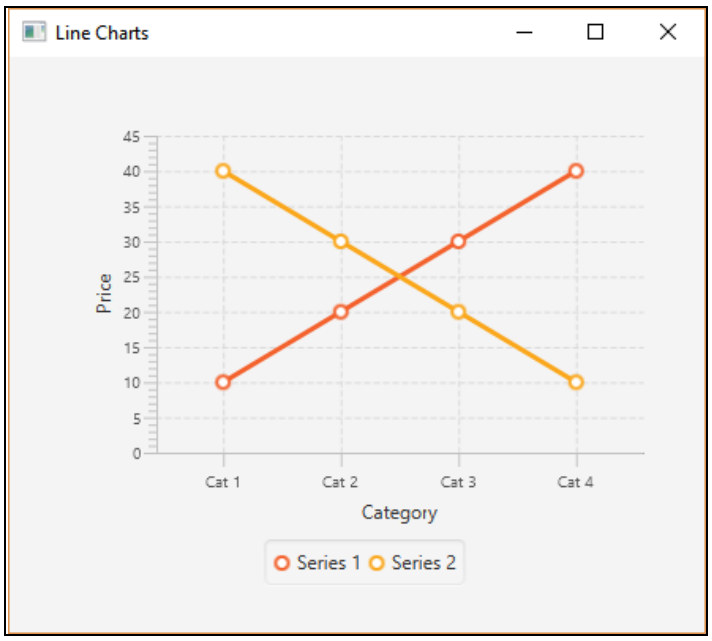
```
lineChart.getData().add(series);
```

We can create one more series on similar lines:

```
XYChart.Series<String, Number> series2 = new XYChart.Series<>();
series2.setName("Series 2");
series2.getData().add(new XYChart.Data<String, Number>("Cat 1", 40));
```

```
series2.getData().add(new XYChart.Data<String, Number>("Cat 2", 30));
series2.getData().add(new XYChart.Data<String, Number>("Cat 3", 20));
series2.getData().add(new XYChart.Data<String, Number>("Cat 4", 10));
lineChart.getData().add(series2);
```

The chart created looks like the following:



See also

Refer to the following recipes of this chapter:

- Creating a bar chart
- Creating an area chart
- Creating a bubble chart
- Creating a scatter chart

Creating an area chart

An area chart is similar to a line chart, the only difference being the area between the plotted line and the axis is colored, with different series being colored by different colors. In this recipe, we will look at creating an area chart.

Getting ready

We will make use of the crude oil and Brent oil price variation data from the previous recipe (*Creating a line chart*) to plot an area chart.

How to do it...

1. We will create a POJO to represent the oil price in a given month and year:

```
public class OilPrice{
  public String period;
  public Double value;
}
```

2. Next, we'll write a method, `getOilData(String oilType)`, which will read the data from the given file and construct `List<OilPrice>`:

```
private List<OilPrice> getOilData(String oilType)
          throws IOException{
  Scanner reader = new Scanner(getClass()
                  .getModule()
                  .getResourceAsStream("com/packt/"+oilType)
  );
  List<OilPrice> data = new LinkedList<>();
  while(reader.hasNext()){
    String line = reader.nextLine();
    String[] elements = line.split("t");
    OilPrice op = new OilPrice();
    op.period = elements[0];
    op.value = Double.parseDouble(elements[1]);
    data.add(op);
  }
  Collections.reverse(data);
  return data;
}
```

3. Next, we will write a method that will take the name of the series and the data to be populated in the series:

```
private XYChart.Series<String,Number> getSeries(
  String seriesName, List<OilPrice> data
) throws IOException{
  XYChart.Series<String,Number> series = new XYChart.Series<>();
  series.setName(seriesName);
  data.forEach(d -> {
    series.getData().add(new XYChart.Data<String, Number>(
      d.period, d.value
    ));
  });
  return series;
}
```

4. Create an empty `start(Stage stage)` method, which we will override in the next few steps:

```
@Override
public void start(Stage stage) throws IOException {
  //code to be added here from the next few steps
}
```

5. Create and configure `GridPane`:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

6. Create `CategoryAxis` as the *x* axis and `NumberAxis` as the *y* axis and label them accordingly:

```
final CategoryAxis xAxis = new CategoryAxis();
final NumberAxis yAxis = new NumberAxis();
xAxis.setLabel("Month");
yAxis.setLabel("Price");
```

7. Initialize `AreaChart` with the axes created in the preceding steps:

```
final AreaChart<String,Number> areaChart =
                new AreaChart<>(xAxis,yAxis);
```

8. Load crude oil data into `List<OilPrice>`:

```
List<OilPrice> crudeOil = getOilData("crude-oil");
```

9. Load Brent oil data into `List<OilPrice>`:

```
List<OilPrice> brentOil = getOilData("brent-oil");
```

10. Create a series each for crude oil and brent oil, and add it to `AreaChart`:

```
areaChart.getData().add(getSeries("Crude Oil", crudeOil));
areaChart.getData().add(getSeries("Brent Oil", brentOil));
```

11. Add `AreaChart` to `GridPane`:

```
gridPane.add(areaChart, 1, 1);
```

12. Create a scene graph with `GridPane` as the root and set the size as required:

```
Scene scene = new Scene(gridPane, 800, 600);
```

13. Set the properties for the `Stage` object passed into the `start(Stage stage)` method and associate the scene graph created in the preceding step:

```
stage.setTitle("Area Charts");
stage.setScene(scene);
stage.show();
```

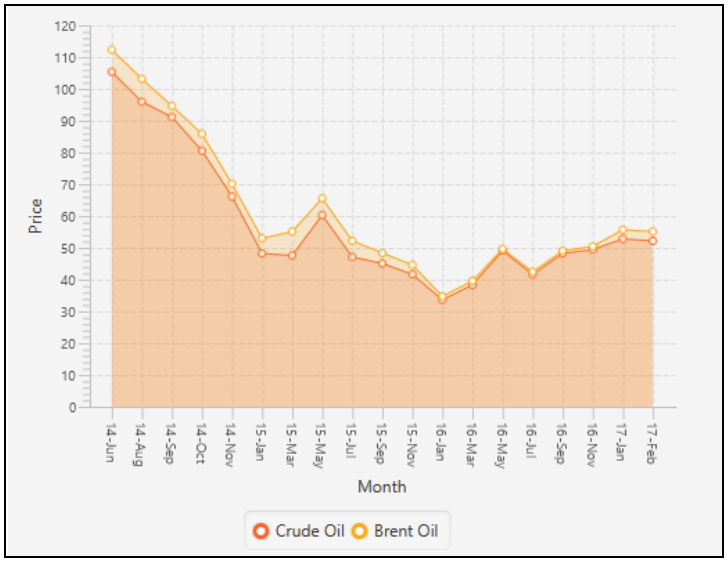
14. Launch the UI by invoking the `javafx.application.Application.launch()` method from the `main` method:

```
public static void main(String[] args) {
  Application.launch(args);
}
```

The complete code can be found at `chp9/7_area_charts`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/7_area_charts`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh` and you will see the following GUI:



How it works...

Area charts are similar to line charts so we would highly recommend reading the recipe, *Creating a line chart*. An area chart is made up of two axes with data plotted on the *x* and *y* axis. The data to be plotted is provided as an instance of `XYChart.Series`. The axis of the chart can be either `javafx.scene.chart.CategoryAxis` or `javafx.scene.chart.NumberAxis`.

`XYChart.Series` contains data encapsulated in instances of `XYChart.Data`.

See also

Refer to the following recipes of this chapter:

- Creating a bar chart
- Creating an area chart
- Creating a bubble chart
- Creating a pie chart
- Creating a scatter chart

Creating a bubble chart

A Bubble chart is also a two-axis chart with a third dimension to the data, that is, the radius of the bubble. Bubble chart supports only `NumberAxis`, so we can have only numbers on both *x* and *y* axes.

In this recipe, we will create a simple bubble chart.

Getting ready

We have provided a sample store visit data at different times during the day along with the sales information during that hour. This sample data file can be found at the location, `chp9/8_bubble_charts/src/gui/com/packt/store`. Each line in the data file consists of three parts:

1. The hour of the day
2. The number of visits
3. Total sales

How to do it...

1. Let's create a method to read the data from the file into `StoreVisit` objects:

```
private List<StoreVisit> getData() throws IOException{
  Scanner reader = new Scanner(getClass()
                .getModule()
                .getResourceAsStream("com/packt/store")
  );
  List<StoreVisit> data = new LinkedList<>();
  while(reader.hasNext()){
    String line = reader.nextLine();
    String[] elements = line.split(",");
    StoreVisit sv = new StoreVisit(elements);
    data.add(sv);
  }
  return data;
}
```

2. We also need the maximum sale value during any part of the day. So, let's create a method, which accepts the `List<StoreVisit>` and returns the maximum sale. We will use this maximum sale to determine the radius of the bubble in the chart:

```
private Integer getMaxSale(List<StoreVisit> data){
  return data.stream()
             .mapToInt(StoreVisit::getSales)
             .max()
             .getAsInt();
}
```

3. Create and configure a `GridPane` object where we will place the chart:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

4. Create two `NumberAxis` objects for *x* and *y* axis and give them corresponding names:

```
final NumberAxis xAxis = new NumberAxis();
final NumberAxis yAxis = new NumberAxis();
```

```
xAxis.setLabel("Hour");
yAxis.setLabel("Visits");
```

5. Create an instance of `BubbleChart` with these two axes:

```
final BubbleChart<Number,Number> bubbleChart =
                    new BubbleChart<>(xAxis,yAxis);
```

6. Create an `XYChart.Series` from the store visit data, read from the file:

```
List<StoreVisit> data = getData();
Integer maxSale = getMaxSale(data);
XYChart.Series<Number,Number> series =
                new XYChart.Series<>();
series.setName("Store Visits");
data.forEach(sv -> {
  series.getData().add(
    new XYChart.Data<Number, Number>(
      sv.hour, sv.visits, (sv.sales/(maxSale * 1d)) * 2
    )
  );
});
```

7. Populate `bubbleChart` with the series created in the preceding step and add it to `gridPane`:

```
bubbleChart.getData().add(series);
gridPane.add(bubbleChart, 1, 1);
```

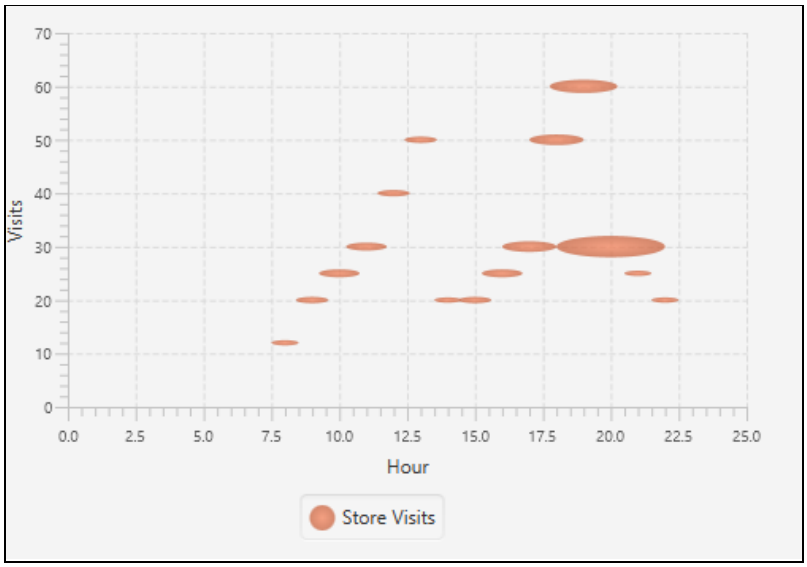
8. Render the chart by creating a scene graph with `gridPane` and set it to the `Stage` object passed into the `start(Stage stage)` method:

```
Scene scene = new Scene(gridPane, 600, 400);
stage.setTitle("Bubble Charts");
stage.setScene(scene);
stage.show();
```

The complete code can be found at `chp9/8_bubble_charts`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/8_bubble_charts`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh` and you will see the following GUI:


70
60
50
40
30
20
10
0
Visits
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Hour
Store Visits

How it works...

Bubble chart is another two-axis chart just like area chart, line chart, and bar chart. The only difference being that the `XYChart.Data` object takes in the third argument in its constructor, which determines the radius of the bubble. The general idea is that the greater the bubble radius, the greater is the impact/contribution of that data point. So, in our example, we have used the sales value and the maximum sales value to determine the radius by using the formula, `(sales/(maxSale  * 1d)) * 2`, which means we are sizing the sales to the scale of 2.

The rest of the details are exactly similar to the other two-axis charts we have seen, namely bar charts, line charts, and area charts. Therefore, we are not going into its details and we would highly recommend you visit those recipes.

See also

Refer to the following recipes of this chapter:

- Creating a bar chart
- Creating an area chart
- Creating a line chart
- Creating a scatter chart

Creating a scatter chart

A scatter chart is another type of two-axis charts, where the data is presented as a set of points. Each series in the chart is presented by a different shape. These points are not joined, unlike line charts. Such charts help us in identifying where the bulk of the data lies by looking at the density of the data points. In this recipe, we will look at creating a simple scatter chart.

Getting ready

I have collected the statistics from the one-day international cricket match between India and New Zealand held on October 26, 2016 (http://www.espncricinfo.com/series/1030193/scorecard/1030221). The data collected is the score and overs in progress during the fall of wickets during both New Zealand and India's innings. This data can be found in the file, `chp9/9_scatter_charts/src/gui/com/packt/wickets`. The data looks like the following:

```
NZ,0.2,0
NZ,20.3,120
NZ,30.6,158
NZ,40.5,204
```

In the preceding sample, each line of data has three parts:

- The team
- The over in progress during the fall of the wicket
- Team score during the fall of the wicket

We will plot this data on a scatter chart and get an idea as to how the wickets fell during each team's innings.

For those who are wondering what this game called cricket is all about, we would suggest you spend a few minutes reading about it here: https://en.wikipedia.org/wiki/Cricket.

How to do it...

1. Let's first create a method to read the fall of wickets data from the file:

```
private Map<String, List<FallOfWicket>> getFallOfWickets()
        throws IOException{
  Scanner reader = new Scanner(getClass()
                   .getModule()
                   .getResourceAsStream("com/packt/wickets")
  );
  Map<String, List<FallOfWicket>> data = new HashMap<>();
  while(reader.hasNext()){
    String line = reader.nextLine();
    String[] elements = line.split(",");
    String country = elements[0];
    if ( !data.containsKey(country)){
      data.put(country, new ArrayList<FallOfWicket>());
    }
    data.get(country).add(new FallOfWicket(elements));
  }
  return data;
}
```

2. A scatter chart is also an X-Y chart; we will use `XYChart.Series` to create data for the chart. Let's write a method to create an instance of `XYChart.Series<Number,Number>` using the data parsed from the file:

```
private XYChart.Series<Number, Number> getSeries(
  List<FallOfWicket> data, String seriesName
){
  XYChart.Series<Number,Number> series = new XYChart.Series<>();
  series.setName(seriesName);
  data.forEach(s -> {
    series.getData().add(
      new XYChart.Data<Number, Number>(s.over, s.score)
    );
  });
  return series;
}
```

3. Let's now build the UI (all this code goes within the `start(Stage stage)` method), starting with creating an instance of `GridPane` and configuring it:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
```

```
        gridPane.setVgap(10);
        gridPane.setPadding(new Insets(25, 25, 25, 25));
```

4. Load the data from the file using the method created in the first step:

```
        Map<String, List<FallOfWicket>> fow = getFallOfWickets();
```

5. Create the required *x* and *y* axes and add them to `ScatterChart`:

```
        final NumberAxis xAxis = new NumberAxis();
        final NumberAxis yAxis = new NumberAxis();
        xAxis.setLabel("Age");
        yAxis.setLabel("Marks");
        final ScatterChart<Number,Number> scatterChart =
                           new ScatterChart<>(xAxis,yAxis);
```

6. Create `XYChart.Series` for each team's innings and add it to `ScatterChart`:

```
        scatterChart.getData().add(getSeries(fow.get("NZ"), "NZ"));
        scatterChart.getData().add(getSeries(fow.get("IND"), "IND"));
```

7. Add `ScatterChart` to `gridPane` and create a new `Scene` graph with `gridPane` as the root:

```
        gridPane.add(scatterChart, 1, 1);
        Scene scene = new Scene(gridPane, 600, 400);
```

8. Set the scene graph to the `Stage` instance to be rendered on the display:

```
        stage.setTitle("Bubble Charts");
        stage.setScene(scene);
        stage.show();
```

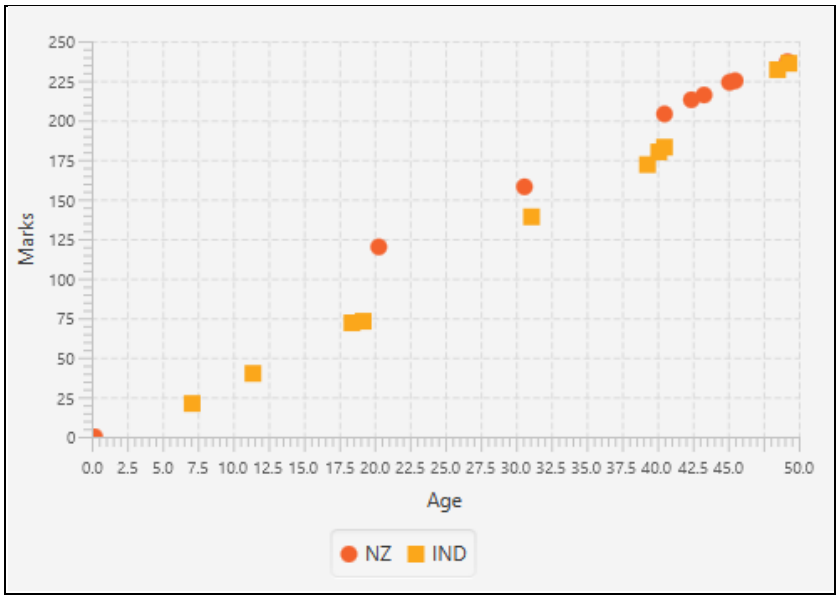
9. Launch the UI from the `main` method, as shown here:

```
        public static void main(String[] args) {
          Application.launch(args);
        }
```

You can find the complete code at the location, `chp9/9_scatter_charts`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/9_scatter_charts`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh` and you will see the following GUI:



How it works...

A scatter chart works in a similar way to how a bar chart or a line chart works. It's another two-axis graph, where the data points are plotted on the two axes. The axes are created using `javafx.scene.chart.CategoryAxis` or `javafx.scene.chart.NumberAxis`, depending on whether the data is string or numeric.

The data to be plotted is provided in the form of `XYChart.Series<X,Y>`, where `X` and `Y` can be `String` or any type extending `Number`, and it contains the data in the form of a list of `XYChart.Data` objects, something like the following:

```
XYChart.Series<Number,Number> series  = new XYChart.Series<>();
series.getData().add(new XYChart.Data<Number, Number>(xValue, yValue));
```

In a scatter chart, the data is plotted in the form of points, with each series having a specific color and shape, and these points are not joined, unlike line or area charts. This can be seen in the example used in the recipe.

See also

Refer to the following recipes of this chapter:

- Creating a bar chart
- Creating an area chart
- Creating a line chart
- Creating a pie chart
- Creating a bubble chart

Embedding HTML in an application

JavaFX provides support for managing web pages via the classes defined in the `javafx.scene.web` package. It supports loading the web page, either by accepting the web page URL or by accepting the web page content. It also manages the document model of the web page, applies the relevant CSS, and runs the relevant JavaScript code. It also extends support for a two-way communication between JavaScript and the Java code.

In this recipe, we will build a very primitive and simple web browser, which supports the following:

- Navigating through the history of the pages visited
- Reloading the current page
- An address bar for accepting the URL
- A button for loading the entered URL
- Showing the web page
- Showing the status of loading of the web page

Getting ready

We will require an internet connection to test the loading of pages. So, make sure you are connected to the internet. Apart from this, there is nothing specific required to work with this recipe.

How to do it...

1. Let's first create a class with empty methods, which would represent the main application for launching the application as well as the JavaFX UI:

```
public class BrowserDemo extends Application{
  public static void main(String[] args) {
    Application.launch(args);
  }
  @Override
  public void start(Stage stage) {
    //this will have all the JavaFX related code
  }
}
```

   In the subsequent steps, we will write all our code within the `start(Stage stage)` method.

2. Let's create a `javafx.scene.web.WebView` component, which will render our web page. This has the required `javafx.scene.web.WebEngine` instance, which manages loading of the web page:

```
WebView webView = new WebView();
```

3. Get the instance of `javafx.scene.web.WebEngine` used by `webView`. We will use this instance of `javafx.scene.web.WebEngine` to navigate through the history and load other web pages. Then we will, by default, load the URL, `http://www.google.com`:

```
WebEngine webEngine = webView.getEngine();
webEngine.load("http://www.google.com/");
```

4. Now, let's create a `javafx.scene.control.TextField` component, which will act as our browser's address bar:

```
TextField webAddress = new TextField("http://www.google.com/");
```

5. We want to change the title of the browser and the web page in the address bar, based on the title and URL of the completely loaded web

page. This can be done by listening to the change in the `stateProperty` of `javafx.concurrent.Worker` obtained from the `javafx.scene.web.WebEngine` instance:

```
webEngine.getLoadWorker().stateProperty().addListener(
  new ChangeListener<State>() {
    public void changed(ObservableValue ov,
                        State oldState, State newState) {
      if (newState == State.SUCCEEDED) {
        stage.setTitle(webEngine.getTitle());
        webAddress.setText(webEngine.getLocation());
      }
    }
  }
);
```

6. Let's create a `javafx.scene.control.Button` instance, which, on click, will load the web page identified by the URL entered in the address bar:

```
Button goButton = new Button("Go");
goButton.setOnAction((event) -> {
  String url = webAddress.getText();
  if ( url != null && url.length() > 0){
    webEngine.load(url);
  }
});
```

7. Let's create a `javafx.scene.control.Button` instance, which, on click, will go to the previous web page in the history. To achieve this, we will execute the JavaScript code, `history.back()`, from within the action handler:

```
Button prevButton = new Button("Prev");
prevButton.setOnAction(e -> {
  webEngine.executeScript("history.back()");
});
```

8. Let's create a `javafx.scene.control.Button` instance, which, on click, will go to the next entry in the history maintained by `javafx.scene.web.WebEngine` instance. For this, we will make use of the `javafx.scene.web.WebHistory` API:

```
Button nextButton = new Button("Next");
nextButton.setOnAction(e -> {
  WebHistory wh = webEngine.getHistory();
  Integer historySize = wh.getEntries().size();
  Integer currentIndex = wh.getCurrentIndex();
  if ( currentIndex < (historySize - 1)){
```

```
            wh.go(1);
          }
        });
```

9. Next is the button for reloading the current page. Again, we will make use of `javafx.scene.web.WebEngine` to reload the current page:

```
Button reloadButton = new Button("Refresh");
reloadButton.setOnAction(e -> {
  webEngine.reload();
});
```

10. Now, we need to group all the components created so far, namely, `prevButton`, `nextButton`, `reloadButton`, `webAddress`, and `goButton` so that they align horizontally with each other. To achieve this, we will make use of `javafx.scene.layout.HBox` with relevant spacing and padding to make the components look well spaced:

```
HBox addressBar = new HBox(10);
addressBar.setPadding(new Insets(10, 5, 10, 5));
addressBar.setHgrow(webAddress, Priority.ALWAYS);
addressBar.getChildren().addAll(
  prevButton, nextButton, reloadButton, webAddress, goButton
);
```

11. We would want to know whether the web page is loading and whether it has finished. Let's create a `javafx.scene.layout.Label` field to update the status if the web page is loaded. Then, we listen to the updates to `workDoneProperty` of the `javafx.concurrent.Worker` instance, which we can get from the `javafx.scene.web.WebEngine` instance:

```
Label websiteLoadingStatus = new Label();
webEngine.getLoadWorker().workDoneProperty().addListener(
                    new ChangeListener<Number>(){
  public void changed(ObservableValue ov, Number oldState,
                      Number newState) {
    if (newState.doubleValue() != 100.0){
      websiteLoadingStatus.setText("Loading " +
                                   webAddress.getText());
    }else{
      websiteLoadingStatus.setText("Done");
    }
  }
});
```

12. Let's align the entire address bar (with its navigation buttons), `webView`, and `websiteLoadingStatus` vertically:

```
VBox root = new VBox();
root.getChildren().addAll(
  addressBar, webView, websiteLoadingStatus
);
```

13. Create a new `Scene` object with the `VBox` instance created in the preceding step as the root:

```
Scene scene = new Scene(root);
```

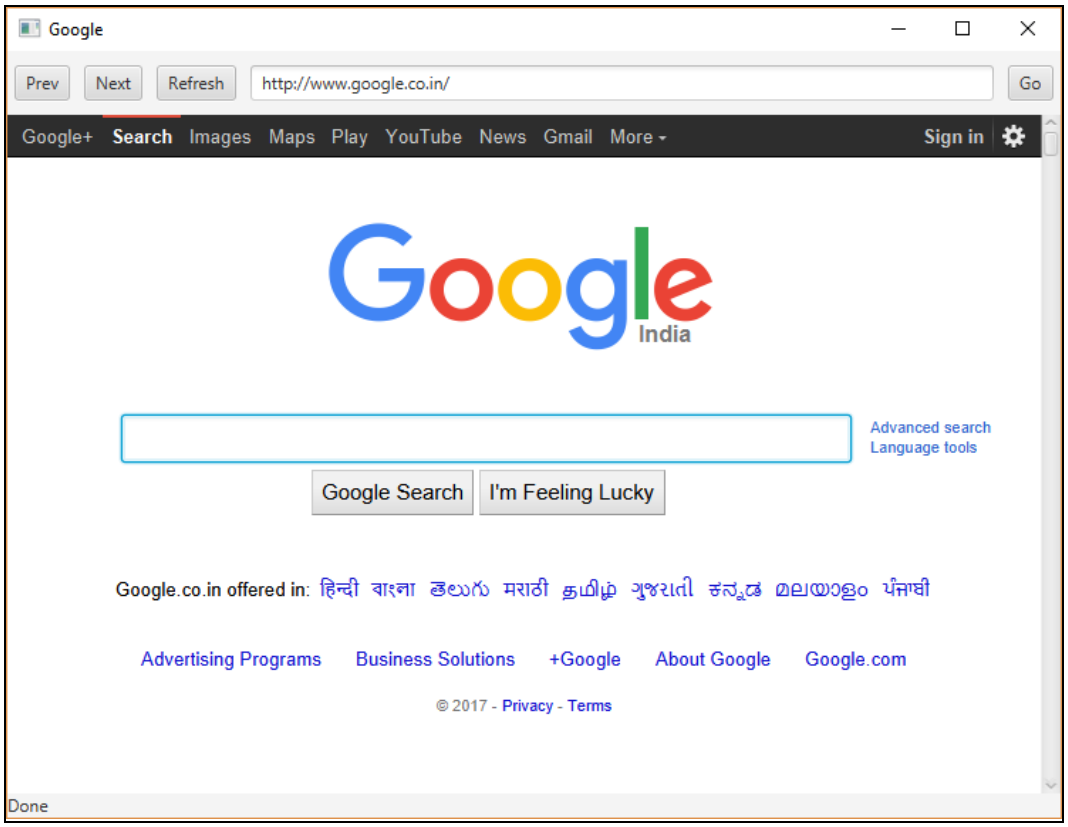
14. We want the `javafx.stage.Stage` instance to occupy the complete screen size; for this, we will make use of `Screen.getPrimary().getVisualBounds()`. Then, as usual, we will render the scene graph on the stage:

```
Rectangle2D primaryScreenBounds =
            Screen.getPrimary().getVisualBounds();
stage.setTitle("Web Browser");
stage.setScene(scene);
stage.setX(primaryScreenBounds.getMinX());
stage.setY(primaryScreenBounds.getMinY());
stage.setWidth(primaryScreenBounds.getWidth());
stage.setHeight(primaryScreenBounds.getHeight());
stage.show();
```

The complete code can be found at the location, `chp9/10_embed_html`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/10_embed_html`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh`, and you will see the following GUI:


Google
Prev
Next
Refresh
http://www.google.co.in/
Go
Google+
Search
Images
Maps
Play
YouTube
News
Gmail
More
Sign in
Google
India
Advanced search
Language tools
Google Search
I'm Feeling Lucky
Google.co.in offered in: हिन्दी বাংলা తెలుగు मराठी தமிழ் ગુજરાતી ಕನ್ನಡ മലയാളം ਪੰਜਾਬੀ
Advertising Programs
Business Solutions
+Google
About Google
Google.com
© 2017 - Privacy - Terms
Done

How it works...

The web-related APIs are available in the module, `javafx.web`, so we will have to require it in `module-info`:

```
module gui{
  requires javafx.controls;
  requires javafx.web;
  opens com.packt;
}
```

The following are the important classes in `javafx.scene.` web package when dealing with web pages in JavaFX:

- `WebView`: This UI component uses `WebEngine` to manage the loading, rendering, and interaction with the web page
- `WebEngine`: This is the main component that deals with loading and managing the web page
- `WebHistory`: This records the web pages visited in the current `WebEngine` instance
- `WebEvent`: These are the instances passed to the event handlers of `WebEngine` invoked by the JavaScript event

In our recipe, we make use of the first three classes.

We don't directly create an instance of `WebEngine`; instead, we make use of `WebView` to get a reference to the `WebEngine` instance managed by it.
The `WebEngine` instance loads the web page asynchronously by submitting the task of loading the page to `javafx.concurrent.Worker` instances. Then, we register change listeners on these worker instance properties to track the progress of loading the web page. We have made use of two such properties in this recipe, namely, `stateProperty` and `workDoneProperty`. The former tracks the change of the state of the worker, and the latter tracks the percentage of work done.

A worker can go through the following states (as listed in the `javafx.concurrent.Worker.State` enum):

- `CANCELLED`
- `FAILED`
- `READY`
- `RUNNING`
- `SCHEDULED`
- `SUCCEEDED`

In our recipe, we are only checking for `SUCCEEDED`, but you can enhance it to check for `FAILED` as well. This will help us report invalid URLs or even get the message from the event object and show it to the user.

The way we add the listeners to track the change in the properties is by using the `addListener()` method on `*Property()`, where `*` can be `state`, `workDone`, or any other attribute of the worker that has been exposed as a property:

```
webEngine.getLoadWorker().stateProperty().addListener(
  new ChangeListener<State>() {
    public void changed(ObservableValue ov,
      State oldState, State newState) {
        //event handler code here
    }
  }
);

webEngine.getLoadWorker().workDoneProperty().addListener(
  new ChangeListener<Number>(){
    public void changed(ObservableValue ov,
      Number oldState, Number newState) {
        //event handler code here
    }
  }
);
```

Then `javafx.scene.web.WebEngine` component also supports:

- Reloading the current page
- Getting the history of the pages loaded by it
- Executing the JavaScript code
- Listening to JavaScript properties, such as showing an alert box or confirmation box

- Interacting with the document model of the web page using the `getDocument()` method

In this recipe, we also looked at using `WebHistory` obtained from `WebEngine`. `WebHistory` stores the web pages loaded by the given `WebEngine` instance, which means one `WebEngine` instance will have one `WebHistory` instance. `WebHistory` supports the following:

- Getting the list of entries by using the `getEntries()` method. This will also get us the number of entries in the history. This is required while navigating forward and backward in history; otherwise, we will end up with an index out of bounds exception.
- Getting `currentIndex`, that is, its index within the `getEntries()` list.
- Navigating to the specific entry in the entries list of `WebHistory`. This can be achieved by using the `go()` method, which accepts an offset. This offset indicates which web page to load, relative to the current position. For example, *+1* indicates the next entry and *-1* indicates the previous entry. It's important to check for the boundary conditions; otherwise, you will end up with going before *0*, that is, *-1*, or going past the entry list size.

There's more...

In this recipe, we showed you a basic approach to creating a web browser using the support provided by JavaFX. You can enhance this to support the following:

- Better error handling and user messages, that is, to show whether the web address is valid or not by tracking the state change of the worker
- Multiple tabs
- Bookmarking
- Storing the state of the browser locally so that the next time it is run, it loads all the bookmarks and history

Embedding media in an application

JavaFX provides a component, `javafx.scene.media.MediaView`, for viewing videos and listening to audios. This component is backed by a media engine, `javafx.scene.media.MediaPlayer`, which loads and manages the playback of the media.

In this recipe, we will look at playing a sample video and controlling its playback by using the methods on the media engine.

Getting ready

We will make use of the sample video available at `chp9/11_embed_audio_video/sample_video1.mp4`.

How to do it...

1. Let's first create a class with empty methods, which would represent the main application for launching the application as well as the JavaFX UI:

```
public class EmbedAudioVideoDemo extends Application{
  public static void main(String[] args) {
    Application.launch(args);
  }
  @Override
  public void start(Stage stage) {
    //this will have all the JavaFX related code
  }
}
```

2. Create a `javafx.scene.media.Media` object for the video located at `chp9/11_embed_audio_video/sample_video1.mp4`:

```
File file = new File("sample_video1.mp4");
Media media = new Media(file.toURI().toString());
```

3. Create a new media engine, `javafx.scene.media.MediaPlayer`, using the `javafx.scene.media.Media` object created in the previous step:

```
MediaPlayer mediaPlayer = new MediaPlayer(media);
```

4. Let's track the status of the media player by registering a change listener on `statusProperty` of the `javafx.scene.media.MediaPlayer` object:

```
mediaPlayer.statusProperty().addListener(
            new ChangeListener<Status>() {
  public void changed(ObservableValue ov,
                      Status oldStatus, Status newStatus) {
    System.out.println(oldStatus +"->" + newStatus);
  }
});
```

5. Let's now create a media viewer using the media engine created in the previous step:

```
MediaView mediaView = new MediaView(mediaPlayer);
```

6. We will restrict the width and height of the media viewer:

```
mediaView.setFitWidth(350);
mediaView.setFitHeight(350);
```

7. Next, we create three buttons to pause the video playback, resume the playback, and stop the playback. We will make use of the relevant methods in the `javafx.scene.media.MediaPlayer` class:

```
Button pauseB = new Button("Pause");
pauseB.setOnAction(e -> {
  mediaPlayer.pause();
});

Button playB = new Button("Play");
playB.setOnAction(e -> {
  mediaPlayer.play();
});

Button stopB = new Button("Stop");
stopB.setOnAction(e -> {
  mediaPlayer.stop();
});
```

8. Align all these buttons horizontally using `javafx.scene.layout.HBox`:

```
HBox controlsBox = new HBox(10);
controlsBox.getChildren().addAll(pauseB, playB, stopB);
```

9. Align the media viewer and the buttons bar vertically using `javafx.scene.layout.VBox`:

```
VBox vbox = new VBox();
vbox.getChildren().addAll(mediaView, controlsBox);
```

10. Create a new scene graph using the `VBox` object as the root and set it to the stage object:

```
Scene scene = new Scene(vbox);
stage.setScene(scene);
// Name and display the Stage.
stage.setTitle("Media Demo");
```

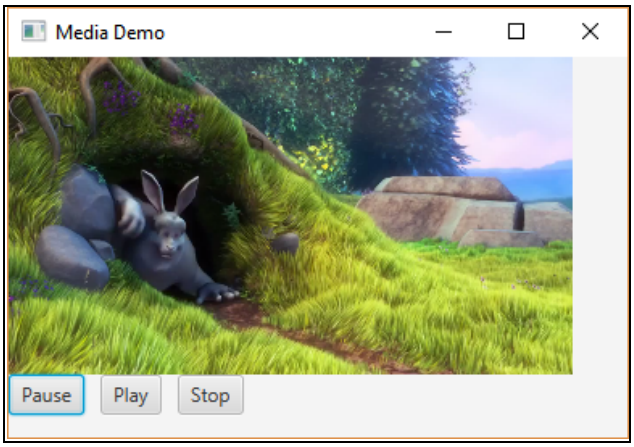
11. Render the stage on the display:

```
stage.setWidth(400);
stage.setHeight(400);
stage.show();
```

The complete code can be found at `chp9/11_embed_audio_video`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/11_embed_audio_video`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh`, and you will see the following GUI:



How it works...

The important classes, in the `javafx.scene.media` package for media playback are as follows:

- `Media`: This represents the source of the media, that is, either the video or audio. This accepts the source in the form of HTTP/HTTPS/FILE and JAR URLs.
- `MediaPlayer`: This manages the playback of the media.
- `MediaView`: This is the UI component that allows viewing the media.

There are a few other classes, but we haven't covered them in this recipe. The media-related classes are in the `javafx.media` module. So, do not forget to require a dependency on it, as shown here:

```
module gui{
  requires javafx.controls;
  requires javafx.media;
  opens com.packt;
}
```

In this recipe, we have a sample video at `chp9/11_embed_audio_video/sample_video1.mp4`, and we make use of the `java.io.File` API to build `File` URL to locate the video:

```
File file = new File("sample_video1.mp4");
Media media = new Media(file.toURI().toString());
```

The media playback is managed by using the API exposed by the `javafx.scene.media.MediaPlayer` class. In this recipe, we made use of a few of its methods, namely `play()`, `pause()`, and `stop()`. The `javafx.scene.media.MediaPlayer` class is initialized by using the `javafx.scene.media.Media` object:

```
MediaPlayer mediaPlayer = new MediaPlayer(media);
```

Rendering the media on the UI is managed by the `javafx.scene.media.MediaView` class, and it is backed by a

`javafx.scene.media.MediaPlayer` object:

```
MediaView mediaView = new MediaView(mediaPlayer);
```

We can set the height and width of the viewer by using the `setFitWidth()` and `setFitHeight()` methods.

There's more...

We gave a basic demo of media support in JavaFX. There's a lot more to explore. You can add volume control options, options to seek forward or backward, playing of audios, and audio equalizer.

Adding effects to controls

Adding effects in a controlled way gives a good appearance to the user interface. There are multiple effects like blurring, shadows, reflection, blooming, and so on. JavaFX provides a set of classes under the `javafx.scene.effects` package, which can be used to add effects to enhance the look of the application. This package is available in the `javafx.graphics` module.

In this recipe, we will look at a few effects: blur, shadow, and reflection.

How to do it...

1. Let's first create a class with empty methods, which would represent the main application for launching the application as well as the JavaFX UI:

```
public class EffectsDemo extends Application{
  public static void main(String[] args) {
    Application.launch(args);
  }
  @Override
  public void start(Stage stage) {
    //code added here in next steps
  }
}
```

2. The subsequent code will be written within the `start(Stage stage)` method. Create and configure `javafx.scene.layout.GridPane`:

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));
```

3. Create rectangles required for applying the blur effects:

```
Rectangle r1 = new Rectangle(100,25, Color.BLUE);
Rectangle r2 = new Rectangle(100,25, Color.RED);
Rectangle r3 = new Rectangle(100,25, Color.ORANGE);
```

4. Add `javafx.scene.effect.BoxBlur` to `Rectangle r1`, `javafx.scene.effect.MotionBlur` to `Rectangle r2`, and `javafx.scene.effect.GaussianBlur` to `Rectangle r3`:

```
r1.setEffect(new BoxBlur(10,10,3));
r2.setEffect(new MotionBlur(90, 15.0));
r3.setEffect(new GaussianBlur(15.0));
```

5. Add the rectangles to `gridPane`:

```
gridPane.add(r1,1,1);
gridPane.add(r2,2,1);
gridPane.add(r3,3,1);
```

6. Create three circles, required for applying shadows:

```
Circle c1 = new Circle(20, Color.BLUE);
Circle c2 = new Circle(20, Color.RED);
Circle c3 = new Circle(20, Color.GREEN);
```

7. Add `javafx.scene.effect.DropShadow` to `c1` and `javafx.scene.effect.InnerShadow` to `c2`:

```
c1.setEffect(new DropShadow(0, 4.0, 0, Color.YELLOW));
c2.setEffect(new InnerShadow(0, 4.0, 4.0, Color.ORANGE));
```

8. Add these circles to `gridPane`:

```
gridPane.add(c1,1,2);
gridPane.add(c2,2,2);
gridPane.add(c3,3,2);
```

9. Create a simple text, `Reflection Sample`, on which we will apply the reflection effect:

```
Text t = new Text("Reflection Sample");
t.setFont(Font.font("Arial", FontWeight.BOLD, 20));
t.setFill(Color.BLUE);
```

10. Create a `javafx.scene.effect.Reflection` effect and add it to the text:

```
Reflection reflection = new Reflection();
reflection.setFraction(0.8);
t.setEffect(reflection);
```

11. Add the text component to `gridPane`:

```
gridPane.add(t, 1, 3, 3, 1);
```

12. Create a scene graph using `gridPane` as the root node:

```
Scene scene = new Scene(gridPane, 500, 300);
```

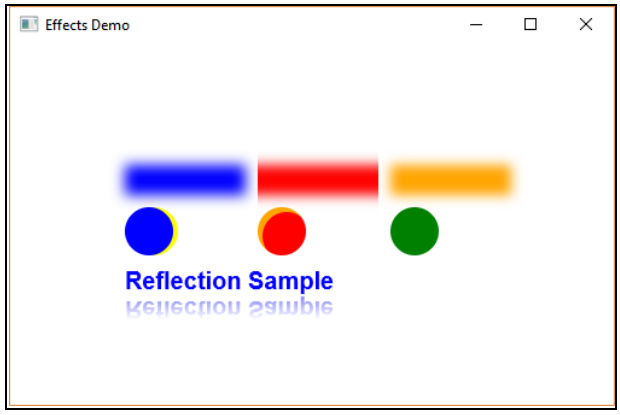
13. Set the scene graph to the stage and render it on the display:

```
stage.setScene(scene);
stage.setTitle("Effects Demo");
stage.show();
```

The complete code can be found at `chp9/12_effects_demo`.

We have provided two run scripts, `run.bat` and `run.sh`, under `chp9/12_effects_demo`. The `run.bat` script will be for running the application on Windows and `run.sh` will be for running the application on Linux.

Run the application using `run.bat` or `run.sh` and you will see the following GUI:



How it works...

In this recipe, we have made use of the following effects:

- `javafx.scene.effect.BoxBlur`
- `javafx.scene.effect.MotionBlur`
- `javafx.scene.effect.GaussianBlur`
- `javafx.scene.effect.DropShadow`
- `javafx.scene.effect.InnerShadow`
- `javafx.scene.effect.Reflection`

The `BoxBlur` effect is created by specifying the width and height of the blur effect, and also the number of times the effect needs to be applied:

```
BoxBlur boxBlur = new BoxBlur(10,10,3);
```

The `MotionBlur` effect is created by providing the angle of the blur and its radius. This gives an effect of something captured while in motion:

```
MotionBlur motionBlur = new MotionBlur(90, 15.0);
```

The `GaussianBlur` effect is created by providing the radius of the effect, and the effect uses the Gaussian formula to apply the effect:

```
GaussianBlur gb = new GaussianBlur(15.0);
```

The `DropShadow` adds the shadow behind the object whereas `InnerShadow` adds the shadow within the object. Each of these takes the radius of the shadow, the *x* and *y* location of the start of shadow, and the color of the shadow:

```
DropShadow dropShadow = new DropShadow(0, 4.0, 0, Color.YELLOW);
InnerShadow innerShadow = new InnerShadow(0, 4.0, 4.0, Color.ORANGE);
```

`Reflection` is a pretty simple effect, which adds the reflection of the object. We can set the fraction of how much of the original object is reflected:

```
Reflection reflection = new Reflection();
reflection.setFraction(0.8);
```

There's more...

There are quite a few more effects:

- The blend effect, which blends two different inputs with a predefined blending approach
- The bloom effect, which makes the brighter portions appear brighter
- The glow effect, which makes the object glow
- Lighting effect, which simulates a light source on the object thereby giving it a 3D appearance.

We would recommend you to try out these effects in the same way as we have tried them out.

Using the new TIFF I/O API to read TIFF images

Tagged Image File Format (**TIFF**) is a common image format for exchanging images between applications. Previously, JDK didn't have any support for reading TIFF images, and developers had to use the Java Image API, which was external to JDK.

In this recipe, we will show you how to read a TIFF image file.

Getting ready

We have a sample TIFF image file at the location, `chp9/13_tiff_reader/sample.tif`.

How to do it...

1. Get the image readers by format name, which is `tiff` for TIFF images

```
Iterator iterator = ImageIO.getImageReadersByFormatName("tiff");
```

2. Get the first `ImageReader` object from the image readers obtained in the previous step:

```
ImageReader reader = (ImageReader) iterator.next();
```

3. Create a `FileImageInputStream` object for the `sample.tif` image:

```
try(ImageInputStream is =
        new FileImageInputStream(new File("sample.tif"))) {
  //image reading code here.
} catch (Exception ex){
  //exception handling
}
```

4. Use the reader obtained to read the image file:

```
reader.setInput(is, false, true);
```

5. Let's get some attributes, such as the number of images, width, and height, just to confirm that we have really read the image:

```
System.out.println("Number of Images: " +
                   reader.getNumImages(true));
System.out.println("Height: " + reader.getHeight(0));
System.out.println("Width: " + reader.getWidth(0));
System.out.println(reader.getFormatName());
```

The complete code for this can be found at `chp9/13_tiff_reader`. You can run the sample either by using `run.bat` or `run.sh`.

RESTful Web Services Using Spring Boot

In this chapter, we are going to cover the following recipes:

- Creating a simple Spring Boot application
- Interacting with the database
- Creating a RESTful web service
- Creating multiple profiles for Spring Boot
- Deploying RESTful web services to Heroku
- Containerizing the RESTful web service using Docker

Introduction

In recent years, the drive for microservice-based architecture has gained wide adoption, thanks to the simplicity and ease of maintenance it provides when done the right way. A lot of companies, such as Netflix, Amazon, and the like, have moved from monolithic systems to more focused and lighter systems, all talking with each other via RESTful web services. The advent of RESTful web services and its straightforward approach to creating web services using the known HTTP protocol has made it easier for communication between applications than the older SOAP-based web services.

In this chapter, we will look at the **Spring Boot** framework, which provides a convenient way to create production-ready microservices using Spring libraries. Using Spring Boot, we will develop a simple RESTful web service and deploy the same on the cloud.

Creating a simple Spring Boot application

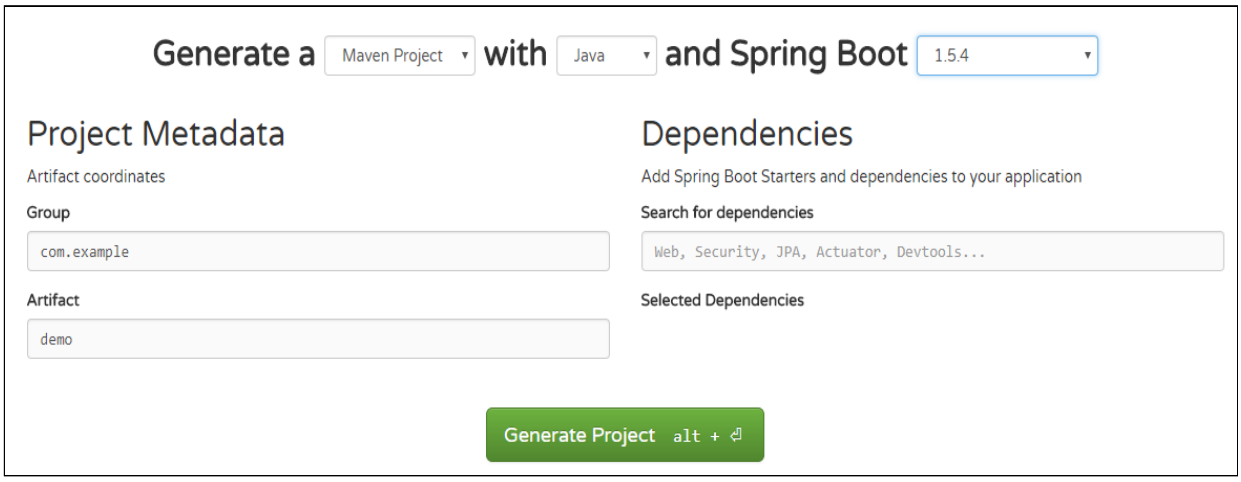
Spring Boot helps in creating production-ready Spring-based applications easily. It provides support for working with almost all Spring libraries, without any need for configuring them explicitly. There are autoconfiguration classes provided for easy integration with most commonly used libraries, databases, message queues, and the likes.

In this recipe, we will look at creating a simple Spring Boot application with a controller that prints a message when opened in the browser.

Getting ready

Spring Boot supports Maven and Gradle as the build tools and we will be using Maven in our recipes. The URL, `http://start.spring.io/`, provides a convenient way to create an empty project with the required dependencies. We will use it to download an empty project. Follow these steps to create and download an empty Spring Boot-based project:

1. Navigate to `http://start.spring.io/` to see something similar to the following screenshot:



2. You can select the dependency management and build tool, selecting the appropriate option in the dropdown after the text, **Generate a**.
3. Spring Boot supports Java, Kotlin, and Groovy. You can choose the language by changing the dropdown after the text, **with**.

4. Select the Spring Boot version by choosing its value from the dropdown after the text, and Spring Boot. For this recipe, we'll use the latest milestone edition of Spring Boot 2 I.E 2.0.0 M2.
5. On the left-hand side, under Project Metadata, we have to provide Maven-related information, that is, the group ID and artifact ID. We'll

use Group as `com.packt` and Artifact as `boot_demo`.

6. On the right-hand side, under Dependencies, you can search for the dependencies you want to add. For this recipe, we need web and Thymeleaf dependencies. This means that we want to create a web application which uses Thymeleaf UI templates and would want all the dependencies, such as Spring MVC, Embedded Tomcat, and others, to be part of the application.
7. Then, click on the Generate Project button to download the empty project. You can load this empty project in any IDE of your choice, just like any other Maven project.

At this point, you will have your empty project loaded into an IDE of your choice and ready to explore further. In this recipe, we will make use of the Thymeleaf template engine to define our web pages and create a simple controller to render the web page.

The complete code for this recipe can be found at the location, `chp10/1_boot_demo`.

How to do it...

1. If you have followed the group ID and artifact ID naming as mentioned in the *Getting ready* section, you will have a package structure, `com.packt.boot_demo`, and a `BootDemoApplication.java` main class already created for you. There will be an equivalent package structure and a `BootDemoApplicationTests.java` main class under the `tests` folder.
2. Create a new class, `SimpleViewController`, under the `com.packt.boot_demo` package, with the following code:

```
@Controller
public class SimpleViewController{
  @GetMapping("/message")
  public String message(){
    return "message";
  }
}
```

3. Create a web page, `message.html`, under the location, `src/main/resources/templates`, with the following code:

```
<h1>Hello, this is a message from the Controller</h1>
<h2>The time now is [[${#dates.createNow()}]]</h2>
```

4. From the command prompt, navigate to the project root folder and issue the command, `mvn spring-boot:run`; you'll see the application being launched. Once it completes the initialization and starts, it would be running on the default port, `8080`. Navigate to `http://localhost:8080/message` to see the message.

We are using Spring Boot's Maven plugin, which provides us with convenient tools to launch the application during development. But for production, we will create a fat JAR, that is, a JAR comprising all the dependencies, and deploy it as a Linux or Windows service. We can even run the fat JAR using the `java -jar` command.

How it works...

We will not go into the working of Spring Boot or the other Spring libraries. But to state in brief, Spring Boot creates an embedded Tomcat running on the default port, that is, `8080`. It then registers all the controllers, components, and services that are available in the packages and sub packages of the class with the annotation, `@SpringBootApplication`.

In our recipe, the `BootDemoApplication` class in the `com.packt.boot_demo` package is annotated with `@SpringBootApplication`. So, all the classes that are annotated with `@Controller`, `@Service`, `@Configuration`, `@Component`, and the likes of it get registered with the Spring framework as beans and are managed by it. Now, these can be injected into the code by using the `@Autowired` annotation.

There are two ways we can create a web controller:

1. Annotating with `@Controller`
2. Annotating with `@RestController`

In the first approach, we create a controller that can serve both raw data and HTML data (generated by template engines such as Thymeleaf, Freemarker, JSP, and others). In the second approach, the controller supports endpoints that can only serve raw data in the form of JSON or XML. In our recipe, we used the former approach, as follows:

```
@Controller
public class SimpleViewController{
  @GetMapping("/message")
  public String message(){
    return "message";
  }
}
```

We can annotate the class with `@RequestMapping` with,
say, `@RequestMapping("/api")`. In this case, any HTTP endpoints exposed in the controller are prepended by `/api`. There is a specialized annotation mapping for HTTP `GET`, `POST`, `DELETE`, and `PUT` methods, namely `@GetMapping`, `@PostMapping`,

`@DeleteMapping`, and `@PutMapping`, respectively. We can also rewrite our controller class as follows:

```
@Controller
@RequestMapping("/message")
public class SimpleViewController{
  @GetMapping
  public String message(){
    return "message";
  }
}
```

We can modify the port by providing `server.port = 9090` in the `application.properties` file. This file can be found in the location, `src/main/resources/application.properties`. There is a whole set of properties (http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) that we can use to customize and connect with different components.

Interacting with the database

In this recipe, we will look at how to integrate with a database to create, read, modify, and delete the data. For this, we will set up a MySQL database with the required table. Subsequently, we will update the data in a table from our Spring Boot application.

We will be using Windows as the platform of development for this recipe. You can perform a similar action on Linux as well, but you would first have to set up your MySQL database.

Getting ready

Before we start integrating our application with the database, we need to set up the database locally on our development machines. In the subsequent sections, we will download and install MySQL tools and then create a sample table with some data, which we will use with our application.

Installing MySQL tools

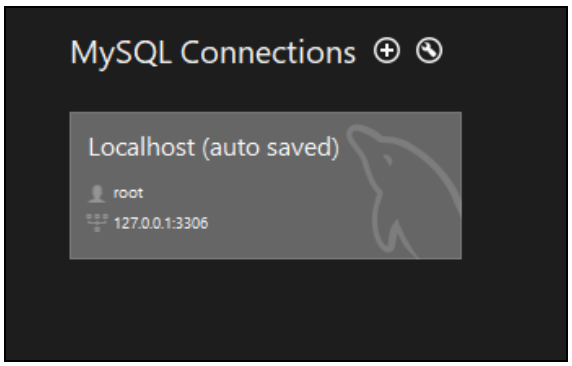
First, download the MySQL installer from `https://dev.mysql.com/downloads/windows/installer/5.7.html`. This MySQL bundle is for Windows only. Follow the onscreen instructions to successfully install MySQL along with other tools such as MySQL Workbench. To confirm that the MySQL daemon (`mysqld`) is running, open the task manager and you should be able to see a process similar to the one shown in the following image:



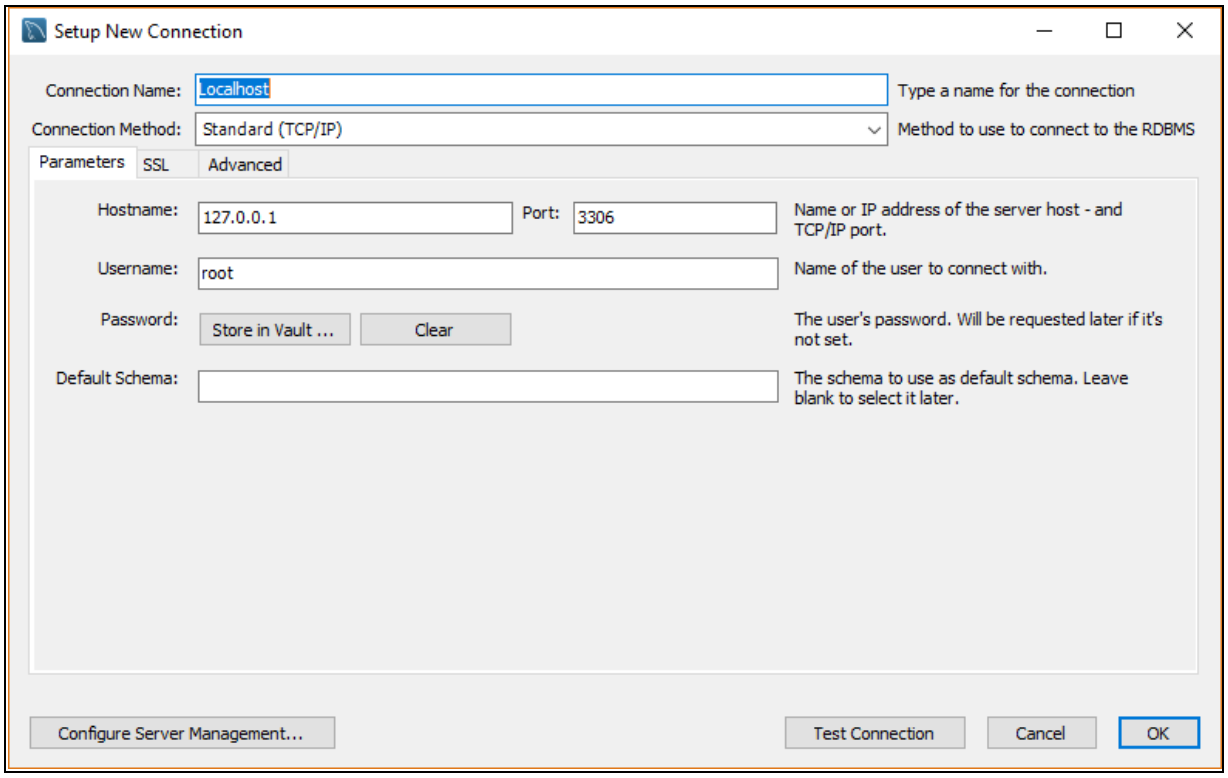


You should remember the password you set for the root user.

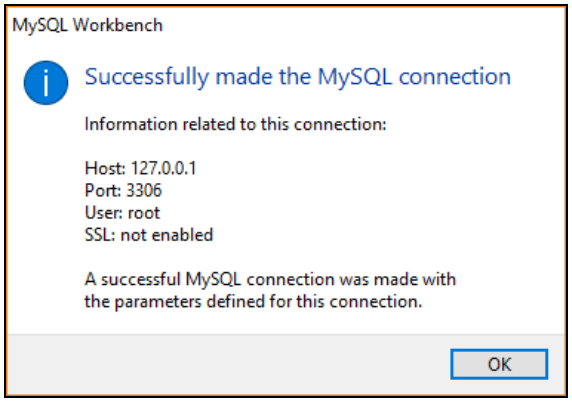
Let's run the MySQL workbench; on starting up, you should be able to see something similar to the following image, among other things provided by the tool:



If you don't find a connection like the preceding image, you can add one using the (+) sign. On clicking on (+), you will see the following dialog. Fill it up and click on Test Connection to get a success message:



A successful Test Connection will result in the following message:



Double-click on the connection to connect to the database, and you should see a list of DBs on the left-hand side, an empty area on the right-hand side, and menu and toolbars on the top. From the File menu, click on New Query Tab or, alternatively, press *Ctrl* + *T* to get a new query window. Here, we will write our queries to create a database and create a table within that database.



The bundled installer downloaded from https://dev.mysql.com/downloads/windows/installer/5.7.html is for Windows only. Linux users have to download the MySQL Server and MySQL Workbench (GUI for interacting with DB) separately.
The MySQL server can be downloaded from https://dev.mysql.com/downloads/mysql/.
The MySQL Workbench can be downloaded from https://dev.mysql.com/downloads/workbench/.

Creating a sample database

Run the following SQL statement to create a database:

```
create database sample;
```

Creating a person table

Run the following SQL statements to use the newly created database and create a simple person table:

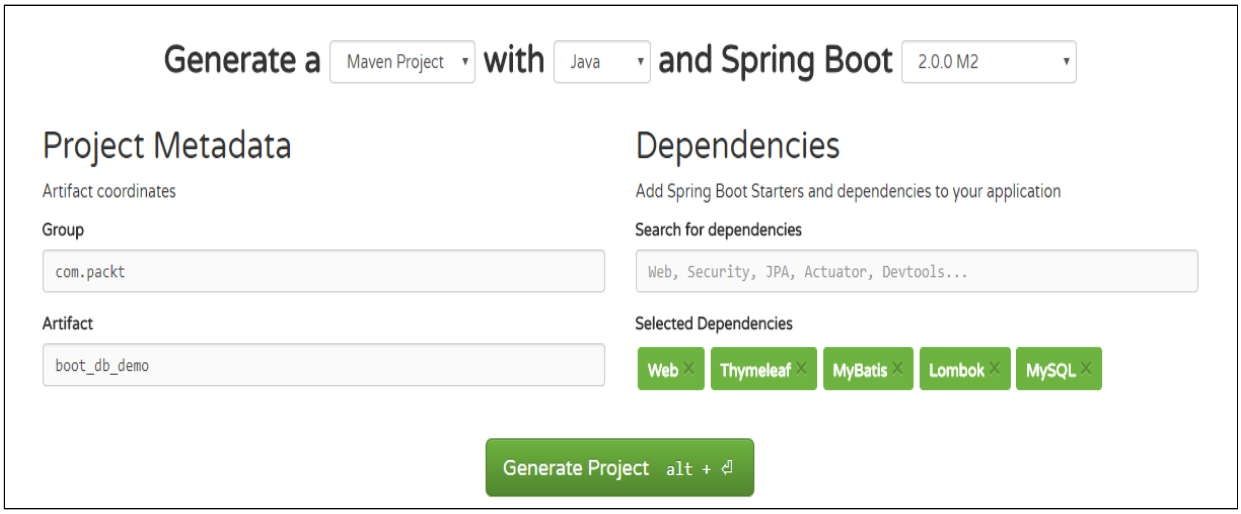
```
create table person(
  id int not null auto_increment,
  first_name varchar(255),
  last_name varchar(255),
  place varchar(255),
  primary key(id)
);
```

Populating sample data

Let's go ahead and insert some sample data in the table we just created:

```
insert into person(first_name, last_name, place)
values('Raj', 'Singh', 'Bangalore');

insert into person(first_name, last_name, place)
values('David', 'John', 'Delhi');
```

Now that we have our database ready, we will go ahead and download the empty Spring Boot project from http://start.spring.io/ with the following options:



How to do it...

1. Create a model class, `com.packt.boot_db_demo.Person`, for representing a person. We will make use of Lombok annotations to generate the getters and setters for us:

```
@Data
public class Person{
  private Integer id;
  private String firstName;
  private String lastName;
  private String place;
}
```

2. Let's create `com.packt.boot_db_demo.PersonMapper` to map the data from the database into our model class, `Person`:

```
@Mapper
public interface PersonMapper {
}
```

3. Let's add a method to get all the rows from the table. Note that the next few methods will be written inside the `PersonMapper` interface:

```
@Select("SELECT * FROM person")
public List<Person> getPersons();
```

4. Another method to get the details of a single person identified by ID is as follows:

```
@Select("SELECT * FROM person WHERE id = #{id}")
public Person getPerson(Integer id);
```

5. The method to create a new row in the table is as follows:

```
@Insert("INSERT INTO person(first_name, last_name, place) " +
        " VALUES (#{firstName}, #{lastName}, #{place})")
@Options(useGeneratedKeys = true)
public void insert(Person person);
```

6. The method to update an existing row in the table, identified by the ID is as follows:

```
@Update("UPDATE person SET first_name = #{firstName}, last_name =
        #{lastName}, "+ "place = #{place}  WHERE id = #{id} ")
public void save(Person person);
```

7. Finally, the method to delete a row from the table, identified by the ID is as follows:

```
@Delete("DELETE FROM person WHERE id = #{id}")
public void delete(Integer id);
```

8. Let's create a `com.packt.boot_db_demo.PersonController` class, which we will use to write our web endpoints:

```
@Controller
@RequestMapping("/persons")
public class PersonContoller {
  @Autowired PersonMapper personMapper;
}
```

9. Let's create an endpoint to list all the entries in the `person` table:

```
@GetMapping
public String list(ModelMap model){
  List<Person> persons = personMapper.getPersons();
  model.put("persons", persons);
  return "list";
}
```

10. Let's create an endpoint to add a new row in the `person` table:

```
@GetMapping("/{id}")
public String detail(ModelMap model, @PathVariable Integer id){
  System.out.println("Detail id: " + id);
  Person person = personMapper.getPerson(id);
  model.put("person", person);
  return "detail";
}
```

11. Let's create an endpoint to add a new row or edit an existing row in the `person` table:

```
@PostMapping("/form")
public String submitForm(Person person){
  System.out.println("Submiting form person id: " +
                     person.getId());
  if ( person.getId() != null ){
    personMapper.save(person);
  }else{
    personMapper.insert(person);
  }
```

```
    return "redirect:/persons/";
  }
```

12. Let's create an endpoint to delete a row from the `person` table:

```
@GetMapping("/{id}/delete")
public String deletePerson(@PathVariable Integer id){
  personMapper.delete(id);
  return "redirect:/persons";
}
```

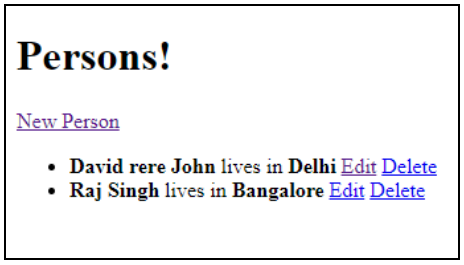
13. Finally, we need to update the `src/main/resources/application.properties` file to provide the configuration related to our data source, that is, our MySQL database:

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/sample?useSSL=false
spring.datasource.username=root
spring.datasource.password=mohamed
mybatis.configuration.map-underscore-to-camel-case=true
```

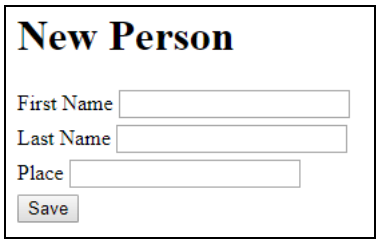
You can run the application from the command line using `mvn spring-boot:run`. This application starts up on the default port, that is, `8080`. Navigate to `http://localhost:8080/persons` in your browser.

The complete code for this recipe can be found at the location, `chp10/2_boot_db_demo`.

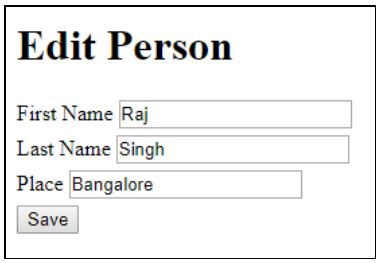
On visiting `http://localhost:8080/persons`, this is what you will find:



On clicking on **New Person**, you'll get the following:



On clicking on **Edit**, you'll get the following:



How it works...

Firstly, `com.packt.boot_db_demo.PersonMapper` annotated with `org.apache.ibatis.annotations.Mapper` knows how to execute the query provided within the `@Select`, `@Update`, or `@Delete` annotations and to return relevant results. This is all managed by the MyBatis and Spring Data libraries.

You must be wondering how the connection to database was achieved. One of the Spring Boot autoconfiguration classes, `DataSourceAutoConfiguration`, does the work of setting up by making use of the `spring.datasource.*` properties defined in your `application.properties` file to give us an instance of `javax.sql.DataSource`. This `javax.sql.DataSource` object is then used by the MyBatis library to provide you with an instance of `SqlSessionTemplate`, which is what is used by our `PersonMapper` under the hood.

Then, we make use of `com.packt.boot_db_demo.PersonMapper` by injecting it into the `com.packt.boot_db_demo.PersonController` class by using `@AutoWired`. The `@AutoWired` annotation looks for any Spring managed beans, which are either instances of the exact type or its implementation. Take a look at the *Creating a simple Spring Boot application* recipe in this chapter to understand the `@Controller` annotation.

With very little configuration, we have been able to quickly set up simple CRUD operations. This is the flexibility and agility that Spring Boot provides to developers!

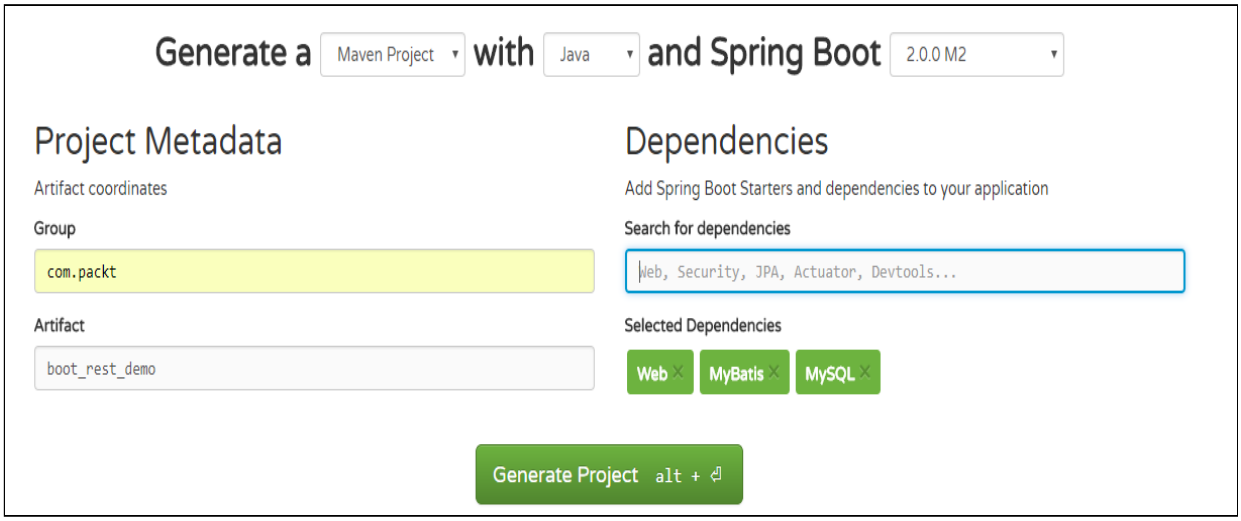
Creating a RESTful web service

In our previous recipe, we interacted with data using web forms. In this recipe, we will see how to interact with data using RESTful web services. These web services are a means to interact with other applications using the known HTTP protocol and its methods, namely GET, POST, PUT, and others. The data can be exchanged in the form of XML, JSON, or even plain text. We will be using JSON in our recipe.

So, we will create RESTful APIs to support retrieving data, creating new data, editing data, and deleting data.

Getting ready

As usual, download the starter project from `http://start.spring.io/` by selecting the dependencies as shown in the following screenshot:



How to do it...

1. We will copy the `Person` class from the previous recipe:

```
public class Person {
  private Integer id;
  private String firstName;
  private String lastName;
  private String place;
  //required getters and setters
}
```

2. We will do the `PersonMapper` part in a different way. We will write all our SQL queries in a mapper XML file and then refer to them from the `PersonMapper` interface. We will place the mapper XML under the `src/main/resources/mappers` folder. We'll set the value of the `mybatis.mapper-locations` property to `classpath*:mappers/*.xml`. This way, the `PersonMapper` interface can discover the SQL queries corresponding to its methods.
3. First, let's create the `com.packt.boot_rest_demo.PersonMapper` interface:

```
@Mapper
public interface PersonMapper {
  public List<Person> getPersons();
  public Person getPerson(Integer id);
  public void save(Person person);
  public void insert(Person person);
  public void delete(Integer id);
}
```

4. Now, let's create the SQL in `PersonMapper.xml`. One thing to make sure of is that the `namespace` attribute of the `<mapper>` tag should be the same as the fully qualified name of the `PersonMapper` mapper interface:

```
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.packt.boot_rest_demo.PersonMapper">
  <select id="getPersons"
   resultType="com.packt.boot_rest_demo.Person">
    SELECT id, first_name firstname, last_name lastname, place
    FROM person
  </select>

  <select id="getPerson"
   resultType="com.packt.boot_rest_demo.Person"
```

```
     parameterType="long">
      SELECT id, first_name firstname, last_name lastname, place
      FROM person
      WHERE id = #{id}
    </select>

    <update id="save"
     parameterType="com.packt.boot_rest_demo.Person">
      UPDATE person SET
        first_name = #{firstName},
        last_name = #{lastName},
        place = #{place}
      WHERE id = #{id}
    </update>

    <insert id="insert"
     parameterType="com.packt.boot_rest_demo.Person"
     useGeneratedKeys="true" keyColumn="id" keyProperty="id">
      INSERT INTO person(first_name, last_name, place)
      VALUES (#{firstName}, #{lastName}, #{place})
    </insert>

    <delete id="delete" parameterType="long">
      DELETE FROM person WHERE id = #{id}
    </delete>
  </mapper>
```

5. We will define the application properties in the `src/main/resources/application.properties` file:

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/sample?useSSL=false
spring.datasource.username=root
spring.datasource.password=mohamed
mybatis.mapper-locations=classpath*:mappers/*.xml
```

6. Create an empty controller for our REST APIs. This controller will be marked with the `@RestController` annotation because all the APIs in it are going to deal solely with data:

```
@RestController
@RequestMapping("/api/persons")
public class PersonApiController {
  @Autowired PersonMapper personMapper;
}
```

7. Let's add an API to list all the rows in the `person` table:

```
@GetMapping
public ResponseEntity<List<Person>> getPersons(){
  return new ResponseEntity<>(personMapper.getPersons(),
                              HttpStatus.OK);
}
```

8. Let's add an API to get the details of a single person:

```
@GetMapping("/{id}")
public ResponseEntity<Person> getPerson(@PathVariable Integer id){
  return new ResponseEntity<>(personMapper.getPerson(id),
                              HttpStatus.OK);
}
```

9. Let's add an API to add new data to the table:

```
@PostMapping
public ResponseEntity<Person> newPerson
               (@RequestBody Person person){
  personMapper.insert(person);
  return new ResponseEntity<>(person, HttpStatus.OK);
}
```

10. Let's add an API to edit the data in the table:

```
@PostMapping("/{id}")
public ResponseEntity<Person> updatePerson
               (@RequestBody Person person,
  @PathVariable Integer id){
    person.setId(id);
    personMapper.save(person);
    return new ResponseEntity<>(person, HttpStatus.OK);
  }
```

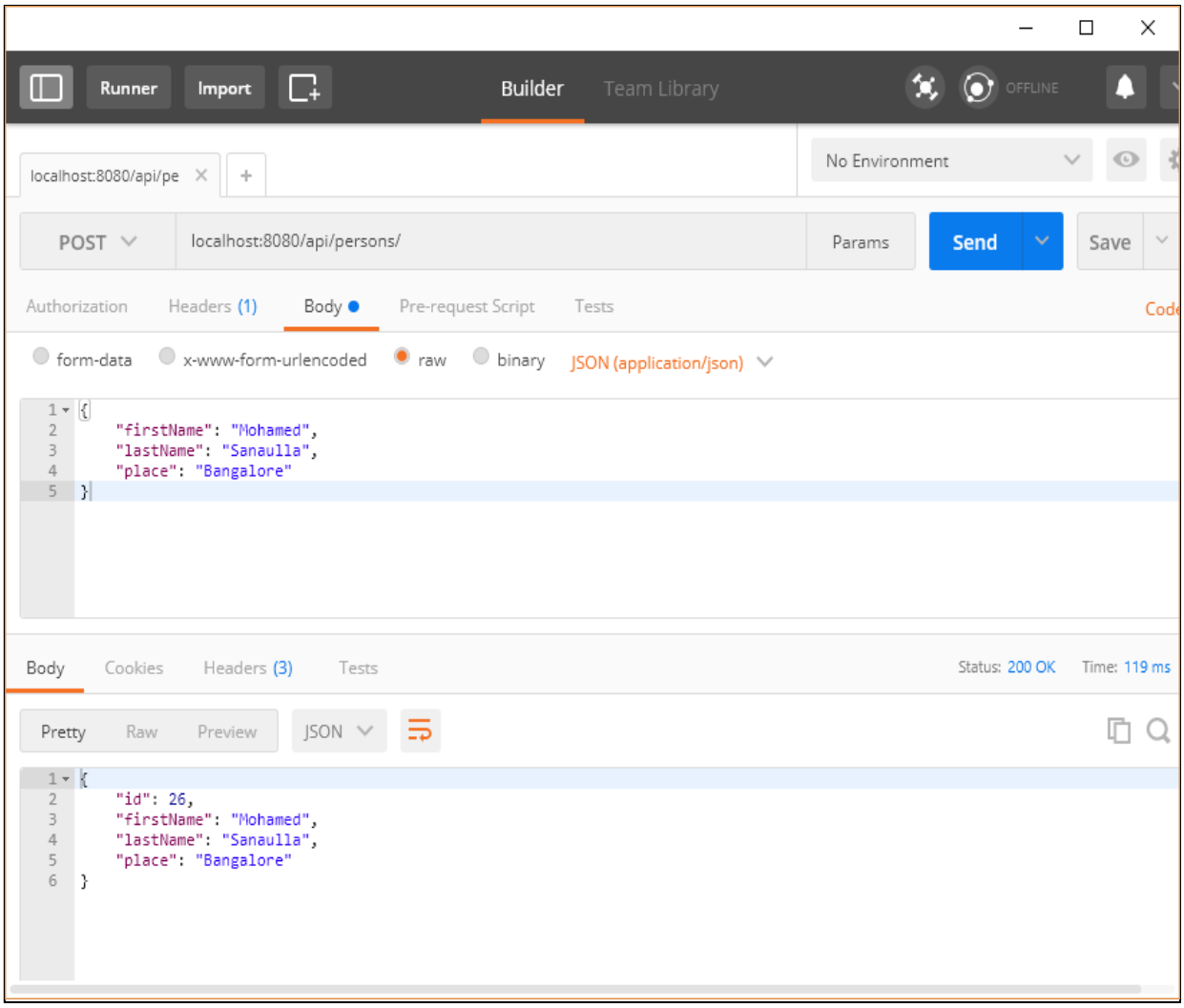
11. Let's add an API to delete the data in the table:

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> deletePerson
               (@PathVariable Integer id){
  personMapper.delete(id);
  return new ResponseEntity<>(HttpStatus.OK);
}
```

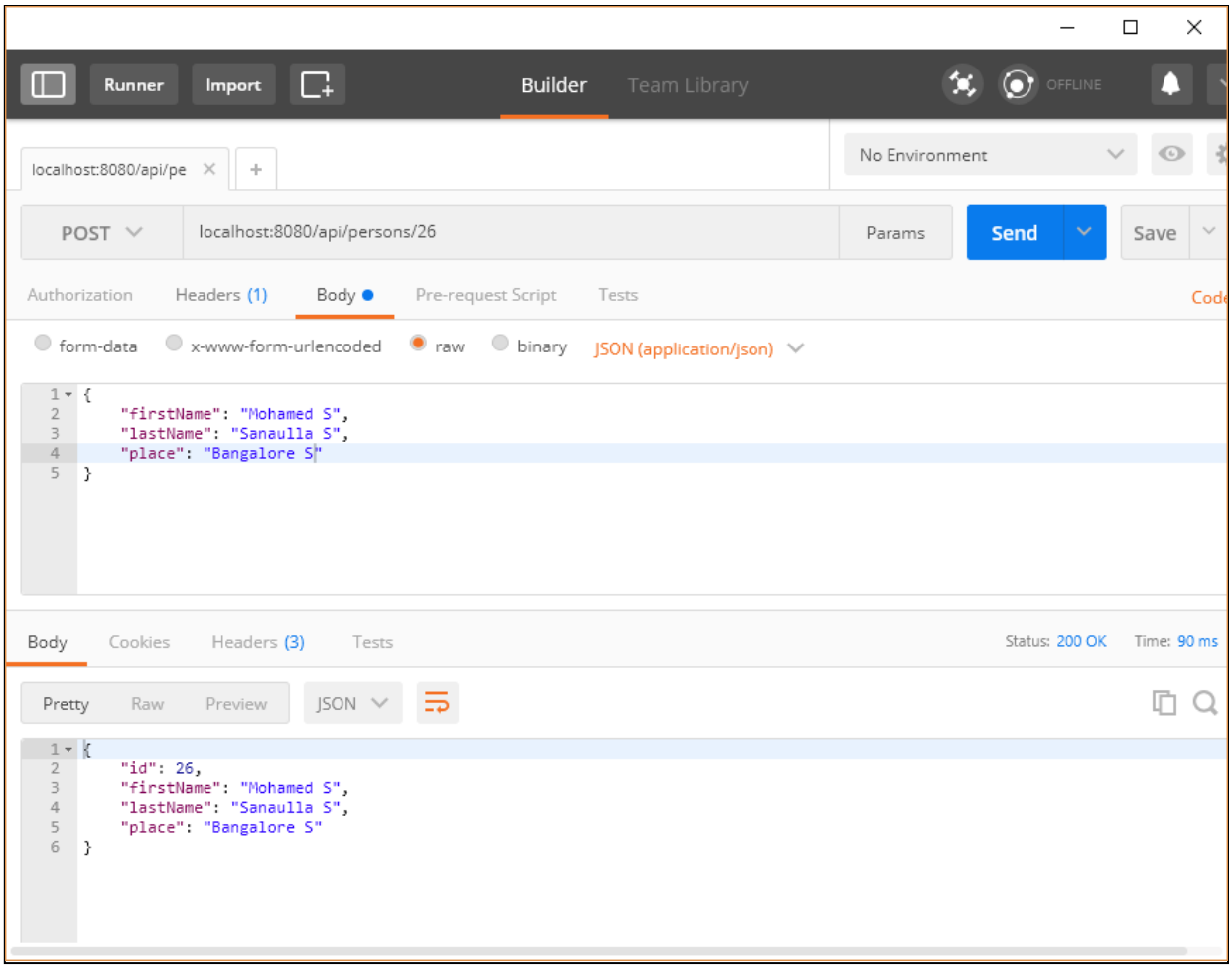
You can find the complete code at the location, `chp10/3_boot_rest_demo`. You can launch the application by using `mvn spring-boot:run` from the project folder. Once the application has started, navigate to `http://localhost:8080/api/persons` to view all the data in the person table.

To test the other APIs, we will make use of the Postman REST client app for Google Chrome.

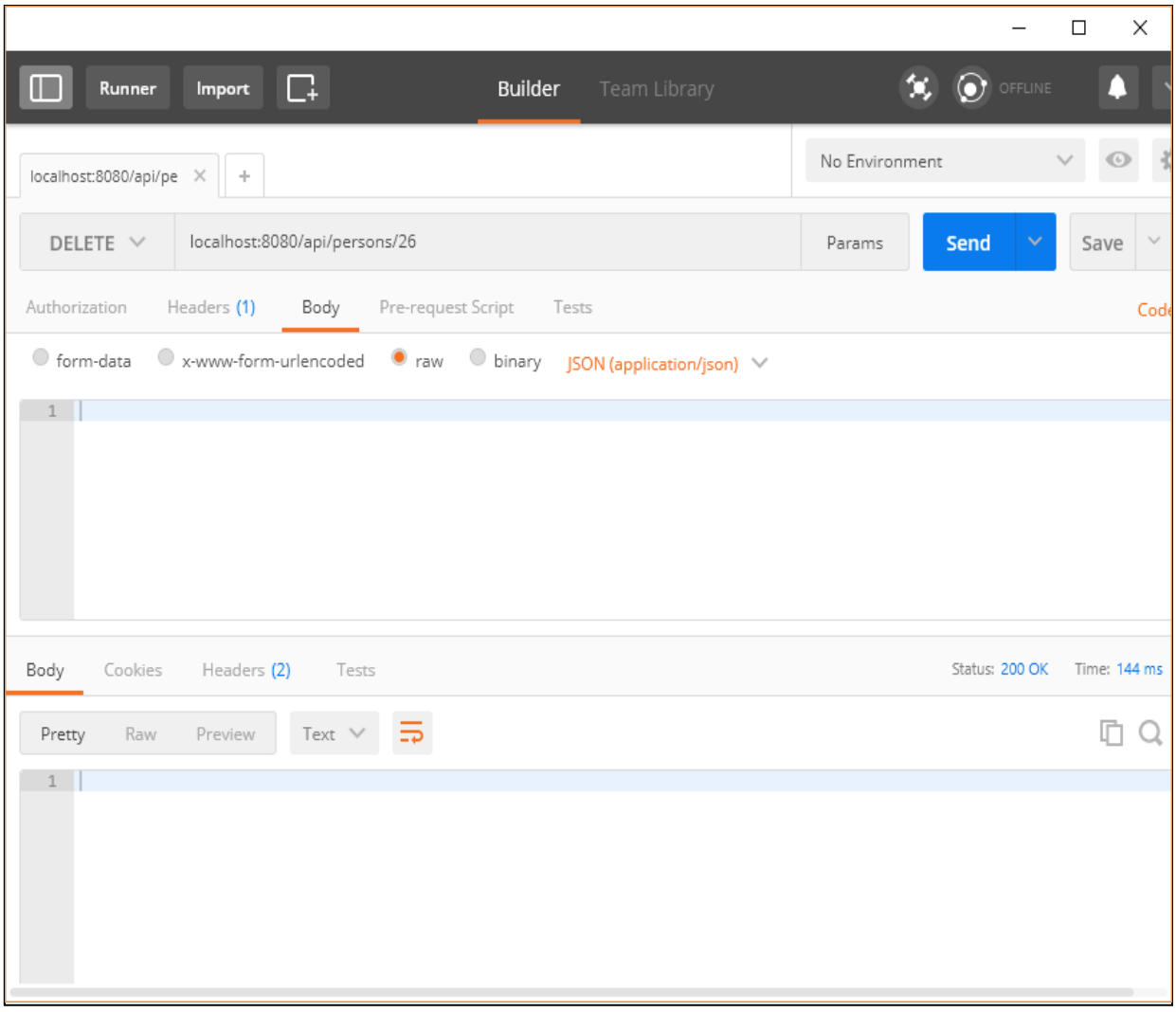
This is what adding a new person looks like. Look at the request body, that is, the person detail specified in JSON:



This is how we edit a person's detail:



This is what deleting a person looks like:


Runner
Import
Builder
Team Library
OFFLINE
localhost:8080/api/pe
No Environment
DELETE
localhost:8080/api/persons/26
Params
Send
Save
Authorization
Headers (1)
Body
Pre-request Script
Tests
form-data
x-www-form-urlencoded
raw
binary
JSON (application/json)
1
Body
Cookies
Headers (2)
Tests
Status: 200 OK
Time: 144 ms
Pretty
Raw
Preview
Text
1

How it works...

First, let's look at how the `PersonMapper` interface discovers the SQL statements to execute. If you look at `src/main/resources/mappers/PersonMapper.xml`, you will find that the `<mapper>` `namespace` attribute is `org.packt.boot_rest_demo.PersonMapper`. This is a requirement that the value of `namespace` attribute should be the fully qualified name of the mapper interface, which, in our case, is `org.packt.boot_rest_demo.PersonMapper`.

Next the `id` attributes of the individual SQL statements defined within `<select>`, `<insert>`, `<update>`, and `<delete>` should match the name of the method in the mapper interface. For example, the `getPersons()` method in the `PersonMapper` interface looks for an SQL statement with `id="getPersons"`.

Now the MyBatis library discovers the location of this mapper XML by reading the value of the `mybatis.mapper-locations` property.

Coming to the controller, we have introduced a new annotation, `@RestController`. This special annotation indicates, in addition to it being a web controller, that all the methods defined in the class return a response that is sent via the HTTP response body; so do all the REST APIs. They just work with the data.

As usual, you can launch your Spring Boot application either by using the Maven Spring-Boot plugin, `mvn spring-boot:run` or by executing the JAR created by the Maven package, `java -jar my_jar_name.jar`.

Creating multiple profiles for Spring Boot

Generally, web applications are deployed on different environments--first, they are run locally on a developer's machine, then deployed on test servers, and finally deployed on production servers. We would have the application interacting with components located in different places for each environment. The best approach for this is to maintain different profiles for each environment. One way to do this is by creating different versions of the `application.properties` file, that is, different versions of the file that stores the application-level properties. These property files in Spring Boot can also be YML files, such as `application.yml`. Even if you create different versions, you need a mechanism to tell your applications to pick the relevant version of the file, based on the environment it has been deployed to.

Spring Boot provides amazing support for such a feature. It allows you to have multiple configuration files, each representing a specific profile, and then, you can launch your application in different profiles, depending on the environment it is being deployed to. Let's see this in action, and then we will explain how it works.

Getting ready

For this recipe, there are two options to host another instance of your MySQL database:

1. Use a cloud provider such as AWS and use its Amazon **Relational Database Service** (**RDS**) (`https://aws.amazon.com/rds/`). They have a certain free usage limit.
2. Use a cloud provider such as DigitalOcean (`https://www.digitalocean.com/`) to purchase a droplet (that is, a server) for as little as $5 per month. Then install the MySQL server on it.
3. Use VirtualBox to install Linux on your machine, assuming we are using Windows, or vice versa if you are using Linux. Then install the MySQL server on it.

The options are much more right from hosted database services to servers, which give you complete root access to install the MySQL server. For this recipe, we did the following:

1. We purchased a basic droplet from DigitalOcean.
2. We installed MySQL using `sudo apt-get install mysql-server-5.7` with a password for the root user.
3. We created another user, `springboot`, so that we can use this user to connect from our RESTful web service application:

```
$ mysql -uroot -p
Enter password:
mysql> create user 'springboot'@'%' identified by 'springboot';
```

4. We modified the MySQL configuration file so that the MySQL allows remote connections. This can be done by editing the `bind-address` property in the `/etc/mysql/mysql.conf.d/mysqld.cnf` file to the IP of the server.
5. From MySQL workbench, we added the new MySQL connection by using `IP = <Digital Ocean droplet IP>`, `username = springboot`, and `password =`

`springboot`.



The location for the MySQL configuration file in Ubuntu OS is `/etc/mysql/mysql.conf.d/mysqld.cnf`. One way to find out the location of a configuration file specific to your OS is to do the following:

1. *Run `mysql --help`*
2. *In the output, search for `Default options are read from the following files in the given order:` and what follows is the possible locations for the MySQL configuration file.*

We will create the required table and populate some data. But before that, we will create the `sample` database as `root` and grant all privileges on it to the `springboot` user.

```
mysql -uroot
Enter password:

mysql> create database sample;

mysql> GRANT ALL ON sample.* TO 'springboot'@'%';
Query OK, 0 rows affected (0.00 sec)

mysql> flush privileges;
```

Now, let's connect to the database as the `springboot` user, create the required table, and populate it with some sample data:

```
mysql -uspringboot -pspringboot

mysql> use sample
Database changed
mysql> create table person(
-> id int not null auto_increment,
-> first_name varchar(255),
-> last_name varchar(255),
-> place varchar(255),
-> primary key(id)
-> );
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO person(first_name, last_name, place) VALUES('Mohamed',
'Sanaulla', 'Bangalore');
mysql> INSERT INTO person(first_name, last_name, place) VALUES('Nick',
'Samoylov', 'USA');

mysql> SELECT * FROM person;
```

```
+----+------------+-----------+-----------+
| id | first_name | last_name | place     |
+----+------------+-----------+-----------+
| 1  | Mohamed    | Sanaulla  | Bangalore |
| 2  | Nick       | Samoylov  | USA       |
+----+------------+-----------+-----------+
2 rows in set (0.00 sec)
```

Now we have our cloud instance of the MySQL DB ready. Let's look at how to manage the information of two different connections based on the profile the application is running in.

The initial sample app required for this recipe can be found at the location, `chp10/4_boot_multi_profile_incomplete`. We will convert this app to make it run on different environments.

How to do it...

1. In the `src/main/resources/application.properties` file, add a new `springboot` property, `spring.profiles.active = local`.
2. Create a new file, `application-local.properties`, at the location, `src/main/resources/`.
3. Add the following properties to `application-local.properties` and remove them from the `application.properties` file:

```
        spring.datasource.url=jdbc:mysql://localhost/sample?useSSL=false
        spring.datasource.username=root
        spring.datasource.password=mohamed
```

4. Create another file, `application-cloud.properties`, at `src/main/resources/`.
5. Add the following properties to `application-cloud.properties`:

```
        spring.datasource.url=jdbc:mysql://<digital_ocean_ip>/sample?
useSSL=false
        spring.datasource.username=springboot
        spring.datasource.password=springboot
```

The complete code for the complete application can be found at `chp10/4_boot_multi_profile_incomplete`**.** You can run the application by using the `mvn spring-boot:run` command. Spring Boot reads the `spring.profiles.active` property from the `application.properties` file and runs the application in a local profile. Open the URL, `http://localhost:8080/api/persons`, in the browser to find the following data:

```
[
  {
    "id": 1,
    "firstName": "David ",
    "lastName": "John",
    "place": "Delhi"
  },
  {
    "id": 2,
    "firstName": "Raj",
    "lastName": "Singh",
    "place": "Bangalore"
  }
]
```

Now, run the application on the cloud profile by using the `mvn spring-boot:run -Dspring.profiles.active=cloud` command. Then open `http://localhost:8080/api/persons` in the browser to find the following data:

```
[
  {
    "id": 1,
    "firstName": "Mohamed",
    "lastName": "Sanaulla",
    "place": "Bangalore"
  },
  {
    "id": 2,
    "firstName": "Nick",
    "lastName": "Samoylov",
    "place": "USA"
  }
]
```

You can see that there is a different set of data returned by the same API and the preceding data was inserted in our MySQL database running on the cloud. So, we have been able to successfully run the app in two different profiles: local and cloud.

How it works...

There are multiple ways Spring Boot can read the configuration for the application. Some significant ones are listed here in the order of their relevance (the property defined in the earlier source overrides the property defined in the later sources):

- From the command line. The properties are specified using the `-D` option, like we did while launching the app in the cloud profile, `mvn spring-boot:run -Dspring.profiles.active=cloud`. Or, if you are using JAR, then it would be `java -Dspring.profiles.active=cloud -jar myappjar.jar`.
- From the Java system properties, using `System.getProperties()`.
- OS Environment variables.
- Profile-specific application properties, `application-{profile}.properties`, or the `application-{profile}.yml` files, outside of the packaged JAR.
- Profile-specific application properties `application-{profile}.properties` or `application-{profile}.yml` files, packaged within the JAR.
- Application properties, `application.properties`, or `application.yml` defined outside of the packaged JAR.
- Application properties, `application.properties`, or `application.yml` packaged within the JAR.
- Configuration classes (that is, annotated with `@Configuration`) serving as property sources (annotated with `@PropertySource`).
- Spring Boot's default properties.

In our recipe, we specified all the generic properties such as the following in the `application.properties` file, and any profile-specific property were specified in the profile-specific application properties file:

```
spring.profiles.active=local
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

mybatis.mapper-locations=classpath*:mappers/*.xml
mybatis.configuration.map-underscore-to-camel-case=true
```

From the preceding list, we can find that the `application.properties` or `application-{profile}.properties` file can be defined outside the application JAR. There are default locations where Spring Boot will search for the properties file, and one such path is the `config` sub directory of the current directory the app is running from.

The complete list of Spring Boot-supported application properties can be found at http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html. In addition to these, we can create our own properties required for our application.

The complete code for this can be found at the location, `chp10/4_boot_multi_profile_complete`.

There's more...

We can create a configuration server using Spring Boot, which will act as a repository for all the properties for all the apps in all the profiles. The client apps can then connect with the configuration server to read the relevant properties based on the app name and the app profile.

In the configuration server, the application properties can be read from the filesystem using the classpath or a GitHub repository. The advantage of using a GitHub repository is that the property files can be versioned. The property files in the configuration server can be updated, and these updates can be pushed to the client apps by setting up a message queue to relay the changes downstream. Also, another way is to use the `@RefreshScope` beans and then invoke the `/refresh` API whenever we need the client apps to pull the configuration changes.

Deploying RESTful web services to Heroku

Platform as a Service (**Paas**) is one of the cloud computing models (the other two being **Software as a Service** (**SaaS**) and **Infrastructure as a Service** (**IaaS**)) where the cloud computing provider provides managed computing platforms, which includes OS, programming language runtime, database, and other add ons such as queues, log management, and alerting. They also provide you tools to ease the deployment and dashboards to monitor your applications.

Heroku is one of the earliest players in the field of PaaS providers. It supports the following programming languages: Ruby, Node.js, Java, Python, Clojure, Scala, Go, and PHP. Heroku supports multiple data stores, such as MySQL, MongoDB, Redis, and Elastic search. It provides integration with logging tools, network utils, email services, and monitoring tools.

Heroku provides a command-line tool called heroku-cli (`cli.heroku.com`), which can be used to create Heroku applications, deploy, monitor, add resources, and more. The functionality provided from their web dashboard is supported by the CLI as well. It uses Git to store the application's source code. So, when you push the application code to Heroku's Git repository, it triggers a build, based on the build pack you are using. Then, it either uses the default way to spawn the application or `ProcFile` to execute your application.

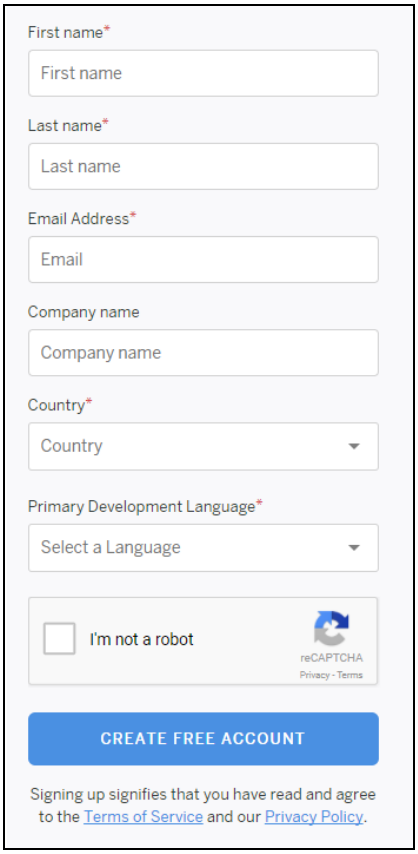
In this recipe, we will deploy our Spring Boot-based RESTful web service to Heroku. We will continue to use the database we created on another cloud provider in the previous recipe, *Creating multiple profiles for Spring Boot*.

Getting ready

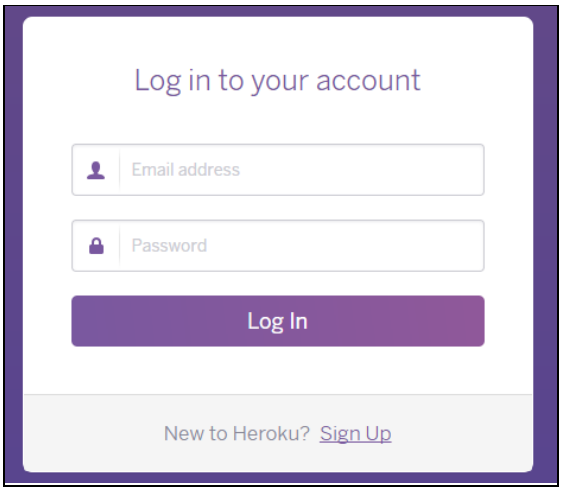
Before we proceed with deploying our sample application on Heroku, we need to sign up for a Heroku account and install its tools, which will enable us to work from the command line. In the subsequent sections, we will guide you through the sign up process, creating a sample app via the web UI, and via the Heroku **command-line interface** (**CLI**).

Setting up a Heroku account

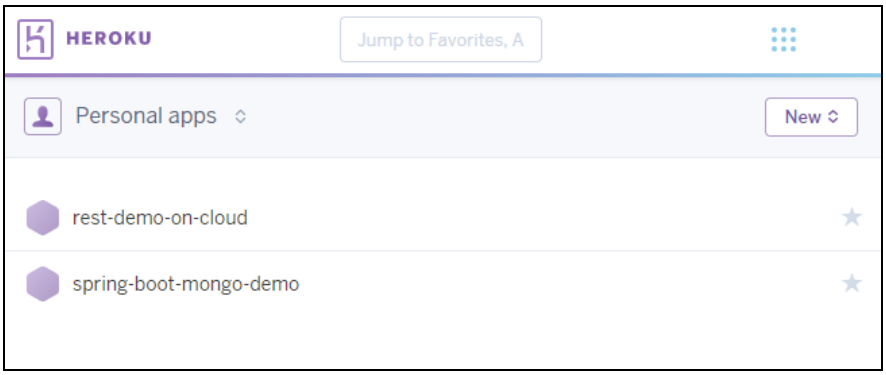
Visit http://www.heroku.com and sign up if you don't have an account. If you have an account, then you can log in. For signing up, the URL is https://signup.heroku.com:



For login, the URL is https://id.heroku.com/login:

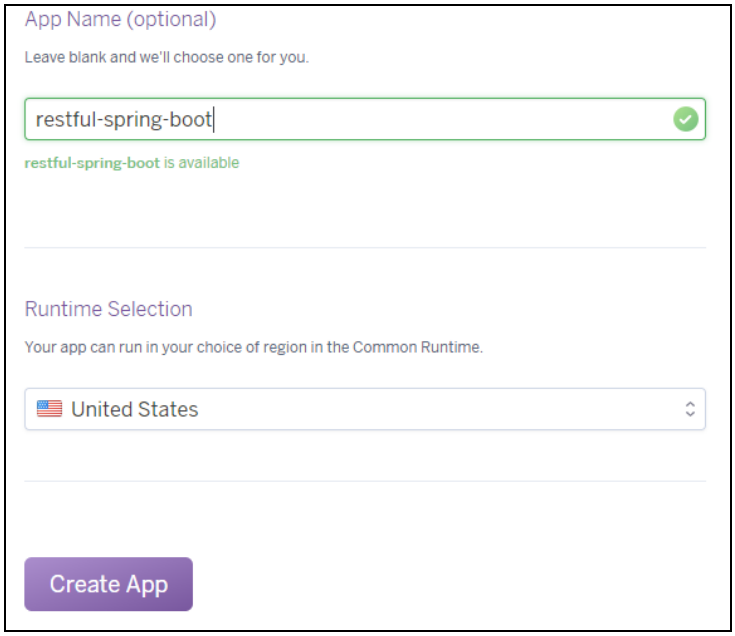


Once you log in successfully, you will see a dashboard with the list of apps, if you have any:



Creating a new app from the UI

Click on New | Create new app, and fill in the details, as shown in the following screenshot, and click on Create App:



Creating a new app from the CLI

Perform the following steps to create a new app from the CLI:

1. Install the Heroku CLI from `https://cli.heroku.com`.
2. Once installed, Heroku should be in your system's `PATH` variable.
3. Open a command prompt and run `heroku create`. You will see output similar to the following:

```
Creating app... done, glacial-beyond-27911
https://glacial-beyond-27911.herokuapp.com/ |
https://git.heroku.com/glacial-beyond-27911.git
```

4. The app name is generated dynamically and a remote Git repository is created. You can specify the app name and region (as done via the UI) by running the command:

```
$ heroku create test-app-9812 --region us
Creating test-app-9812... done, region is us
https://test-app-9812.herokuapp.com/ |
https://git.heroku.com/test-app-9812.git
```

The deployment to Heroku is done via `git push` to the remote Git repository created on Heroku. We will see this in the next section.

We have the source code for the app at `chp10/5_boot_on_heroku`. So, copy this application and go ahead and deploy on Heroku.



You have to log in to the Heroku account before running any of the commands in Heroku's cli. You can log in by running the command, `heroku login`.

How to do it...

1. Run the following command to create a Heroku application:

```
$ heroku create <app_name> -region us
```

2. Initialize the Git repository in the project folder:

```
$ git init
```

3. Add the Heroku Git repository as a remote to your local Git repository:

```
$ heroku git:remote -a <app_name_you_chose>
```

4. Push the source code, that is, the master branch, to the Heroku Git repository:

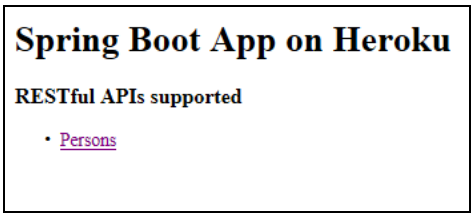
```
$ git add .
$ git commit -m "deploying to heroku"
$ git push heroku master
```

5. When the code is pushed to the Heroku Git repository, it triggers a build. As we are using Maven, it runs the following command:

```
./mvnw -DskipTests clean dependency:list install
```

6. Once the code has completed the build and deployed, you can open the application by using the `heroku open` command. This will open the application in a browser.
7. You can monitor the logs of the application using the `heroku logs --tail` command.

Once the app has been successfully deployed, and after you run the `heroku open` command, you should see the URL being loaded by the browser:



Clicking on the Persons link will display the following information:

```
[
  {
    "id":1,
    "firstName":"Mohamed",
    "lastName":"Sanaulla",
    "place":"Bangalore"
  },
  {
    "id":2,
    "firstName":"Nick",
    "lastName":"Samoylov",
    "place":"USA"
  }
]
```

The interesting thing here is that we have our app running on Heroku, which is connecting to a MySQL database on a DigitalOcean server. We can even provision a database along with the Heroku app and connect to that database. Check out how to do this in the *There's more...* section.

There's more...

1. Add a new DB add-on to the application:

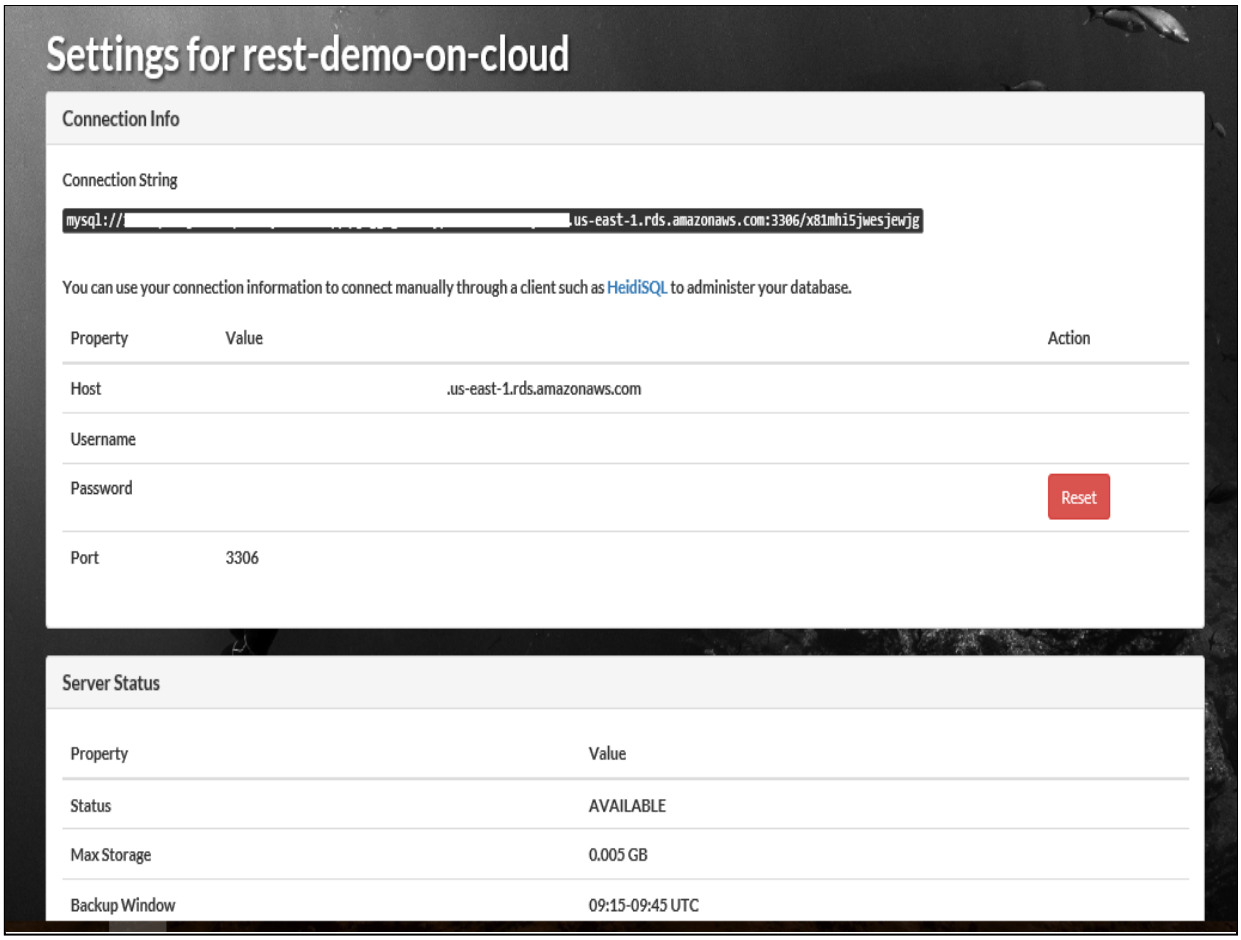
```
$ heroku addons:create jawsdb:kitefin
```

   Here, `addons:create` takes the add-on name and the service plan name, both separated by a colon (`:`). You can know more about the add-on details and plans at https://elements.heroku.com/addons/jawsdb-maria. Also, the Heroku CLI command to add the add-on to your application is given towards the end of the add-on details page for all add-ons.

2. Open the DB dashboard to view the connection details, such as URL, username, password, and the database name:

```
$ heroku addons:open jawsdb
```

   The `jawsdb` dashboard looks something similar to as shown below:



3. You can even get the MySQL connection string from the `JAWSDB_URL` configuration property. You can list the configuration for your app by using the following command:

```
$ heroku config
=== rest-demo-on-cloud Config Vars
JAWSDB_URL: <URL>
```

4. Copy the connection details and create a new connection in MySQL Workbench and connect to this connection. The database name is also created by the add-on. Run the following SQL statements after connecting to the database:

```
use x81mhi5jwesjewjg;
create table person(
  id int not null auto_increment,
  first_name varchar(255),
  last_name varchar(255),
  place varchar(255),
  primary key(id)
```

```
);

INSERT INTO person(first_name, last_name, place)
VALUES('Heroku First', 'Heroku Last', 'USA');

INSERT INTO person(first_name, last_name, place)
VALUES('Jaws First', 'Jaws Last', 'UK');
```

5. Create a new properties file for the Heroku profile, `application-heroku.properties` at `src/main/resources`, with the following properties:

```
spring.datasource.url=jdbc:mysql://
<URL DB>:3306/x81mhi5jwesjewjg?useSSL=false
spring.datasource.username=zzu08pc38j33h89q
spring.datasource.password=<DB password>
```

   You can find the connection related details from the add-on dashboard.

6. Update the `src/main/resources/application.properties` file to replace the value of the `spring.profiles.active` property to `heroku`
7. Commit and push the changes to Heroku remote:

```
$ git commit -am"using heroky mysql addon"
$ git push heroku master
```

8. Once the deployment succeeds, run the `heroku open` command. Then, once the page loads in the browser, click on the Persons link. This time, you will see a different set of data, the one which we entered in our Heroku add-on:

```
[
  {
    "id":1,
    "firstName":"Heroku First",
    "lastName":"Heroku Last",
    "place":"USA"
  },
  {
    "id":2,
    "firstName":"Jaws First",
    "lastName":"Jaws Last",
    "place":"UK"
  }
]
```

With this, we have integrated with a database created on Heroku.

Containerizing the RESTful web service using Docker

We have advanced a lot from the time where an app would be installed across servers to each server being virtualized and the app then being installed on these smaller virtual machines. Scalability issues for the applications were resolved by adding more virtual machines, with the app running to the load balancer.

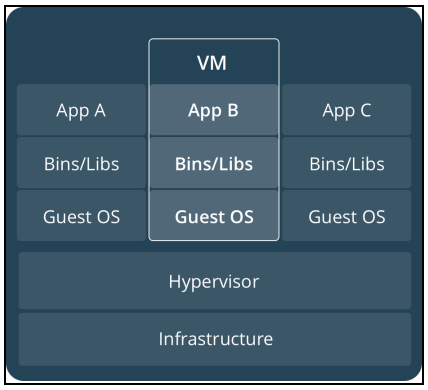
In virtualization, a large server is divided into multiple virtual machines by allocating the computing power, memory, and storage among the multiple virtual machines. This way, each of the virtual machines is in itself capable of all those things that a server was, albeit on a smaller scale. This way, virtualization has helped us a lot in judiciously making use of the server's computing, memory, and storage resources.

However, virtualization needs some setup, that is, you need to create the virtual machine, install the required dependencies, and then run the app. Moreover, you may not be 100% sure if the app would run successfully. The reason for failure may be due to the incompatible OS versions or even due to some configuration missed while setting up or some missing dependency. This setup also leads to some difficulty in horizontal scaling because there is some time spent in the provisioning of the virtual machine and then deploying the app.

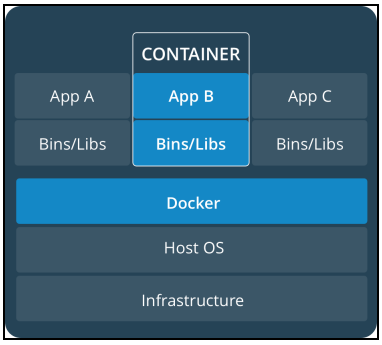
Using tools such as Puppet and Chef does help in provisioning, but then the setting up of the app can often result in issues that might be due to a missing or wrong configuration. This led to the introduction of another concept called containerization.

In the world of virtualization, we have the host OS and then a virtualization software, that is, the hypervisor. We then end up creating multiple machines, where each machine has its own OS on which apps are deployed.

However, in containerization, we don't divide the resources of the server. Instead, we have the server with its host OS, and above that, we have a containerization layer which is a software abstraction layer. We package apps as containers, where a container is packaged with just enough OS functions required to run the app, the software dependencies for the app, and then the app itself. The following image taken from https://docs.docker.com/get-started/#container-diagram best depicts this:



The preceding image illustrates a typical architecture of virtualization systems. The following image illustrates a typical architecture of containerization systems:



The biggest advantage of containerization is that you bundle all the dependencies of the app into a container image. This image is then run on the containerization platform, leading to the creation of a container. We can have multiple containers running simultaneously on the server. If there is a need to add more instances, we can just deploy the image, and this deployment can be automated to support high scalability in an easy way.

Docker is the world's most popular software containerization platform. In this recipe, we will package our sample app found at the location into a Docker image and run the Docker image to launch our application.

Getting ready

For this recipe, we will use a Linux server running Ubuntu 16.04.2 x64.

1. Download the latest `.deb` file from https://download.docker.com/linux/ubuntu/dists/xenial/pool/stable/amd64/. For other Linux distros, you can find the packages at https://download.docker.com/linux/:

```
$ wget https://download.docker.com/linux/ubuntu/dists/xenial
/pool/stable/amd64/docker-ce_17.03.2~ce-0~ubuntu-xenial_amd64.deb
```

2. Install the Docker package using the `dpkg` package manager:

```
$  sudo dpkg -i docker-ce_17.03.2~ce-0~ubuntu-xenial_amd64.deb
```

   The name of the package will vary based on the version you have downloaded.

3. After successful installation, the Docker service starts running. You can verify this by using the `service` command:

```
$ service docker status
  docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled;
  vendor preset: enabled)
Active: active (running) since Fri 2017-07-28 13:46:50 UTC;
        2min 3s ago
Docs: https://docs.docker.com
Main PID: 22427 (dockerd)
```

The application to be dockerized is available at the location, `chp10/6_boot_with_docker`, of the source code downloaded for this book.

How to do it...

1. Create `Dockerfile` at the root of the application with the following content:

```
FROM ubuntu:17.10
FROM openjdk:9-b177-jdk
VOLUME /tmp
ADD target/boot_docker-1.0.jar restapp.jar
ENV JAVA_OPTS="-Dspring.profiles.active=cloud"
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -jar /restapp.jar" ]
```

2. Run the following command to build a Docker image using `Dockerfile` created in the preceding step:

```
$ docker build --tag restapp-image .

Sending build context to Docker daemon 18.45 MB
Step 1/6 : FROM ubuntu:17.10
---> c8cdcb3740f8
Step 2/6 : FROM openjdk:9-b177-jdk
---> 38d822ff5025
Step 3/6 : VOLUME /tmp
---> Using cache
---> 38367613d375
Step 4/6 : ADD target/boot_docker-1.0.jar restapp.jar
---> Using cache
---> 54ad359f53f7
Step 5/6 : ENV JAVA_OPTS "-Dspring.profiles.active=cloud"
---> Using cache
---> dfa324259fb1
Step 6/6 : ENTRYPOINT sh -c java $JAVA_OPTS -jar /restapp.jar
---> Using cache
---> 6af62bd40afe
Successfully built 6af62bd40afe
```

3. You can view the images installed by using the command:

```
$ docker images


REPOSITORY     TAG         IMAGE ID     CREATED     SIZE
restapp-image  latest      6af62bd40afe 4 hours ago 606 MB
openjdk        9-b177-jdk 38d822ff5025 6 days ago  588 MB
ubuntu         17.10       c8cdcb3740f8 8 days ago  93.9 MB
```

You will see that there are OpenJDK and Ubuntu images as well. These were downloaded to build the image for our app, which is

listed first.

4. Now, we need to run the image to create a container that contains our running application:

```
docker run -p 8090:8080 -d --name restapp restapp-image
d521b9927cec105d8b69995ef6d917121931c1d1f0b1f4398594bd1f1fcbee55
```

   The large string printed after the `run` command is the identifier of the container. You can use the initial few characters to uniquely identify the container. Alternatively, you can use the container name, `restapp`.

5. The app will have already started. You can view the logs by running the following command:

```
docker logs restapp
```

6. You can view the Docker containers created by using the following command:

```
docker ps
```

   The output for the above command looks similar to as shown below:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
d521b9927cec        restapp-image       "sh -c 'java $JAVA..."   2 minutes ago       Up 2 minutes        0.0.0.0:8090->8080/tcp   restapp
```

7. You can manage the container by using the following command:

```
$ docker stop restapp
$ docker start restapp
```

Once the app is running, open `http://<hostname>:8090/api/persons`.

How it works...

You define the container structure and its contents by defining `Dockerfile`. `Dockerfile` follows a structure, where each line is of the form, `INSTRUCTION arguments`. There is a predefined set of instructions, namely `FROM`, `RUN`, `CMD`, `LABEL`, `ENV`, `ADD`, `COPY`, and others. A complete list can be found at https://docs.docker.com/engine/reference/builder/#from. Let's look at our defined `Dockerfile`:

```
FROM ubuntu:17.10
FROM openjdk:9-b177-jdk
VOLUME /tmp
ADD target/boot_docker-1.0.jar restapp.jar
ENV JAVA_OPTS="-Dspring.profiles.active=cloud"
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -jar /restapp.jar" ]
```

The first two lines using the `FROM` instruction specified the base image for our Docker image. We use the Ubuntu OS image as the base image and then combine it with the OpenJDK 9 image. The `VOLUME` instruction is used to specify the mount point for the image. This is usually a path in the host OS.

The `ADD` instruction is used to copy the file from the source to the destination directory under the working directory. The `ENV` instruction is used for defining the environment variables.

The `ENTRYPOINT` instruction is used to configure the container to run as an executable. For this instruction, we pass an array of arguments, which we would otherwise have executed directly from the command line. In our scenario, we are using the bash shell to run `java -$JAVA_OPTS -jar <jar name>`.

Once we have defined `Dockerfile`, we instruct the Docker tool to build an image using `Dockerfile`. We also provide a name for the image using the `--tag` option. When building our app image, it will download the required based images, which, in our case, are Ubuntu and OpenJDK images. So, if you list the Docker images, then you will see the base images along with our app image.

This Docker image is a reusable entity. If we need more instances of the app, we spawn a new container using the `docker run` command. When we run the Docker image, we have multiple options, where one of them is a `-p` option, which maps the ports from within the container to the host OS. In our case, we map the `8080` port of our Spring Boot app to `8090` of the host OS.

Now, to check the status of our running app, we can check the logs using `docker logs restapp`. Apart from this, the `docker` tool supports multiple commands. It's highly recommended to run `docker help` and explore the commands supported.

Docker, the company behind Docker has created a set of base images, which can be used to create containers. For example, there are images for MySQL DB, Couchbase, Ubuntu, and other operating systems. You can explore the packages at https://store.docker.com/.

Networking

In this chapter, we will cover the following recipes:

- Making an HTTP GET request
- Making an HTTP POST request
- Making an HTTP request for a protected resource
- Making an asynchronous HTTP request
- Making an HTTP request using Apache HttpClient
- Making an HTTP request using the Unirest HTTP client library

Introduction

Java's support for interacting with HTTP-specific features has been very primitive. The `HttpURLConnection` class, available since JDK 1.1, provides APIs for interacting with URLs with HTTP-specific features. Since this API has been there even before HTTP/1.1, it lacked advanced features and was a pain to use. This is why developers mostly resorted to using third-party libraries, such as **Apache HttpClient**, Spring frameworks, HTTP APIs, and so on.

In JDK 9, a new HTTP client API is being introduced under JEP 110 (http://openjdk.java.net/jeps/110). Unfortunately, this API is being introduced as an incubator module (http://openjdk.java.net/jeps/11). An incubator module contains non-final APIs, which are significantly larger and not mature completely to be included in Java SE. This is a form of beta release of the API so that the developers get to use the APIs much earlier. But the catch here is that there is no backward compatibility support for these APIs in the newer versions of JDK. This means that code that is dependent on the incubator modules might break with the newer versions of JDK. This might be due to the incubator module being promoted to Java SE or being silently dropped from the incubator modules.

In any case, it will be good to know about the HTTP client APIs, which might come into future JDK releases. In addition to this, it is good to know about the alternatives we have for now. So, in this chapter, we will cover a few recipes showing how to use the HTTP client APIs in the JDK 9 incubator modules, and then a few other APIs, which make use of the Apache HttpClient (http://hc.apache.org/httpcomponents-client-ga/) API and the Unirest Java HTTP library (http://unirest.io/java.html).

Making an HTTP GET request

In this recipe, we will look at using the JDK 9 HTTP Client API to make a `GET` request to the URL, `http://httpbin.org/get`.

How to do it...

1. Create an instance of `jdk.incubator.http.HttpClient` using its builder, `jdk.incubator.http.HttpClient.Builder`:

```
HttpClient client = HttpClient.newBuilder().build();
```

2. Create an instance of `jdk.incubator.http.HttpRequest` using its builder, `jdk.incubator.http.HttpRequest.Builder`. The requested URL should be provided as an instance of `java.net.URI`:

```
HttpRequest request = HttpRequest
            .newBuilder(new URI("http://httpbin.org/get"))
            .GET()
            .version(HttpClient.Version.HTTP_1_1)
            .build();
```

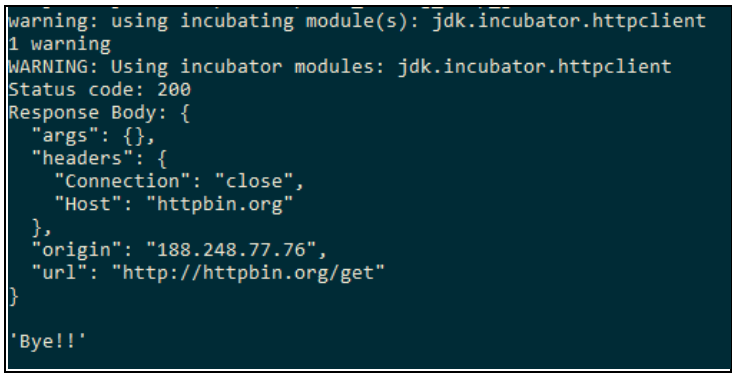
3. Send the HTTP request using the `send` API of `jdk.incubator.http.HttpClient`. This API takes an instance of `jdk.incubator.http.HttpRequest` and an implementation of `jdk.incubator.http.HttpResponse.BodyHandler`:

```
HttpResponse<String> response = client.send(request,
                      HttpResponse.BodyHandler.asString());
```

4. Print the `jdk.incubator.http.HttpResponse` status code and the response body:

```
System.out.println("Status code: " + response.statusCode());
System.out.println("Response Body: " + response.body());
```

The complete code for this can be found at the location `chp11/1_making_http_get`. You can make use of the run scripts, `run.bat` or `run.sh`, to compile and run the code:


warning: using incubating module(s): jdk.incubator.httpclient
1 warning
WARNING: Using incubator modules: jdk.incubator.httpclient
Status code: 200
Response Body: {
"args": {},
"headers": {
"Connection": "close",
"Host": "httpbin.org"
},
"origin": "188.248.77.76",
"url": "http://httpbin.org/get"
}
'Bye!!'

How it works...

There are two main steps in making an HTTP call to a URL:

1. Creating an HTTP client to initiate the call.
2. Setting up the destination URL, required HTTP headers, and the HTTP method type, that is, `GET`, `POST`, or `PUT`.

The Java HTTP Client API provides a builder class, `jdk.incubator.http.HttpClient.Builder`, which can be used to build an instance of `jdk.incubator.http.HttpClient` at the same time, making use of the builder APIs to set up `jdk.incubator.http.HttpClient`. The following code snippet shows how to get an instance of `jdk.incubator.http.HttpClient` with the default configuration:

```
HttpClient client = HttpClient.newHttpClient();
```

The following code snippet uses the builder to configure and then create an instance of `jdk.incubator.http.HttpClient`:

```
HttpClient client = HttpClient
                    .newBuilder()
                    //redirect policy for the client. Default is NEVER
                    .followRedirects(HttpClient.Redirect.ALWAYS)
                    //HTTP client version. Defabult is HTTP_2
                    .version(HttpClient.Version.HTTP_1_1)
                    //few more APIs for more configuration
                    .build();
```

There are more APIs in the builder, such as setting authentication, proxy, and providing SSL context, which we will look at in different recipes.

Setting up the destination URL is nothing but creating an instance of `jdk.incbator.http.HttpRequest` using its builder and its APIs to configure the same. The following code snippet shows how to create an instance of `jdk.incbator.http.HttpRequest`:

```
HttpRequest request = HttpRequest
                .newBuilder()
```

```
                .uri(new URI("http://httpbin.org/get")
                .headers("Header 1", "Value 1", "Header 2", "Value 2")
                .timeout(Duration.ofMinutes(5))
                .version(HttpClient.Version.HTTP_1_1)
                .GET()
                .build();
```

The `jdk.incubator.http.HttpClient` object provides two APIs to make an HTTP call:

- Send synchronously using the `HttpClient#send()` method
- Send asynchronously using the `HttpClient#sendAsync()` method

The `send()` method takes in two parameters: the HTTP request and the handler for the HTTP response. The handler for the response is represented by the implementation of the `jdk.incubator.http.HttpResponse.BodyHandler` interface. There are a few implementations available, such as `asString()`, which reads the response body as `String`, `asByteArray()`, which reads the response body as a byte array, and so on. We will use the `asString()` method, which returns the response `Body` as a string:

```
HttpResponse<String> response = client.send(request,
                                HttpResponse.BodyHandler.asString());
```

The instance of `jdk.incubator.http.HttpResponse` represents the response from the HTTP server. It provides APIs for the following:

- Getting the response body (`body()`)
- HTTP headers (`headers()`)
- The initial HTTP request (`request()`)
- The response status code (`statusCode()`)
- The URL used for the request (`uri()`)

The `HttpResponse.BodyHandler` implementation passed to the `send()` method helps in converting the HTTP response into a compatible format, such as `String`, a `byte` array, and so on.

Making an HTTP POST request

In this recipe, we will look at posting some data to an HTTP service via the request body. We will post the data to a URL: `http://httpbin.org/post`.



We will skip the package prefix for the classes, as it is assumed to be `jdk.incubator.http`.

How to do it...

1. Create an instance of `HttpClient` using its `HttpClient.Builder` builder:

```
HttpClient client = HttpClient.newBuilder().build();
```

2. Create the required data to be passed into the request body:

```
Map<String, String> requestBody =
            Map.of("key1", "value1", "key2", "value2");
```

3. Create a `HttpRequest` object with the request method as POST and by providing the request body data as `String`. We make use of Jackson's `ObjectMapper` to convert the request body, `Map<String, String>`, into a plain JSON `String` and then make use of `HttpRequest.BodyProcessor` to process the `String` request body:

```
ObjectMapper mapper = new ObjectMapper();
HttpRequest request = HttpRequest
           .newBuilder(new URI("http://httpbin.org/post"))
           .POST(
  HttpRequest.BodyProcessor.fromString(
    mapper.writeValueAsString(requestBody)
  )
)
.version(HttpClient.Version.HTTP_1_1)
.build();
```

4. The request is sent and the response is obtained by using the `send(HttpRequest, HttpRequest.BodyHandler)` method:

```
HttpResponse<String> response = client.send(request,
                      HttpResponse.BodyHandler.asString());
```

5. We then print the response status code and the response body sent by the server:

```
System.out.println("Status code: " + response.statusCode());
System.out.println("Response Body: " + response.body());
```

The complete code for this can be found at `chp11/2_making_http_post`. Make sure that there are the following Jackson JARs in the

location, `chp11/2_making_http_post/mods`:

- `jackson.databind.jar`
- `jackson.core.jar`
- `jackson.annotations.jar`

Also, take note of the module definition, `module-info.java`, available at the location, `chp11/2_making_http_post/src/http.client.demo`.



To understand how Jackson JARs are used in this modular code, please refer to the recipes Bottom-up migration and Top-down migration in Chapter 3, Modular Programming.

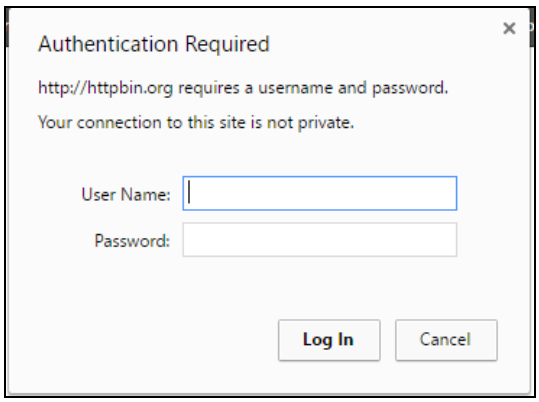
Run scripts, `run.bat` and `run.sh`, are provided to facilitate the compilation and execution of the code:

```
G:\java9\java9-samples\chp11\2_making_http_post>run.bat
warning: using incubating module(s): jdk.incubator.httpclient
1 warning
WARNING: Using incubator modules: jdk.incubator.httpclient
Status code: 200
Response Body: {
  "args": {},
  "data": "{\"key2\":\"value2\",\"key1\":\"value1\"}",
  "files": {},
  "form": {},
  "headers": {
    "Connection": "close",
    "Content-Length": "33",
    "Host": "httpbin.org"
  },
  "json": {
    "key1": "value1",
    "key2": "value2"
  },
  "origin": "188.248.77.76",
  "url": "http://httpbin.org/post"
}

'Bye!!'
```

Making an HTTP request for a protected resource

In this recipe, we will look at invoking an HTTP resource that has been protected by user credentials. The URL, `http://httpbin.org/basic-auth/user/passwd`, has been protected by HTTP basic authentication. Basic authentication requires a username and a password to be provided in plain text, which is then used by the HTTP resources to decide whether the user authentication is successful or not.

If you open the link, `http://httpbin.org/basic-auth/user/passwd`, in the browser, it will prompt for the username and password, as shown in the following screenshot:



Use the username as `user` and password as `passwd`, and you will be authenticated to be shown a JSON response, as follows:

```
{
  "authenticated": true,
  "user": "user"
}
```

Let's achieve the same using the `HttpClient` API.

How to do it...

1. We need to extend `java.net.Authenticator` and override its `getPasswordAuthentication()` method. This method should return an instance of `java.net.PasswordAuthentication`. Let's create a class, `UsernamePasswordAuthenticator`, which extends `java.net.Authenticator`:

```
public class UsernamePasswordAuthenticator
  extends Authenticator{
}
```

2. We will create two instance variables in the `UsernamePasswordAuthenticator` class to store the username and password, and we'll provide a constructor to initialize the same:

```
private String username;
private String password;

public UsernamePasswordAuthenticator(){}
public UsernamePasswordAuthenticator ( String username,
                                       String password){
  this.username = username;
  this.password = password;
}
```

3. We then override the `getPasswordAuthentication()` method to return an instance of `java.net.PasswordAuthentication`, initialized with the username and password:

```
@Override
protected PasswordAuthentication getPasswordAuthentication(){
  return new PasswordAuthentication(username,
                                    password.toCharArray());
}
```

4. We then create an instance of `UsernamePasswordAuthenticator`:

```
String username = "user";
String password = "passwd";
UsernamePasswordAuthenticator authenticator =
        new UsernamePasswordAuthenticator(username, password);
```

5. We provide the instance of `UsernamePasswordAuthenticator` while initializing the `HttpClient`:

```
HttpClient client = HttpClient.newBuilder()
                              .authenticator(authenticator)
                              .build();
```

6. A corresponding `HttpRequest` object is created to call the protected HTTP resource, http://httpbin.org/basic-auth/user/passwd:

```
HttpRequest request = HttpRequest.newBuilder(new URI(
  "http://httpbin.org/basic-auth/user/passwd"
))
.GET()
.version(HttpClient.Version.HTTP_1_1)
.build();
```

7. We obtain `HttpResponse` by executing the request and print the status code and the request body:

```
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandler.asString());

System.out.println("Status code: " + response.statusCode());
System.out.println("Response Body: " + response.body());
```

The complete code for this is available at the location, `chp11/3_making_http_request_protected_res`. You can run the code by using the run scripts, `run.bat` or `run.sh`:

```
G:\java9\java9-samples\chp11\3_making_http_request_protected_res>run.bat
warning: using incubating module(s): jdk.incubator.httpclient
1 warning
WARNING: Using incubator modules: jdk.incubator.httpclient
Status code: 200
Response Body: {
  "authenticated": true,
  "user": "user"
}

'Bye!!'
```

How it works...

The `Authenticator` object is used by the network calls to obtain the authentication information. Developers generally extend the `java.net.Authenticator` class and override its `getPasswordAuthentication()` method. The username and password are read either from the user input or from the configuration and are used by the extended class to create an instance of `java.net.PasswordAuthentication`.

In the recipe, we created an extension of `java.net.Authenticator`, as follows:

```
public class UsernamePasswordAuthenticator
  extends Authenticator{
    private String username;
    private String password;

    public UsernamePasswordAuthenticator(){}

    public UsernamePasswordAuthenticator ( String username,
                                           String password){
        this.username = username;
        this.password = password;
    }

    @Override
    protected PasswordAuthentication getPasswordAuthentication(){
      return new PasswordAuthentication(username,
                        password.toCharArray());
    }
}
```

The instance of `UsernamePasswordAuthenticator` is then provided to the `HttpClient.Builder` API. The HttpClient instance makes use of this authenticator to get the username and password while invoking the protected HTTP request.

Making an asynchronous HTTP request

In this recipe, we will look at how to make an asynchronous GET request. In an asynchronous request, we don't wait for the response; instead, we handle the response whenever it is received by the client. In jQuery, we will make an asynchronous request and provide a callback that takes care of processing the response, while in the case of Java, we get an instance of `java.util.concurrent.CompletableFuture`, and then we invoke the `thenApply` method to process the response. Let's see this in action.

How to do it...

1. Create an instance of `HttpClient` using its builder, `HttpClient.Builder`:

```
HttpClient client = HttpClient.newBuilder().build();
```

2. Create an instance of `HttpRequest` using its `HttpRequest.Builder` builder, representing the URL and the corresponding HTTP method to be used:

```
HttpRequest request = HttpRequest
                .newBuilder(new URI("http://httpbin.org/get"))
                .GET()
                .version(HttpClient.Version.HTTP_1_1)
                .build();
```

3. Use the `sendAsync` method to make an asynchronous HTTP request and keep a reference to the `CompletableFuture<HttpResponse<String>>` object thus obtained. We will use this to process the response:

```
CompletableFuture<HttpResponse<String>> responseFuture =
          client.sendAsync(request,
                 HttpResponse.BodyHandler.asString());
```

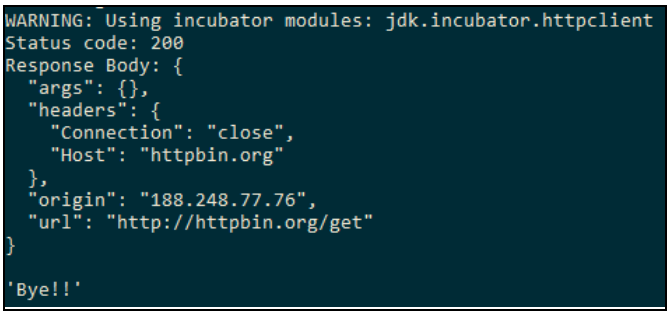
4. We provide `CompletionStage` so as to process the response once the previous stage completes. For this, we make use of the `thenAccept` method, which takes a lambda expression:

```
CompletableFuture<Void> processedFuture =
           responseFuture.thenAccept(response -> {
  System.out.println("Status code: " + response.statusCode());
  System.out.println("Response Body: " + response.body());
});
```

5. Finally, we wait for the future to complete:

```
CompletableFuture.allOf(processedFuture).join();
```

The complete code for this recipe can be found at the location, `chp11/4_async_http_request`. We have provided the `run.bat` and `run.sh` scripts to compile and run the recipe:

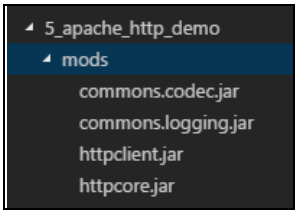

WARNING: Using incubator modules: jdk.incubator.httpclient
Status code: 200
Response Body: {
"args": {},
"headers": {
"Connection": "close",
"Host": "httpbin.org"
},
"origin": "188.248.77.76",
"url": "http://httpbin.org/get"
}
'Bye!!'

Making an HTTP request using Apache HttpClient

In this recipe, we will make use of the Apache HttpClient (https://hc.apache.org/httpcomponents-client-4.5.x/index.html) library to make a simple HTTP GET request. As we are using Java 9, we would want to make use of the module path and not the classpath. Hence, we need to modularize the Apache HttpClient library. One way to achieve this is to use the concept of automatic modules. Let's see how to set up the dependencies for the recipe in the *Getting ready* section.

Getting ready

All the required JARs are already present in the location, `chp11\5_apache_http_demo\mods`:



Once these JARs are on the module path, we can declare a dependency on these JARs in `module-info.java`, which is present at the location, `chp11\5_apache_http_demo\src\http.client.demo`, as shown in the following code snippet:

```
module http.client.demo{
  requires httpclient;
  requires httpcore;
  requires commons.logging;
  requires commons.codec;
}
```

How to do it...

1. Create a default instance of `org.http.client.HttpClient` using its `org.apache.http.impl.client.HttpClients` factory:

```
CloseableHttpClient client = HttpClients.createDefault();
```

2. Create an instance of `org.apache.http.client.methods.HttpGet` along with the required URL. This represents both the HTTP method type and the requested URL:

```
HttpGet request = new HttpGet("http://httpbin.org/get");
```

3. Execute the HTTP request using the `HttpClient` instance to obtain an instance of `CloseableHttpResponse`:

```
CloseableHttpResponse response = client.execute(request);
```

   The `CloseableHttpResponse` instance returned after executing the HTTP request can be used to obtain details such as the response status code and other contents of the response embedded within the instance of an implementation of `HttpEntity`.

4. We make use of `EntityUtils.toString()` to obtain the response body embedded within the instance of an implementation of `HttpEntity` and print both the status code and response body:

```
int statusCode = response.getStatusLine().getStatusCode();
String responseBody = EntityUtils.toString(response.getEntity());
System.out.println("Status code: " + statusCode);
System.out.println("Response Body: " + responseBody);
```

The complete code for this recipe can be found at the location, `chp11\5_apache_http_demo`. We have provided `run.bat` and `run.sh` to compile and execute the recipe code:

```
Status code: 200
Response Body: {
  "args": {},
  "headers": {
    "Accept-Encoding": "gzip,deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "Apache-HttpClient/4.5.3 (Java/9-ea)"
  },
  "origin": "188.248.77.76",
  "url": "http://httpbin.org/get"
}

'Bye!!'
```

There is more...

We can provide a custom response handler while invoking the `HttpClient.execute` method, as follows:

```
String responseBody = client.execute(request, response -> {
  int status = response.getStatusLine().getStatusCode();
  HttpEntity entity = response.getEntity();
  return entity != null ? EntityUtils.toString(entity) : null;
});
```

In this case, the response is processed by the response handler and returns us the response body string. The complete code for this can be found at the location, `chp11\5_1_apache_http_demo_response_handler`.

Making an HTTP request using the Unirest HTTP client library

In this recipe, we will make use of the Unirest HTTP (http://unirest.io/java.html) Java library to access HTTP services. Unirest Java is a library based on Apache's HTTP client library and provides a fluent API for making HTTP requests.

Getting ready

As the Java library is not modular, we will make use of the concept of automatic modules as explained in `Chapter 3`, *Modular Programming*. The JARs belonging to the library are placed on the module path of the application, and the application then declares a dependency on the JARs by using the name of the JAR as its module name. This way, a JAR file automatically becomes a module and is hence called an automatic module.

The Maven dependency for the Java library is:

```
<dependency>
  <groupId>com.mashape.unirest</groupId>
  <artifactId>unirest-java</artifactId>
  <version>1.4.9</version>
</dependency>
```

As we are not using Maven in our samples, we have downloaded the JARs into the folder, `chp11/6_unirest_http_demo/mods`.

The module definition is as follows:

```
module http.client.demo{
  requires httpasyncclient;
  requires httpclient;
  requires httpmime;
  requires json;
  requires unirest.java;
  requires httpcore;
  requires httpcore.nio;
  requires commons.logging;
  requires commons.codec;
}
```

How to do it...

Unirest provides a very fluid API for making HTTP requests. We can make a `GET` request as follows:

```
HttpResponse<JsonNode> jsonResponse =
  Unirest.get("http://httpbin.org/get")
         .asJson();
```

The response status and response body can be obtained from the `jsonResponse` object, as follows:

```
int statusCode = jsonResponse.getStatus();
JsonNode jsonBody = jsonResponse.getBody();
```

We can make a `POST` request and pass some data, as follows:

```
jsonResponse = Unirest.post("http://httpbin.org/post")
                      .field("key1", "val1")
                      .field("key2", "val2")
                      .asJson();
```

We can make a call to a protected HTTP resource, as follows:

```
jsonResponse = Unirest.get("http://httpbin.org/basic-auth/user/passwd")
                      .basicAuth("user", "passwd")
                      .asJson();
```

The code for this can be found at the location, `chp11\6_unirest_http_demo`.

We have provided the `run.bat` and `run.sh` scripts to execute the code.

There's more...

The Unirest Java library provides much more advanced functionality, such as making async requests, file uploads, and using proxy, among other features. It's advisable that you try out these different features of the library.

Memory Management and Debugging

This chapter deals with managing the memory of your Java application. Understanding the garbage collection process is crucial to developing memory-efficient applications. We will introduce you to the garbage collection algorithm being used in Java 9. Then, we will introduce you to some new features of Java 9, which help in advanced application diagnostics. We'll also show you how to manage the resources by using the new *try with resources* construct. Later, we'll show you the new stack walking API introduced in Java 9. The following recipes will be covered:

- Understanding the G1 garbage collector
- Unified logging for JVM
- Using the new diagnostic commands for the JVM
- Try with resources for better resource handling
- Stack walking for improved debugging
- Some best practices for better memory usage

Introduction

Memory management is the process of memory allocation (for program execution) and memory reuse (after some of the allocated memory is not used anymore). In Java, this process happens automatically and is called **Garbage Collection** (**GC**). The effectiveness of GC affects two major application characteristics--responsiveness and throughput.

Responsiveness is measured by how quickly an application responds (brings necessary data) to the request. For example, how quickly a website returns a page or how quickly a desktop application responds to an event. Naturally, the lower the response time, the better the user experience, which is the goal of design and implementation for many applications.

Throughput indicates the amount of work an application can do in a unit of time. For example, how many requests a web application can serve or how many transactions a database can support. The bigger the number, the more value the application can potentially generate and the greater number of users it can accommodate.

Not every application needs to have the minimal possible responsiveness and the maximum achievable throughput. An application may be an asynchronous submit-and-go-do-something-else, which does not require much user interaction. There may be a few potential application users too, so a lower than average throughput could be more than enough. Yet, there are applications that have high requirements to one or both of these characteristics and cannot tolerate long pauses imposed by the GC process.

GC, on the other hand, needs to stop any application execution to reassess the memory usage and to release it from data not used anymore. Such periods of GC activity are called stop-the-world. The longer they are, the quicker the GC does its job and the longer an application freeze lasts, which can eventually grow big enough to affect both the application responsiveness and throughput. If that is the case, the GC tuning and JVM

optimization become important and require an understanding of the GC principles and its modern implementations.

Unfortunately, this step is often missed. Trying to improve responsiveness and/or throughput, companies and individuals just add memory and other computing capacities, thus providing the originally small existing problem with the space to grow. The enlarged infrastructure, in addition to hardware and software costs, requires more people to maintain it and eventually justifies the building of a whole new organization dedicated to keeping up the system. By then, the problem reaches the scale of becoming virtually unsolvable and feeds on those who have created it by forcing them to do the routine--almost menial--work for the rest of their professional lives.

In this chapter, we will focus on **Garbage-First** (**G1**) garbage collector that is going to be the default one in Java 9. However, we'll also refer to a few other available GC implementations to contrast and explain some design decisions that have brought G1 to life. Besides, they might be more appropriate than G1 for some of the systems.

Memory organization and management are very specialized and complex areas of expertise in JVM development. This book is not intended to address the implementation details on such a level. Our focus is on those aspects of GC that can help an application developer to tune it for the application needs by setting the corresponding parameters of the JVM runtime.

There are two memory areas that are used by GC, heap and stack, which illustrate the main principle of any GC implementation. The first one is used by the JVM to allocate memory and store objects created by the program. When an object is created with the `new` keyword, it is allocated on the heap, and the reference to it is stored on the stack. The stack also stores primitive variables and references to heap objects that are used by the current method or thread. The stack operates in **Last-In-First-Out** (**LIFO**). The stack is much smaller than the heap, and only the GC reads it.

Here is a slightly simplistic, but good enough for our purpose, high-level view of the main activity of any GC: walking through objects in the heap and removing those that don’t have any references in the stack.

Understanding the G1 garbage collector

The previous GC implementations include **Serial GC**, **Parallel GC**, and **Concurrent Mark-Sweep** (**CMS**) collector. They divide the heap into three sections: young generation, old or tenured generation, and humongous regions for holding the objects that are 50% the size of a standard region or larger. The young generation contains most of the newly created objects; this is the most dynamic area because a majority of the objects are short-lived and soon (as they age) become eligible for collection. The term age refers to the number of collection cycles the object has survived. The young generation has three collection cycles: an *Eden space* and two survivor spaces, such as survivor 0 (*S0*) and survivor 1 (*S1*). The objects are moved through them (according to their age and some other characteristics) until they are eventually discarded or placed in the old generation.

The old generation contains objects that are older than a certain age. This area is bigger than the young generation, and because of this, the garbage collection here is more expensive and happens not as often as in the young generation.

The permanent generation contains metadata that describes the classes and methods used in applications. It also stores strings, library classes, and methods.

When the JVM starts, the heap is empty and then the objects are pushed into Eden. When it is filling up, a minor GC process starts. It removes the unreferenced and circular referred objects and moves the others to the *S0* area.

First, any new object is allocated to the Eden space. Both the survivor spaces start out empty. When the Eden space fills up, a minor garbage

collection is triggered. Referenced objects are moved to the S0 space. Unreferenced objects are deleted.

The next minor GC process migrates the referenced objects to *S1* and increments the age of those that survived the previous minor collection. After all the surviving objects (of different ages) are moved to *S1*, both *S0* and Eden become empty.

In the next minor collection, *S0* and *S1* switch their roles. The referenced objects are moved from Eden to *S1* and *S1* to *S0*.

In each of the minor collections, the objects that have reached a certain age are moved to the old generation. As we mentioned earlier, the old generation is checked eventually (after several minor collections), the unreferenced objects are removed from there, and the memory is defragmented. This cleaning of the old generation is considered a major collection.

The permanent generation is cleaned at different times by different GC algorithms.

The G1 GC does it somewhat differently. It divides the heap into equal-sized regions and assigns each of them one of the same roles--Eden, survivor, or old--but changes the number of regions with the same role dynamically, depending on the need. It makes the memory cleaning process and the memory defragmentation more predictable.

Getting ready

The serial GC cleans the young and the old generations in the same cycle (serially, thus the name). During the task, it stops the world. That is why it is used for non-server applications with one CPU and a heap size of a few hundred MB.

The parallel GC works in parallel on all available cores, although the number of threads can be configured. It also stops the world and is appropriate only for applications that can tolerate long freezing times.

The CMS collector was designed to address this very issue of long pauses. It does it at the expense of not defragmenting the old generation and doing some analysis in parallel to the application execution (typically using 25% of CPU). The collection of old generation starts when it is 68% full (by default, but this value can be configured).

The G1 GC algorithm is similar to the CMS collector. First, it concurrently identifies all the referenced objects in the heap and marks them correspondingly. Then it collects the emptiest regions first, thus releasing a lot of free space. That's why it is called *Garbage-First*. Because it uses many small dedicated regions, it has a better chance in predicting the amount of time it needs to clean one of them and in fitting a user-defined pause time (G1 may exceed it occasionally, but it is pretty close most of the times).

The main beneficiaries of G1 are applications that require large heaps (6 GB or more) and that do not tolerate long pauses (0.5 sec or less). If an application encounters an issue of too many and/or too long pauses, it can benefit from switching from the CMS or parallel GC (especially the parallel GC of the old generation) to the G1 GC. If that is not the case, switching to the G1 collector is not a requirement for using JDK 9.

The G1 GC starts with the young generation collection using stop-the-world pauses for evacuation (moving objects inside the young generation and out to the old generation). After the occupancy of the old generation reaches a certain threshold, it is collected too. The collection of some of the objects in the old generation is done concurrently and some objects are collected using stop-the-world pauses. The steps include the following:

- The initial marking of the survivor regions (root regions), which may have references to objects in the old generation, done using stop-the-world pauses
- The scanning of survivor regions for references to the old generation, done concurrently while the application continues to run
- The concurrent marking of live objects over the entire heap, done concurrently while the application continues to run
- The remark step completes the marking of live objects, done using stop-the-world pauses
- The cleanup process calculates the age of live objects, frees the regions (using stop-the-world pauses), and returns them to the free list (concurrently)

The preceding sequence might be interspersed with young generation evacuations because most of the objects are short-lived and it is easier to free a lot of memory by scanning the young generation more often.

There is also a mixed phase, when G1 collects the regions already marked as mostly garbage in both the young and old generations, and humongous allocation, when large objects are moved to or evacuated from humongous regions.

There are a few occasions when full GC is performed, using stop-the-world pauses:

- **Concurrent failure**: This happens if the old generation gets full during the marking phase
- **Promotion failure**: This happens if the old generation runs out of space during the mixed phase

- **Evacuation failure**: This happens when the collector cannot promote objects to the survivor space and the old generation
- **Humongous allocation**: This happens when an application tries to allocate a very big object

If tuned properly, your applications should avoid full GC.

To help with GC tuning, the JVM documentation describes *ergonomics*--the process by which "*the JVM and garbage collection tuning, such as behavior-based tuning, improve application performance. The JVM provides platform-dependent default selections for the garbage collector, heap size, and runtime compiler. These selections match the needs of different types of applications while requiring less command-line tuning. In addition, behavior-based tuning dynamically tunes the sizes of the heap to meet a specified behavior of the application*" (from the JVM documentation).

How to do it...

1. To see how GC works, write the following program:

```
package com.packt.cookbook.ch12_memory;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;
public class Chapter12Memory {
  public static void main(String... args) {
    int max = 99888999;
    System.out.println("Chapter12Memory.main() for "
                       + max + " is running...");
    List<AnObject> list = new ArrayList<>();
    IntStream.range(0, max)
             .forEach(i -> list.add(new AnObject(i)));
  }
  private static class AnObject {
    private int prop;
    AnObject(int i){ this.prop = i; }
  }
}
```

   As you can see, it creates 99,888,999 objects and adds them to the `List<AnObject> list` collection. You might tune it by decreasing the maximum number of objects (`max`) to match the configuration of your computer.

2. The G1 GC is the default collector in Java 9, so you don't have to set anything if it is good enough for your application. Nevertheless, you can explicitly enable G1 by providing `-XX:+UseG1GC` on the command line (run in the same folder, where the executable `.jar` file is located, which contains the `com.packt.cookbook.ch12_memory.Chapter12Memory` class with the `main()` method):

```
java -XX:+UseG1GC -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```



Note that we assume you can build an executable `.jar` file and understand the basic Java execution command. If not, please refer to the JVM documentation.

Other available GCs can be used by setting one of the following options:

- `-XX:+UseSerialGC` for using a Serial collector.
- `-XX:+UseParallelGC` for using parallel collector with parallel compaction (which enables the parallel collector to perform major collections in parallel). Without parallel compaction, major collections are performed using a single thread, which can significantly limit the scalability. Parallel compaction is disabled by the `-XX:+UseParallelOldGC` option.
- `-XX:+UseConcMarkSweepGC` for using the CMS collector.

3. To see the log messages of GC, set `-Xlog:gc`. You can also use the Unix utility, `time`, to measure the time it took to do the job (the utility publishes the last three lines of the output, so you do not need to use it if you cannot or do not want to do it):

```
time java -Xlog:gc -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

4. Run the preceding command, the result may look as follows (the actual values may be different on your computer):

```
[5.584s][info][gc] GC(14) Concurrent Cycle 3357.727ms
[5.912s][info][gc] GC(19) Pause Young (G1 Evacuation Pause) 2112M->2114M(3378M) 223.731ms
[6.231s][info][gc] GC(20) Pause Initial Mark (G1 Evacuation Pause) 2261M->2263M(3378M) 188.743ms
[6.231s][info][gc] GC(21) Concurrent Cycle
[9.387s][info][gc] GC(21) Pause Remark 2408M->2408M(3378M) 0.659ms
[9.873s][info][gc] GC(21) Pause Cleanup 2408M->1961M(3378M) 1.316ms
[9.884s][info][gc] GC(21) Concurrent Cycle 3652.662ms

real    0m10.096s
user    0m36.037s
sys     0m1.694s
```

You can see that the GC went through most of the steps we have described. It has started with collecting the young generation. Then, when the `List<AnObject> list` object (see the preceding code) becomes too big (more than 50% of a young generation region), the memory for it is allocated in the *humongous* region. You can also see the initial mark step and the following remark and other steps described earlier.

Each line starts with an accumulative time amount (in seconds) the JVM was running for and ends with the time (in milliseconds) that every step took. At the bottom of the screenshot we see three lines printed by the `time` utility:

- `real` is the amount of wall clock time spent--all the time elapsed (should align with the first column of the JVM uptime value) since the command was run
- `user` is the amount of time all the CPUs spent in the user-mode code (outside the kernel) within the process; it is bigger because GC worked concurrently with the application
- `sys` is the amount of time the CPU spent in the kernel within the process
- `user`+`sys` is the amount of CPU time the process used

5. Set the `-XX:+PrintGCDetails` option (or just add a `*` to the log option `-Xlog:gc*`) to see more details about GC activity (in the following screenshot, we provide only the beginning of the log related to GC step 0):

```
Chapter12Memory.main() for 99888999 is running...
[0.195s][info][gc,start     ] GC(0) Pause Young (G1 Evacuation Pause)
[0.196s][info][gc,task      ] GC(0) Using 8 workers of 8 for evacuation
[0.258s][info][gc,phases    ] GC(0)   Evacuate Collection Set: 62.1ms
[0.258s][info][gc,phases    ] GC(0)   Code Roots: 0.0ms
[0.258s][info][gc,phases    ] GC(0)   Clear Card Table: 0.0ms
[0.258s][info][gc,phases    ] GC(0)   Expand Heap After Collection: 0.0ms
[0.258s][info][gc,phases    ] GC(0)   Free Collection Set: 0.1ms
[0.258s][info][gc,phases    ] GC(0)   Merge Per-Thread State: 0.0ms
[0.258s][info][gc,phases    ] GC(0)   Other: 0.4ms
[0.258s][info][gc,heap      ] GC(0) Eden regions: 24->0(9)
[0.258s][info][gc,heap      ] GC(0) Survivor regions: 0->3(3)
[0.258s][info][gc,heap      ] GC(0) Old regions: 0->18
[0.258s][info][gc,heap      ] GC(0) Humongous regions: 23->23
[0.258s][info][gc,metaspace ] GC(0) Metaspace: 5493K->5493K(1056768K)
[0.258s][info][gc           ] GC(0) Pause Young (G1 Evacuation Pause) 47M->43M(256M) 62.699ms
[0.258s][info][gc,cpu       ] GC(0) User=0.40s Sys=0.03s Real=0.07s
```

Now the log has more than a dozen entries for each of the GC steps and ends up with logging the `User`, `Sys`, and `Real` amount of time (the amounts accumulated by the `time` utility) each step took. You can modify the program (by adding more short-lived objects, for example) and see how the GC activity changes.

6. Get even more information with the `-Xlog:gc*=debug` option (the following is a fragment only):

```
[gc,task,stats       ] GC(0) GC Termination Stats
[gc,task,stats       ] GC(0)      elapsed  --strong roots-- -------termination------- ------waste (KiB)------
[gc,task,stats       ] GC(0) thr     ms        ms      %        ms      %    attempts  total   alloc    undo
[gc,task,stats       ] GC(0) --- --------- --------- ------ --------- ------ -------- ------- ------- -------
[gc,task,stats       ] GC(0)   2     53.48     16.96  31.71      0.03   0.06       94      25      25       0
[gc,task,stats       ] GC(0)   3     53.52     17.07  31.90      0.03   0.05       94      15      15       0
[gc,task,stats       ] GC(0)   4     53.53     17.08  31.90      0.02   0.04       63      15      15       0
[gc,task,stats       ] GC(0)   0     53.55     16.89  31.54      0.02   0.04       83      20      20       0
[gc,task,stats       ] GC(0)   1     53.56     17.10  31.93      0.00   0.00        1      23      23       0
[gc,task,stats       ] GC(0)   7     53.52     17.02  31.80      0.03   0.06       93      21      21       0
[gc,task,stats       ] GC(0)   5     53.54     16.83  31.43      0.03   0.06      102      11      11       0
[gc,task,stats       ] GC(0)   6     53.54     17.03  31.81      0.03   0.06      106      23      23       0
[gc,ref,start        ] GC(0) SoftReference
[gc,ref              ] GC(0) SoftReference 0.014ms
```

So, it is up to you to choose how much info you need for the analysis.

We will discuss more details of the logging format and other log options in the following recipe, *Unified logging for JVM*, of this chapter.

How it works...

As we had mentioned earlier, the G1 GC uses default ergonomic values that probably would be good enough for most applications. Here is the list of the most important ones (`<ergo>` means that the actual value is determined ergonomically depending on the environment):

- `-XX:MaxGCPauseMillis=200` holds the value for the maximum pause time
- `-XX:GCPauseTimeInterval=<ergo>` holds the maximum pause time between GC steps (not set by default, allowing G1 to perform garbage collections back to back if need be)
- `-XX:ParallelGCThreads=<ergo>` holds the maximum number of threads used for parallel work during garbage collection pauses (by default, derived from the number of available threads; if the number of CPU threads available to the process is less than or equal to 8, it uses this number; otherwise, it adds five-eighths of the threads greater than 8 to the final number of threads)
- `-XX:ConcGCThreads=<ergo>` holds the maximum number of threads used for concurrent work (set by default as `-XX:ParallelGCThreads` divided by 4).
- `-XX:+G1UseAdaptiveIHOP` indicates that the initiating heap occupancy should be adaptive
- `-XX:InitiatingHeapOccupancyPercent=45` sets the first few collection cycles; G1 will use an occupancy of 45% of the old generation as the mark start threshold
- `-XX:G1HeapRegionSize=<ergo>` holds the heap region size based on the initial and maximum heap sizes (by default, because heap contains roughly 2048 heap regions, the size of a heap region can vary from 1 to 32 MB and must be a power of 2)
- `-XX:G1NewSizePercent=5` and`-XX:XX:G1MaxNewSizePercent=60` define the size of the young generation in total, which varies between these two values as percentages of the current JVM heap in use
- `-XX:G1HeapWastePercent=5` holds the allowed unreclaimed space in the collection set candidates as a percentage (G1 stops the space

reclamation if the free space in the collection set candidates is lower than that)
- `-XX:G1MixedGCCountTarget=8` holds the expected length of the space-reclamation phase in a number of collections)
- `-XX:G1MixedGCLiveThresholdPercent=85` holds the percentage of the live object occupancy of the old generation regions, after which a region won't to be collected in this space-reclamation phase

In general, the G1 goals in the default configuration are *to provide relatively small, uniform pauses at high throughput* (from the G1 documentation). If these default settings do not fit your application, you can change the pause time (using `-XX:MaxGCPauseMillis`) and the maximum Java heap size (using the `-Xmx` option). Note, though, that the actual pause time will not be an exact match at runtime, but G1 will try its best to meet the goal.

If you would like to increase the throughput, decrease the pause time goal or request a larger heap. To increase responsiveness, change the pause time value. Note, though, that the limiting of the young generation size (using the `-Xmn`, `-XX:NewRatio`, or other options) can impede the pause time control because *the young generation size is the main means for G1 to allow it to meet the pause time* (from the G1 documentation).

One of the first possible causes of poor performance can be full GC in the presence of pause full (allocation failure) in the log. It usually happens when too many objects are created in a quick succession (and cannot be collected quickly enough) or many large (humongous) objects cannot be allocated in a timely manner. There are several recommended ways to handle this condition:

- In the case of an excessive number of humongous objects, try to reduce their count by increasing the region size, using the `-XX:G1HeapRegionSize` option (the currently selected heap region size is printed at the beginning of the log).
- Increase the size of the heap.
- Increase the number of concurrent marking threads by setting `-XX:ConcGCThreads`.

- Facilitate the beginning of marking earlier (using the fact that G1 makes the decisions based on earlier application behavior). Increase the buffer used in an adaptive IHOP calculation by modifying `-XX:G1ReservePercent`, or disable the adaptive calculation of the IHOP by setting it manually using `-XX:-G1UseAdaptiveIHOP` and `-XX:InitiatingHeapOccupancyPercent`.

Only after addressing full GC can one start tuning the JVM for better responsiveness and/or throughput. The JVM documentation identifies the following cases for responsiveness tuning:

- Unusual system or real-time usage
- Reference processing takes too long
- Young-only collections take too long
- Mixed collections take too long
- High update RS and scan RS times

Better throughput can be achieved by decreasing the overall pause times and the frequency of the pauses. Refer to the JVM documentation for the identification and recommendations of mitigating the issues.

See also

Do refer to the following recipes in this chapter:

- Unified logging for JVM
- Using the new diagnostic commands for the JVM
- Some best practices for better memory usage

Unified logging for JVM

Java 9 implemented *JEP 158: Unified JVM Logging*, which requested to *introduce a common logging system for all the components of the JVM.* The main components of JVM include the following:

- Class loader
- Runtime data area
    - Stack area
    - Method area
    - Heap area
    - PC registers
    - Native method stack
- Execution engine
    - Interpreter
    - The JIT compiler
    - Garbage collection
    - Native method interface JNI
    - Native method library

The log message of all these components can now be captured and analyzed using unified logging, turned on by the `-Xlog` option.

The main features of the new logging system are as follows:

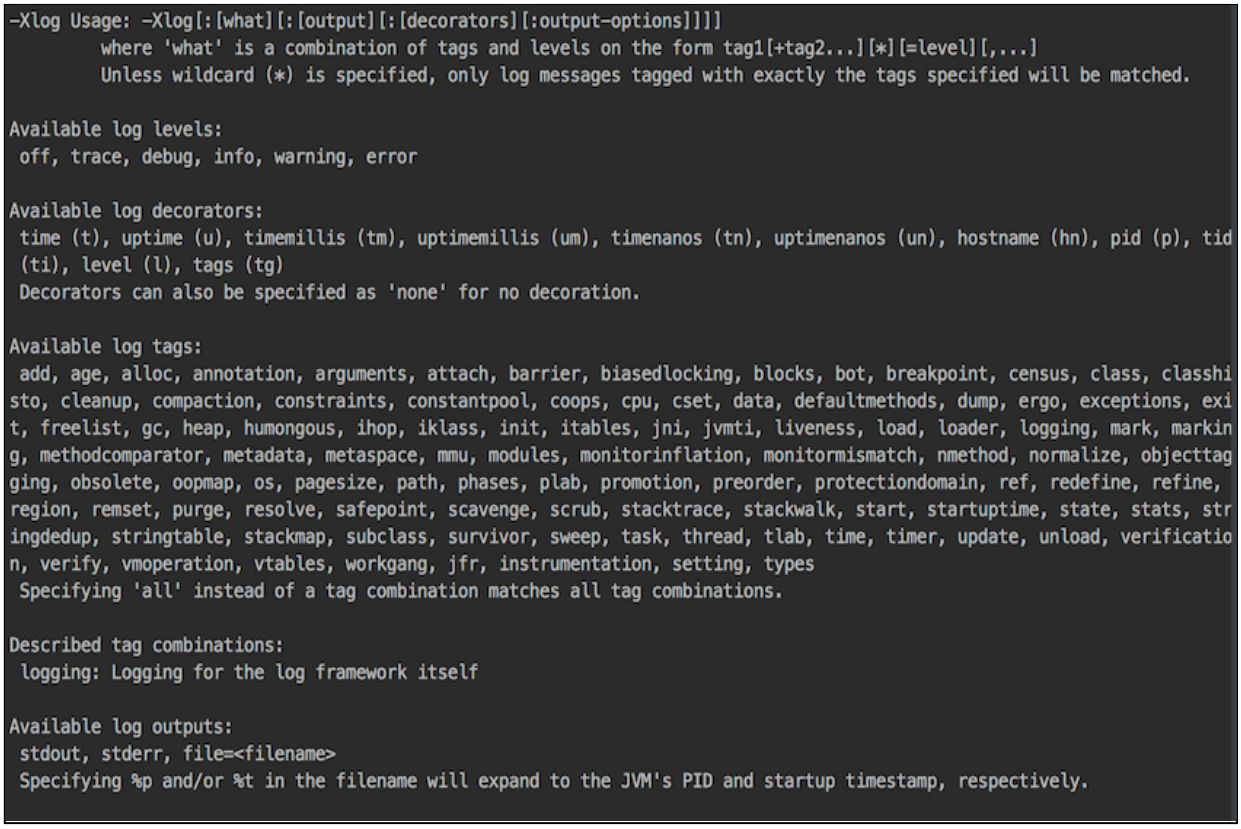
- Usage of the log levels: `trace`, `debug`, `info`, `warning`, `error`
- Message tags that identify the JVM component, action, or message of a specific interest
- Three output types: `stdout`, `stderr`, and `file`
- The enforcement of the one-message-per-line limit

Getting ready

To see all the logging possibilities at a glance, you can run the following command:

```
java -Xlog:help
```

Here is the output:



```
-Xlog Usage: -Xlog[:[what][:[output][:[decorators][:output-options]]]]
        where 'what' is a combination of tags and levels on the form tag1[+tag2...][*][=level][,...]
        Unless wildcard (*) is specified, only log messages tagged with exactly the tags specified will be matched.

Available log levels:
 off, trace, debug, info, warning, error

Available log decorators:
 time (t), uptime (u), timemillis (tm), uptimemillis (um), timenanos (tn), uptimenanos (un), hostname (hn), pid (p), tid
 (ti), level (l), tags (tg)
 Decorators can also be specified as 'none' for no decoration.

Available log tags:
 add, age, alloc, annotation, arguments, attach, barrier, biasedlocking, blocks, bot, breakpoint, census, class, classhi
sto, cleanup, compaction, constraints, constantpool, coops, cpu, cset, data, defaultmethods, dump, ergo, exceptions, exi
t, freelist, gc, heap, humongous, ihop, iklass, init, itables, jni, jvmti, liveness, load, loader, logging, mark, markin
g, methodcomparator, metadata, metaspace, mmu, modules, monitorinflation, monitormismatch, nmethod, normalize, objecttag
ging, obsolete, oopmap, os, pagesize, path, phases, plab, promotion, preorder, protectiondomain, ref, redefine, refine,
region, remset, purge, resolve, safepoint, scavenge, scrub, stacktrace, stackwalk, start, startuptime, state, stats, str
ingdedup, stringtable, stackmap, subclass, survivor, sweep, task, thread, tlab, time, timer, update, unload, verificatio
n, verify, vmoperation, vtables, workgang, jfr, instrumentation, setting, types
 Specifying 'all' instead of a tag combination matches all tag combinations.

Described tag combinations:
 logging: Logging for the log framework itself

Available log outputs:
 stdout, stderr, file=<filename>
 Specifying %p and/or %t in the filename will expand to the JVM's PID and startup timestamp, respectively.
```

As you can see, the format of the `-Xlog` option is defined as follows:

```
-Xlog[:[what][:[output][:[decorators][:output-options]]]]
```

Let's explain the option in detail:

- The `what` is a combination of tags and levels of the form, `tag1[+tag2...][*][=level][,...]`. We have already demonstrated how this construct works

when we used the `gc` tag in the `-Xlog:gc*=debug` option. The wildcard (`*`) indicates that you'd like to see all the messages that have the `gc` tag (maybe among other tags). An absence of the `-Xlog:gc=debug` wildcard indicates that you would like to see messages marked by one tag (`gc`, in this case) only. If only `-Xlog` is used, the log will show all the messages at the `info` level.

- The `output` sets the type of output (the default is `stdout`).
- The `decorators` indicate what will be placed at the beginning of each line of the log (before the actual log message comes from a component). Default decorators are `uptime`, `level`, and `tags`, each included in square brackets.
- The `output_options` may include `filecount=file count` and/or `filesize=file size with optional K, M or G suffix.`

To summarize, the default log configuration is as follows:

```
-Xlog:all=info:stdout:uptime,level,tags
```

How to do it...

Let's run some of the log settings, which are as follows:

1. Run the following command:

```
java -Xlog:cpu -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

   There are no messages because the JVM does not log messages with the `cpu` tag only. The tag is used in combination with other tags.

2. Add a `*` sign and run the command again:

```
java -Xlog:cpu* -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

   The result will look as follows:

```
Chapter12Memory.main() for 99888999 is running...
[0.241s][info][gc,cpu] GC(0) User=0.33s Sys=0.03s Real=0.05s
[0.291s][info][gc,cpu] GC(1) User=0.23s Sys=0.00s Real=0.03s
[0.316s][info][gc,cpu] GC(2) User=0.11s Sys=0.01s Real=0.01s
[0.355s][info][gc,cpu] GC(3) User=0.13s Sys=0.00s Real=0.01s
[0.391s][info][gc,cpu] GC(4) User=0.14s Sys=0.01s Real=0.03s
[0.598s][info][gc,cpu] GC(6) User=0.45s Sys=0.03s Real=0.06s
[0.673s][info][gc,cpu] GC(7) User=0.33s Sys=0.01s Real=0.05s
[0.676s][info][gc,cpu] GC(5) User=0.01s Sys=0.00s Real=0.00s
[0.703s][info][gc,cpu] GC(5) User=0.00s Sys=0.00s Real=0.00s
[0.793s][info][gc,cpu] GC(8) User=0.33s Sys=0.00s Real=0.04s
```

   As you can see, the tag `cpu` brings only messages that log time it took a garbage collection task to execute. Even if we set the log level to `trace` or `debug` (`-Xlog:cpu*=debug`, for example), no other messages will be shown.

3. Now run the command with the `heap` tag:

```
java -Xlog:heap* -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

   You will only get heap-related messages:

```
Chapter12Memory.main() for 99888999 is running...
[0.240s][info][gc,heap      ] GC(0) Eden regions: 24->0(9)
[0.240s][info][gc,heap      ] GC(0) Survivor regions: 0->3(3)
[0.240s][info][gc,heap      ] GC(0) Old regions: 0->18
[0.240s][info][gc,heap      ] GC(0) Humongous regions: 23->23
[0.293s][info][gc,heap      ] GC(1) Eden regions: 9->0(10)
[0.293s][info][gc,heap      ] GC(1) Survivor regions: 3->2(2)
[0.293s][info][gc,heap      ] GC(1) Old regions: 18->29
[0.293s][info][gc,heap      ] GC(1) Humongous regions: 34->34
```

But let's look closer at the first line. It starts with three decorators--`uptime`, `log` level, and `tags`--and then with the message itself, which starts with the collection cycle number (0 in this case) and the information that the number of Eden regions dropped from 24 to 0 (and their count now is 9). It happened because (as we see in the next line) the count of survivor regions grew from 0 to 3 and the count of the old generation (the third line) grew to 18, while the count of humongous regions (23) did not change. These are all the heap-related messages in the first collection cycle. Then, the second collection cycle starts.

4. Add the `cpu` tag again and run:

```
java -Xlog:heap*,cpu* -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

As you can see, the `cpu` message shows how long each cycle took:

```
Chapter12Memory.main() for 99888999 is running...
[0.469s][info][gc,heap      ] GC(0) Eden regions: 24->0(9)
[0.469s][info][gc,heap      ] GC(0) Survivor regions: 0->3(3)
[0.469s][info][gc,heap      ] GC(0) Old regions: 0->18
[0.469s][info][gc,heap      ] GC(0) Humongous regions: 23->23
[0.469s][info][gc,cpu       ] GC(0) User=0.68s Sys=0.06s Real=0.10s
[0.572s][info][gc,heap      ] GC(1) Eden regions: 9->0(10)
```

5. Try and use two tags combined via sign `+` (`-Xlog:gc+heap`, for example). It brings up only the messages that have both tags (similar to the binary `AND` operation). Notice that a wildcard will not work together with the `+` sign (`-Xlog:gc*+heap` , for example, does not work).
6. You can also select the output type and decorators. In practice, the decorator level does not seem very informative and can be easily

omitted by explicitly listing only the decorators that are needed. Consider the following example:

```
java -Xlog:heap*,cpu*::uptime,tags -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

Note how the two colons (`:`) were inserted to preserve the default setting of the output type. We could also show it explicitly:

```
java -Xlog:heap*,cpu*:stdout:uptime,tags -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

To remove any decoration, one can set them to `none`:

```
java -Xlog:heap*,cpu*::none -cp ./cookbook-1.0.jar
com.packt.cookbook.ch12_memory.Chapter12Memory
```

The most useful aspect of a new logging system is tag selection. It allows a better analysis of the memory evolution of each JVM component and its subsystems or to find the performance bottleneck, analyzing the time spent in each collection phase--both are critical for the JVM and application tuning.

See also

Do refer to the other recipes of this chapter:

- Using the new diagnostic commands for the JVM
- Stack walking for improved debugging
- Some best practices for better memory usage

Using the new diagnostic commands for the JVM

If you open the `bin` folder of the Java installation, you can find there quite a few command-line utilities that can be used to diagnose issues and monitor an application deployed with the **Java Runtime Environment** (**JRE**). They use different mechanisms to get the data they report. The mechanisms are specific to the **Virtual Machine** (**VM**) implementation, operating systems, and release. Typically, only a subset of the tools is applicable to a given issue at a particular time.

In this recipe, we will focus on the new diagnostic commands introduced in Java 9, some of them listed in *JEP 228: Add More Diagnostic Commands* (the actual command names are slightly different than in JEP). The new diagnostic commands were implemented as commands of the command-line utility, `jcmd`. You can find this utility in the same `bin` folder of the Java installation and can invoke it (if this folder is on the path) just by typing `jcmd` on the command prompt.

If you do type it and there is no Java process currently running on the machine, you will get back only one line, which looks as follows:

```
87863 jdk.jcmd/sun.tools.jcmd.JCmd
```

It shows that only one Java process is currently running (the `jcmd` utility itself) and it has the **process identifier** (**PID**) 87863 (which will be different with each run).

Let's run another Java program, for example:

```
java -cp ./cookbook-1.0.jar com.packt.cookbook.ch12_memory.Chapter12Memory
```

The output of `jcmd` will show (with different PIDs) the following:

```
87864 jdk.jcmd/sun.tools.jcmd.JCmd
87785 com.packt.cookbook.ch12_memory.Chapter12Memory
```

As you can see, if entered without any options, the `jcmd` utility reports the PIDs of all the currently running Java processes. After getting the PID, you can then use `jcmd` to request data from the JVM that runs the process:

```
jcmd 88749 VM.version
```

Alternatively, you can avoid using PID (and calling `jcmd` without parameters) by referring to the process by the main class of the application:

```
jcmd Chapter12Memory VM.version
```

You can read the JVM documentation for more details about the `jcmd` utility and how to use it. In this recipe, we will focus only on the new diagnostic commands that came with Java 9.

How to do it...

1. Get the full list of the available `jcmd` commands by running the following line:

```
jcmd PID/main-class-name help
```

   Instead of `PID/main-class`, put the process identifier or the main class name. The list is specific to the JVM, so the command requests the data from the specific process.

2. If you can, compile the same class with JDK 8 and with JDK 9 and run the preceding command for each of the JSK versions. This way, you can compare the lists and see that JDK 9 introduced the following new `jcmd` commands:

   - `Compiler.queue`: This prints the methods queued for compilation with either C1 or C2 (separate queues)
   - `Compiler.codelist`: This prints n-methods (compiled) with full signature, address range, and state (alive, non-entrant, and zombie) and allows the selection of printing to `stdout` , a file, XML, or text printout
   - `Compiler.codecache`: This prints the content of the code cache, where the JIT compiler stores the generated native code to improve performance
   - `Compiler.directives_add file`: This adds compiler directives from a file to the top of the directives stack
   - `Compiler.directives_clear`: This clears the compiler directives stack (leaves the default directives only)
   - `Compiler.directives_print`: This prints all the directives on the compiler directives stack from top to bottom
   - `Compiler.directives_remove`: This removes the top directive from the compiler directives stack
   - `GC.heap_info`: This prints the current heap parameters and status

- `GC.finalizer_info`: This shows the status of the finalizer thread, which collects objects with a finalizer (that is, a `finalize()` method)
- `JFR.configure`: This allows configuring the Java Flight Recorder
- `JVMTI.data_dump`: This prints the Java Virtual Machine Tool Interface data dump
- `JVMTI.agent_load`: This loads (attaches) the Java Virtual Machine Tool Interface agent
- `ManagementAgent.status`: This prints the status of the remote JMX agent
- `Thread.print`: This prints all the threads with stack traces
- `VM.log [option]`: This allows setting the JVM log configuration (which we described in the previous recipe) at runtime, after the JVM has started (the availability can be seen by using `VM.log list`)
- `VM.info`: This prints the unified JVM info (version and configuration), a list of all threads and their state (without thread dump and heap dump), heap summary, JVM internal events (GC, JIT, safepoint, and so on), memory map with loaded native libraries, VM arguments and environment variables, and details of the operation system and hardware
- `VM.dynlibs`: This prints information about dynamic libraries
- `VM.set_flag`: This allows setting the JVM *writable* (also called *manageable*) flags (see the JVM documentation for a list of the flags)
- `VM.stringtable` and `VM.symboltable`: These print all UTF-8 string constants
- `VM.class_hierarchy [full-class-name]`: This prints all the loaded classes or just a specified class hierarchy
- `VM.classloader_stats`: This prints information about the classloader
- `VM.print_touched_methods`: This prints all the methods that have been touched at runtime

As you can see, these new commands belong to several groups, denoted by the prefix compiler, **garbage collector** (**GC**), **Java Flight Recorder** (**JFR**), **Java Virtual Machine Tool Interface** (**JVMTI**), **Management Agent** (related to remote JMX agent), thread, and VM. In

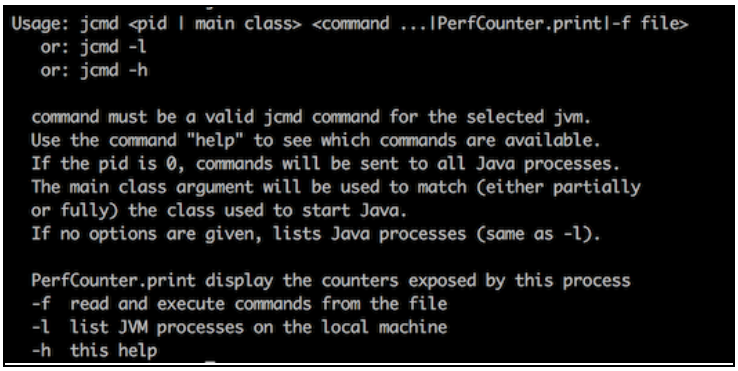
this book, we do not have enough space to go through each command in detail. We will only demonstrate the usage of a few most practical ones.

How it works...

To get help for the `jcmd` utility, run the following command:

```
jcmd -h
```

Here is the result of the command:

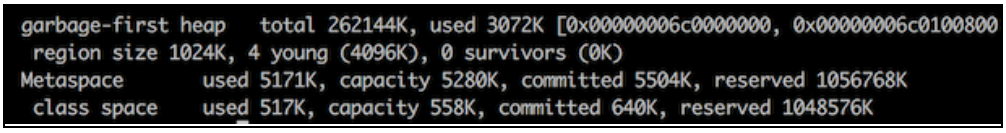


It tells us that the commands can also be read from the file specified after `-f` and that there is a `PerfCounter.print` command, which prints all the performance counters (statistics) of the process.

Let's run the following command:

```
jcmd Chapter12Memory GC.heap_info
```

The output may look as follows:



It shows the total heap size and how much of it was used, the size of a region in the young generation and how many regions are allocated, and the parameters of `Metaspace` and `class space`.

The following command is very helpful in case you are looking for runaway threads or would like to know what else is going on behind the scenes:

```
jcmd Chapter12Memory Thread.print
```

Here is a fragment of the possible output:

```
"main" #1 prio=5 os_prio=31 tid=0x00007ff803002800 nid=0xb07 waiting on condition [0x0000700000219000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(java.base@9-ea/Native Method)
        at com.packt.cookbook.ch12_memory.Chapter12Memory.demo2_ProcessForJcmd(Chapter12Memory.java:32)
        at com.packt.cookbook.ch12_memory.Chapter12Memory.main(Chapter12Memory.java:10)

"VM Thread" os_prio=31 tid=0x00007ff803098000 nid=0x5203 runnable

"GC Thread#0" os_prio=31 tid=0x00007ff803807800 nid=0x2803 runnable
```

The following command is probably used most often, as it produces a wealth of information about the hardware, the JVM process as a whole, and the current state of its components:

```
jcmd Chapter12Memory VM.info
```

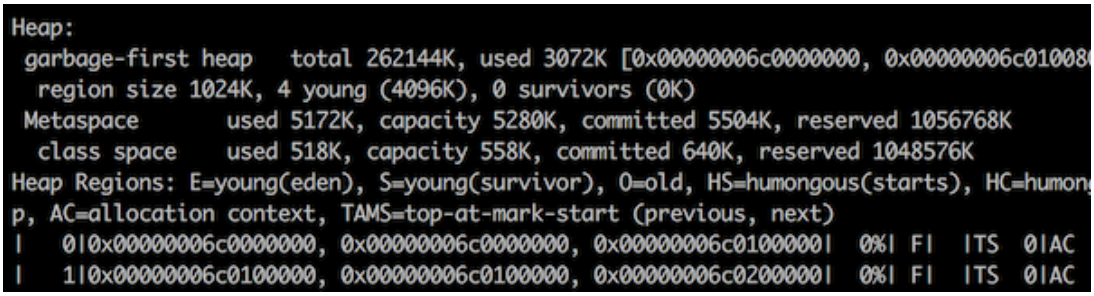
It starts with a summary, as follows:

```
# JRE version: Java(TM) SE Runtime Environment (9.0+143) (build 9-ea+143)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (9-ea+143, mixed mode, tiered, compressed

---------------  S U M M A R Y ------------

Command Line: com.packt.cookbook.ch12_memory.Chapter12Memory

Host: MacBookPro11,5 x86_64 2500 MHz, 8 cores, 16G, Darwin 15.6.0
Time: Mon May 15 21:34:45 2017 MDT elapsed time: 388 seconds (0d 0h 6m 28s)
```

Then the general process description follows:

```
---------------  P R O C E S S  ---------------

Heap address: 0x00000006c0000000, size: 4096 MB, Compressed Oops mode: Zero based,
Narrow klass base: 0x0000000000000000, Narrow klass shift: 3
Compressed class space size: 1073741824 Address: 0x00000007c0000000
```

Then come the details of the heap (this is only a tiny fragment of it):



It then prints the compilation events, GC heap history, deoptimization events, internal exceptions, events, dynamic libraries, logging options, environment variables, VM arguments, and many parameters of the system running the process.

The `jcmd` commands give a deep insight into the JVM process, which helps us debug and tune the process for best performance and optimal resource usage.

See also

Do refer to the other recipes of this chapter:

- Stack walking for improved debugging
- Some best practices for better memory usage

Try with resources for better resource handling

Managing resources is important. Here is an excerpt from the JDK 7 documentation:

"The typical Java application manipulates several types of resources such as files, streams, sockets, and database connections. Such resources must be handled with great care, because they acquire system resources for their operations. Thus, you need to ensure that they get freed even in case of errors. Indeed, incorrect resource management is a common source of failures in production applications, with the usual pitfalls being database connections and file descriptors remaining opened after an exception has occurred somewhere else in the code. This leads to application servers being frequently restarted when resource exhaustion occurs, because operating systems and server applications generally have an upper-bound limit for resources."

The phrase *because they acquire system resources for their operations* is the key. It means that mishandling (not releasing) the resources can exhaust the system's capability to operate. That's why, in JDK 7 the *try-with-resources* statement was introduced and we used it in the examples of `Chapter 6`, *Database Programming*:

```
try (Connection conn = getDbConnection()){
  try (Statement st = createStatement(conn)) {
    st.execute(sql);
  }
} catch (Exception ex) {
  ex.printStackTrace();
}
```

Another variation of the same statement is to include the acquisition of both the resources into the same `try` block:

```
try (Connection conn = getDbConnection();
Statement st = createStatement(conn)) {
```

```
  st.execute(sql);
} catch (Exception ex) {
  ex.printStackTrace();
}
```

As a reminder, we used the `getDbConnection()` and `createStatement()` methods. Here is the `getDbConnection()` method:

```
Connection getDbConnection() {
  PGPoolingDataSource source = new PGPoolingDataSource();
  source.setServerName("localhost");
  source.setDatabaseName("cookbook");
  try {
    return source.getConnection();
  } catch(Exception ex) {
    ex.printStackTrace();
    return null;
  }
}
```

Here is the `createStatement()` method:

```
Statement createStatement(Connection conn) {
  try {
    return conn.createStatement();
  } catch(Exception ex) {
    ex.printStackTrace();
    return null;
  }
}
```

This was very helpful, but in some cases, we were still required to write extra code in old style. For example, if there is an `execute()` method that accepts a `Statement` object as a parameter, and we would like to release (close) it as soon as it was used. In such a case, the code will look as follows:

```
void execute(Statement st, String sql){
  try {
    st.execute(sql);
  } catch (Exception ex) {
    ex.printStackTrace();
  } finally {
    if(st != null) {
      try{
        st.close();
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

As you can see, most of it is just boilerplate copy-and-paste code.

The new *try-with-resources* statement (coming with Java 9) addresses this case by allowing effectively-final variables to be used as resources in the *try-with-resources* statement.

How to do it...

1. Rewrite the previous example using the new *try-with-resources* statement:

```
void execute(Statement st, String sql){
  try (st) {
    st.execute(sql);
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

   As you can see, it is much more concise and focused, without the need of repeatedly writing trivial code that closes the resource. No more `finally` and extra `try...catch` in it.

2. Try and write it so that it closes the connection as soon as it was used:

```
void execute(Connection conn, Statement st, String sql) {
  try (conn; st) {
    st.execute(sql);
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

   It may or may not fit your application connection handling, but often, this capability is handy.

3. Try a different combination, such as the following:

```
Connection conn = getDbConnection();
Statement st = conn.createStatement();
try (conn; st) {
  st.execute(sql);
} catch (Exception ex) {
  ex.printStackTrace();
}
```

   You can also try this combination:

```
Connection conn = getDbConnection();
try (conn; Statement st = conn.createStatement()) {
```

```
    st.execute(sql);
  } catch (Exception ex) {
    ex.printStackTrace();
  }
```

The new statement provides more flexibility to write code that fits the needs without writing lines to close the resource.

The only requirements are as follows:

- The variable included in the `try` statement has to be final or effectively final
- The resource has to implement the `AutoCloseable` interface , which includes only one method:

```
void close() throws Exception;
```

How it works...

To demonstrate how the new statement works, let's create our own resources that implement `AutoCloseable` and use them in a fashion similar to the resources of the previous examples.

Here is one resource:

```
class MyResource1 implements AutoCloseable {
  public MyResource1(){
    System.out.println("MyResource1 is acquired");
  }
  public void close() throws Exception {
    //Do what has to be done to release this resource
    System.out.println("MyResource1 is closed");
  }
}
```

Here is the second resource:

```
class MyResource2 implements AutoCloseable {
  public MyResource2(){
    System.out.println("MyResource2 is acquired");
  }
  public void close() throws Exception {
    //Do what has to be done to release this resource
    System.out.println("MyResource2 is closed");
  }
}
```

Let's use them in the code example:

```
MyResource1 res1 = new MyResource1();
MyResource2 res2 = new MyResource2();
try (res1; res2) {
  System.out.println("res1 and res2 are used");
} catch (Exception ex) {
  ex.printStackTrace();
}
```

If we run it, the result will be as follows:

```
MyResource1 is acquired
MyResource2 is acquired
res1 and res2 are used
MyResource2 is closed
MyResource1 is closed
```

Note that the resource listed first in the `try` statement is closed last. If we make only one change and switch the order in the `try` statement:

```
MyResource1 res1 = new MyResource1();
MyResource2 res2 = new MyResource2();
try (res2; res1) {
  System.out.println("res1 and res2 are used");
} catch (Exception ex) {
  ex.printStackTrace();
}
```

The output confirms it:

```
MyResource1 is acquired
MyResource2 is acquired
res1 and res2 are used
MyResource1 is closed
MyResource2 is closed
```

This rule of closing the resources in the reverse order addresses the most important possible issue of dependency between resources, but it is up to the programmer to define the sequence of closing the resources (by listing them in the `try` statement in the correct order). Fortunately, the closing of most standard resources is handled by the JVM gracefully, and the code does not break if the resources are listed in incorrect order. Still, it is a good idea to list them in the same sequence as they were created.

See also

Do refer to the other recipes of this chapter:

- Some best practices for better memory usage

Stack walking for improved debugging

Stack trace can be very helpful in figuring out the source of the problem, although the need to read it is usually caused by some unpleasant surprise. Once in a while, especially in a big and complex system, the need arises to read it programmatically when an automatic correction is possible.

Since Java 1.4, the current stack trace can be accessed via the `java.lang.Thread` and `java.lang.Throwable` classes. To any method of your code, you can add the following line:

```
Thread.currentThread().dumpStack();
```

You can also add the following line:

```
new Throwable().printStackTrace();
```

It will print the stack trace to the standard output. Alternatively, since Java 8, you can use any of the following lines for the same effect:

```
Arrays.stream(Thread.currentThread().getStackTrace())
      .forEach(System.out::println);

Arrays.stream(new Throwable().getStackTrace())
      .forEach(System.out::println);
```

You can use the stack trace to find the fully qualified name of the caller class, using one of these lines:

```
System.out.println("This method is called by "+Thread.currentThread()
                        .getStackTrace()[1].getClassName());

System.out.println("This method is called by " + new Throwable()
                        .getStackTrace()[0].getClassName());
```

All the aforementioned lines are possible because of the `java.lang.StackTraceElement` class, which represents a stack frame in a stack trace. This class provides other methods that describe the execution point

represented by this stack trace element, which allows programmatic access to the stack trace information. For example, you can run this code snippet from anywhere in your program:

```
Arrays.stream(Thread.currentThread().getStackTrace())
  .forEach(e -> {
    System.out.println();
    System.out.println("e="+e);
    System.out.println("e.getFileName()="+ e.getFileName());
    System.out.println("e.getMethodName()="+ e.getMethodName());
    System.out.println("e.getLineNumber()="+ e.getLineNumber());
});
```

You can also run this one from anywhere in the program:

```
Arrays.stream(new Throwable().getStackTrace())
  .forEach(x -> {
    System.out.println();
    System.out.println("x="+x);
    System.out.println("x.getFileName()="+ x.getFileName());
    System.out.println("x.getMethodName()="+ x.getMethodName());
    System.out.println("x.getLineNumber()="+ x.getLineNumber());
});
```

In either case, you can see how much information you get. Unfortunately, this wealth of data comes with a price. The JVM captures the entire stack (except for hidden stack frames), and it may affect the performance, while the chances are that you only need a fraction of this data (like in the preceding example where we use only one element of the stack trace array).

This is where the new Java 9 class, `java.lang.StackWalker`, with its nested `Option` class and `StackFrame` interface, comes in handy.

Getting ready

The `StackWalker` class has four variants of the static factory method, `getInstance()`, which are different only in their ability to take one of the following several options or no options at all:

- `StackWalker getInstance()`: This is configured to skip all the hidden frames and no caller class reference.
- `StackWalker getInstance(StackWalker.Option option)`: This creates an instance with the given option, specifying the stack frame information it can access.
- `StackWalker getInstance(Set<StackWalker.Option> options)`: This creates an instance with the given set of options, specifying the stack frame information it can access. If the given set is empty, the instance is configured exactly like an instance of `StackWalker getInstance()`.
- `StackWalker getInstance(Set<StackWalker.Option> options, int estimatedDepth)`: This creates a similar instance as the preceding one and accepts the `estimatedDepth` parameter, which specifies the estimated number of stack frames this instance will traverse so that it can estimate the buffer size it might need.

One of the following values is passed as an option:

- `StackWalker.Option.RETAIN_CLASS_REFERENCE`
- `StackWalker.Option.SHOW_HIDDEN_FRAMES`
- `StackWalker.Option.SHOW_REFLECT_FRAMES`

The `StackWalker` class also has three methods:

- `T walk(Function<Stream<StackWalker.StackFrame>, T> function)`: This applies the given function to the stream of `StackFrames` for the current thread, traversing from the top frame of the stack, which is the method calling this walk method.

- `void forEach(Consumer<StackWalker.StackFrame> action)`: This performs the given action on each element of the `StackFrame` stream of the current thread, traversing from the top frame of the stack, which is the method calling the `forEach` method. This method is equivalent to calling `walk(s -> { s.forEach(action); return null; }).`
- `Class<?> getCallerClass()`: This gets the `Class` object of the caller that invoked the method that invoked `getCallerClass()`. This method throws `UnsupportedOperationException` if this `StackWalker` instance is not configured with the `RETAIN_CLASS_REFERENCE` option.

How to do it...

Create several classes and methods that will call each other, so you can perform stack trace processing.

1. Create a `Clazz01` class:

```
public class Clazz01 {
  public void method(){
    new Clazz03().method("Do something");
    new Clazz02().method();
  }
}
```

2. Create a `Clazz02` class:

```
public class Clazz02 {
  public void method(){
    new Clazz03().method(null);
  }
}
```

3. Create a `Clazz03` class:

```
public class Clazz03 {
  public void method(String action){
    if(action != null){
      System.out.println(action);
      return;
    }
    System.out.println("Throw the exception:");
    action.toString();
  }
}
```

4. Write a `demo4_StackWalk()` method:

```
private static void demo4_StackWalk(){
  new Clazz01().method();
}
```

Call this method from the main method of the `Chapter12Memory` class:

```
public class Chapter12Memory {
  public static void main(String... args) {
    demo4_StackWalk();
  }
}
```

If we run now the `Chapter12Memory` class, the result will be as follows:

```
Do something

Throw the exception:
Exception in thread "main" java.lang.NullPointerException
    at com.packt.cookbook.ch12_memory.walk.Clazz03.method(Clazz03.java:13)
    at com.packt.cookbook.ch12_memory.walk.Clazz02.method(Clazz02.java:6)
    at com.packt.cookbook.ch12_memory.walk.Clazz01.method(Clazz01.java:7)
    at com.packt.cookbook.ch12_memory.Chapter12Memory.demo4_StackWalk(Chapter12Memory.java:194)
    at com.packt.cookbook.ch12_memory.Chapter12Memory.main(Chapter12Memory.java:17)
```

The message, `Do something`, is passed from `Clazz01` and printed out in `Clazz03`. Then `Clazz02` passes null to `Clazz03`, and the message, `Throw the exception:`, is printed out before the stack trace caused by the `NullPointerException` from the `action.toString();` line.

How it works...

For a deeper understanding of the concepts here, let's modify `Clazz03`:

```
public class Clazz03 {
  public void method(String action){
    if(action != null){
      System.out.println(action);
      return;
    }
    System.out.println("Print the stack trace:");
    Thread.currentThread().dumpStack();
  }
}
```

The result will be the following:

```
Do something

Print the stack trace:
java.lang.Exception: Stack trace
    at java.lang.Thread.dumpStack(java.base@9-ea/Thread.java:1435)
    at com.packt.cookbook.ch12_memory.walk.Clazz03.method(Clazz03.java:21)
    at com.packt.cookbook.ch12_memory.walk.Clazz02.method(Clazz02.java:7)
    at com.packt.cookbook.ch12_memory.walk.Clazz01.method(Clazz01.java:8)
    at com.packt.cookbook.ch12_memory.Chapter12Memory.demo4_StackWalk(Chapter12Memory.java:195)
    at com.packt.cookbook.ch12_memory.Chapter12Memory.main(Chapter12Memory.java:17)
```

Alternatively, we can get a similar output by using `Throwable` instead of `Thread`:

```
new Throwable().printStackTrace();
```

It does look very familiar:

```
Do something

Print the stack trace:
java.lang.Throwable
    at com.packt.cookbook.ch12_memory.walk.Clazz03.method(Clazz03.java:16)
    at com.packt.cookbook.ch12_memory.walk.Clazz02.method(Clazz02.java:6)
    at com.packt.cookbook.ch12_memory.walk.Clazz01.method(Clazz01.java:7)
    at com.packt.cookbook.ch12_memory.Chapter12Memory.demo4_StackWalk(Chapter12Memory.java:194)
    at com.packt.cookbook.ch12_memory.Chapter12Memory.main(Chapter12Memory.java:17)
```

A similar result will produce each of the following two lines:

```
Arrays.stream(Thread.currentThread().getStackTrace())
                                .forEach(System.out::println);
```

```
Arrays.stream(new Throwable().getStackTrace())
                              .forEach(System.out::println);
```

Now with Java 9, the same output is achieved by using the `StackWalker` class. Let's look at what happens if we modify `Clazz03` as follows:

```
public class Clazz03 {
  public void method(String action){
    if(action != null){
      System.out.println(action);
      return;
    }
    StackWalker stackWalker = StackWalker.getInstance();
    stackWalker.forEach(System.out::println);
  }
}
```

The result will be the same:

```
Do something

Print the stack trace:
com.packt.cookbook.ch12_memory.walk.Clazz03.method(Clazz03.java:15)
com.packt.cookbook.ch12_memory.walk.Clazz02.method(Clazz02.java:6)
com.packt.cookbook.ch12_memory.walk.Clazz01.method(Clazz01.java:7)
com.packt.cookbook.ch12_memory.Chapter12Memory.demo4_StackWalk(Chapter12Memory.java:194)
com.packt.cookbook.ch12_memory.Chapter12Memory.main(Chapter12Memory.java:17)
```

However, contrary to the full stack trace generated and stored in an array in the memory, the `StackWalker` class only brings the requested elements. This is already a big plus. However, `StackWalker` shows the biggest advantage when we need the caller class name only. Instead of getting all the arrays and using only one element, we can now get the info we need by using the following two lines:

```
System.out.println("Print the caller class name:");
System.out.println(StackWalker.getInstance(StackWalker
                        .Option.RETAIN_CLASS_REFERENCE)
                        .getCallerClass().getSimpleName());
```

We will get the result:

```
Print the caller class name:
Clazz02
```

See also

Do refer to the other recipes of this chapter:

- Some best practices for better memory usage

Some best practices for better memory usage

Memory management may never become an issue for you, or it can be a haunting never-ending story of your life, and anything in between. It is probably going to be a non-issue for a majority of programmers, especially with the constantly improving garbage collection algorithms. The G1 garbage collector (default in JVM 9) is definitely a step in the right direction. But there is also a chance you will be called (or will notice yourself) about the degrading application performance, and that is when you'll learn how well you are equipped to meet the challenge.

This recipe is an attempt to help you avoid such a situation or to get out of it successfully.

How it works...

The first line of defense is the code itself. In the previous recipes, we have already discussed the need to release resources as soon as they are not needed anymore and the usage of `StackWalker` to consume less memory. There are plenty of recommendations on the internet, but they might not apply to your application. You'll have to monitor the memory consumption and test your design decisions, especially if your code handles a lot of data before deciding where to concentrate your attention.

For example, the choice of the collection (different collections use more or less memory) may be irrelevant if your collection is going to be small. However, programmers usually use the same coding pattern, and one can identify the code's author by their style. That's why it pays back in a long run to figure the most efficient constructs and use them routinely. However, try to avoid making your code difficult to understand; readability is an important aspect of the code quality.

Here are a few most popular pieces of advice on memory-aware coding style:

- Use lazy initialization and create an object just before the usage, especially if there is a good chance, this need may never materialize at all
- Use `StringBuilder` instead of the `+` operator
- Use `ArrayList` if it fits your needs, before using `HashSet` (the memory usage increases from `ArrayList` to `LinkedList`, `HashTable`, `HashMap`, and `HashSet`, in this sequence)
- Avoid regular expressions and cache `Pattern` references if you cannot avoid them
- Prefer primitives over the class wrappers (use autoboxing)
- Don't forget to clean the cache and remove unnecessary entries
- Pay attention to the object created inside the loop

Test and profile your code as soon as it starts doing what it was supposed to do. You might need to change your design or some details of implementation. It will also inform your future decisions. There are many profilers and diagnostic tools available for any environment. We described one of them (`jcmd`) in the *Using the new diagnostic commands for the JVM* recipe of this chapter.

Learn how your garbage collector works (see the recipe, *Understanding the G1 garbage collector*) and do not forget to use JVM logging (described in the recipe, *Unified logging for JVM*).

After that, you might need to tune the JVM and garbage collector. Here are a few most often used JVM parameters (the size is specified in bytes by default, but you can append the letter k or K to indicate kilobytes, m or M to indicate megabytes, g or G to indicate gigabytes):

- `-Xms size`: This sets the initial size of the heap, which must be a multiple of 1024 and greater than 1 MB.
- `-Xmx size`: This sets the maximum size of the heap, which must be a multiple of 1024 and greater than 2 MB. The default value is chosen at runtime based on the system configuration. For server deployments, `-Xms size` and `-Xmx size` are often set to the same value. The actual memory usage may exceed the amount you have set by `-Xmx size`, because it limits the Java heap size only, while the JVM allocates memory for other purposes too, including a stack for each thread.

- `-Xmn size`: This sets the initial and maximum size of the heap for the young generation (nursery). If the size for the young generation is too small, then a lot of minor garbage collections will be performed. If the size is too large, then only full garbage collections will be performed, which can take a long time to complete. Oracle recommends that you keep the size for the young generation greater than 25% and less than 50% of the overall heap size. This parameter is equivalent to `-XX:NewSize=size`. For efficient garbage collection, `-Xmn size` should be lower than `-Xmx size`.

- `-XX:MaxNewSize=size`: This sets the maximum size of the heap for the young generation (nursery). The default value is set ergonomically. Oracle advises that after the total available memory, the second most influential factor is the proportion of the heap reserved for the young generation. By default, the minimum size of the young generation is 1310 MB, and the maximum size is unlimited.

- `-XX:NewRatio=ratio`: This sets the ratio between the young and old generation sizes; the default is set to 2.

- `-Xss size`: This sets the thread stack size, the default value depends on the platform:
  - Linux/ARM (32-bit): 320 KB
  - Linux/ARM (64-bit): 1024 KB
  - Linux/x64 (64-bit): 1024 KB
  - OS X (64-bit): 1024 KB
  - Oracle Solaris/i386 (32-bit): 320 KB
  - Oracle Solaris/x64 (64-bit): 1024 KB
  - Windows: The default value depends on virtual memory

    Each thread has a stack, so the stack size will limit the number of threads the JVM can have. If the stack size is too small, you can get the `java.lang.StackOverflowError` exception. However, making the stack size too big can exhaust the memory too, as each thread will allocate more memory than it needs.

- `-XX:MaxMetaspaceSize=size`: This sets the upper limit of the memory allocated for class metadata, not limited by default.

The tell-tell sign of a memory leak is the growing of the old generation causing the full GC to run more often. To investigate, you can use the JVM parameters that dump heap memory into a file:

- `-XX:+HeapDumpOnOutOfMemoryError`: This saves the Java heap into a file in the current directory when a `java.lang.OutOfMemoryError` exception is thrown. You can explicitly set the heap dump file path and name using the -

`XX:HeapDumpPath=path` option. By default, this option is disabled and the heap is not dumped when an `OutOfMemoryError` exception is thrown.

- `-XX:HeapDumpPath=path`: This sets the path and filename for writing the heap dump provided by the heap profiler (`hprof`) when the `-XX:+HeapDumpOnOutOfMemoryError` parameter is set. By default, the file is created in the current working directory, and it is named `java_pidpid.hprof` where `pid` is the identifier of the process that caused the error.
- `-XX:OnOutOfMemoryError="< cmd args >;< cmd args >"`: This sets a custom command or a series of semicolon-separated commands to run when an `OutOfMemoryError` exception is first thrown. If the string contains spaces, then it must be enclosed in quotation marks. For an example of a command string, see the description of the `-XX:OnError` parameter.
- `-XX:+UseGCOverheadLimit`: This enables the use of a policy that limits the proportion of time spent by the JVM on GC before an `OutOfMemoryError` exception is thrown. This option is enabled by default, and the parallel GC will throw an `OutOfMemoryError` exception if more than 98% of the total time is spent on garbage collection and less than 2% of the heap is recovered. When the heap is small, this feature can be used to prevent applications from running for long periods of time with little or no progress. To disable this option, specify `-XX:-UseGCOverheadLimit`.

See also

Do refer to the other recipes of this chapter:

- Understanding the G1 garbage collector
- Unified logging for JVM
- Using the new diagnostic commands for the JVM
- Try with resources for better resource handling
- Stack walking for improved debugging

The Read-Evaluate-Print Loop (REPL) Using JShell

In this chapter, we will cover the following recipes:

- Getting familiar with REPL
- Navigating JShell and its commands
- Evaluating code snippets
- Object-oriented programming in JShell
- Saving and restoring the JShell command history
- Using the JShell Java API

Introduction

REPL stands for the **Read-Evaluate-Print Loop** and, as the name states, it reads the command entered on the command line, evaluates it, prints the result of evaluation, and continues this process on any command entered.

All the major languages, such as Ruby, Scala, Python, JavaScript, and Groovy, among others, have REPL tools. Java was missing the much-needed REPL. If we had to try out some sample code, say using `SimpleDateFormat` to parse a string, we had to write a complete program with all the ceremonies, including creating a class, adding a main method, and then the single line of code we want to experiment. Then, we have to compile and run the code. These ceremonies make it harder to experiment and learn the features of the language.

With a REPL, you can type only the line of code that you are interested in experimenting with and you would have immediate feedback on whether the expression is syntactically correct and gives the desired results. REPL is a very powerful tool, especially for people coming to the language for the first time. Suppose you want to show how to print *Hello World* in Java; for this, you'd have to start writing the class definition, then the `public static void main(String [] args)` method, and by the end of it, you would have explained or tried to explain a lot of concepts that would otherwise be difficult for a newbie to comprehend.

Anyways, with Java 9, Java developers can now stop cribbing about the absence of a REPL tool. A new REPL called JShell is being bundled with the JDK installation. So, we can now proudly write *Hello World* as our first *Hello World* code.

In this chapter, we will explore the features of JShell and write code that will truly amaze us and appreciate the power of REPL. We will also see how we can create our own REPLs using the JShell Java API.

Getting familiar with REPL

In this recipe, we will look at a few basic operations to help us get familiarize us with the JShell tool.

Getting ready

Make sure you have the latest JDK 9 version installed, which has JShell.

How to do it...

1. By now, you should have `%JAVA_HOME%/bin` (on Windows) or `$JAVA_HOME/bin` (on Linux) added to your `PATH` variable. If not, then please visit the recipe, *Installing JDK 9 on Windows and setting up the PATH variable* and *Installing JDK 9 on Linux (Ubuntu, x64) and configuring the PATH variable* in `Chapter 1`, *Installation and a Sneak Peek into Java 9*.
2. In your command prompt, type `jshell` and press enter.
3. You will see a message and then a `jshell>` prompt:

```
C:\Users\Mohamed>jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro
```

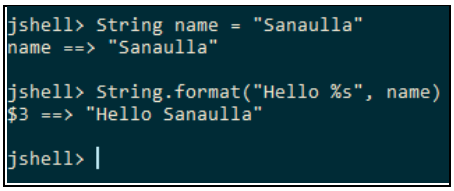
4. Forward slash (`/`), followed by the JShell-supported commands, help you in interacting with JShell. Just like we try `/help intro` to get the following:

```
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
```

5. Let's print a `Hello World` message:

```
jshell> "Hello World"
$1 ==> "Hello World"
```

6. Let's print a customized `Hello World` message:



7. You can navigate through the executed commands using the up and down arrow keys.

How it works...

The code snippets entered at the `jshell` prompt are wrapped with just enough code to execute them. So, variable, method and class declarations get wrapped within a class and expressions get wrapped within a method which is in turn wrapped within the class. Other things such as imports and class definitions remain as is because they are top-level entities that is, wrapping a class definition within another class is not required as a class definition is a top level entity and can exist by itself. Similarly, in Java, import statements can occur by themselves and they occur outside of a class declaration and hence need not be wrapped inside a class.

In the subsequent recipes, we will see how to define a method, import additional packages, define classes, and so on.

In the preceding recipe, we saw `$1 ==> "Hello World"`. If we have some value without any variable associated with it, then `jshell` gives it a variable name, such as `$1`, `$2`, and so on.

Navigating JShell and its commands

In order to leverage a tool, we need to be familiar with how to use it, the commands it provides, and the various shortcut keys that we can use to be productive. In this recipe, we will look at the different ways we can navigate through JShell and also at the different keyboard shortcuts it provides to be productive while using it.

How to do it...

1. Spawn the JShell by typing `jshell` in the command prompt, and you will be greeted with a welcome message containing the instructions to get started.
2. Type `/help intro` to get a brief introduction to JShell:

```
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
```

3. Type `/help` to get a list of the supported commands:

```
jshell> /help
|  Type a Java language expression, statement, or declaration.
|  Or type one of the following commands:
|  /list [<name or id>|-all|-start]
|       list the source you have typed
|  /edit <name or id>
|       edit a source entry referenced by name or id
|  /drop <name or id>
|       delete a source entry referenced by name or id
|  /save [-all|-history|-start] <file>
|       Save snippet source to a file.
|  /open <file>
|       open a file as source input
```

4. To get more information about a command, type `/help <command>`. For example, to get information about `/edit`, type `/help /edit`:

```
jshell> /help /edit
|
|  /edit
|
|  Edit a snippet or snippets of source in an external editor.
|  The editor to use is set with /set editor.
|  If no editor has been set, a simple editor will be launched.
|
|  /edit <name>
|       Edit the snippet or snippets with the specified name (preference for active snippets)
|
|  /edit <id>
|       Edit the snippet with the specified snippet id
|
|  /edit
|       Edit the currently active snippets of code that you typed or read with /open
```

5. There is autocompletion support in JShell. This makes Java developers feel at home. You can invoke autocompletion using the *Tab* key:

```
jshell> System.out.
append(         checkError()    close()         equals(         flush()
format(         getClass()      hashCode()      notify()        notifyAll()
print(          printf(         println(        toString()      wait(
write(

jshell> System.out.
append(         checkError()    close()         equals(         flush()
format(         getClass()      hashCode()      notify()        notifyAll()
print(          printf(         println(        toString()      wait(
write(

jshell> System.out.print
print(     printf(    println(

jshell> System.out.println(
println(

jshell> System.out.println(String.format(
format(

jshell> System.out.println(String.
CASE_INSENSITIVE_ORDER   class                    copyValueOf(
format(                  join(                   valueOf(

jshell> System.out.println(String.format
```

6. You can use `/!` to execute a previously executed command and `/line_number` to re-execute an expression at the line number.
7. To navigate the cursor through the command line, use *Ctrl* + *A* to reach the beginning of the line and *Ctrl* + *E* to reach the end of the line.

Evaluating code snippets

In this recipe, we will look at executing the following code snippets:

- Import statements
- Class declarations
- Interface declarations
- Method declarations
- Field declarations
- Statements

How to do it...

1. Open the command prompt and launch JShell.
2. By default, JShell imports a few libraries. We can check that by issuing the `/imports` command:

```
jshell> /imports
|    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.*
```

3. Let's import `java.text.SimpleDateForm` by issuing the `import java.text.SimpleDateFormat` command. This imports the `SimpleDateFormat` class.
4. Let's declare an `Employee` class. We will issue one statement in each line so that it's an incomplete statement, and we'll proceed in the same way as we do in any ordinary editor. The following illustration will clarify this:

```
class Employee{
  private String empId;
  public String getEmpId() {
    return empId;
  }
  public void setEmpId ( String empId ) {
    this.empId = empId;
  }
}
```

```
jshell> class Employee {
   ...>     private String empId;
   ...>     public String getEmpId() {
   ...>         return empId;
   ...>     }
   ...>     public void setEmpId (String empId ) {
   ...>         this.empId = empId;
   ...>     }
   ...> }
|  created class Employee
```

5. Let's declare an `Employability` interface, which defines a method, `employable()`, as shown in the following code snippet:

```
interface Employability {
  public boolean employable();
}
```

The preceding interface when created via the `jshell` is shown below:

```
jshell> interface Employability {
   ...>     public boolean employable();
   ...> }
|  created interface Employability
```

6. Let's declare a `newEmployee(String empId)` method, which constructs an `Employee` object with the given `empId`:

```
public Employee newEmployee(String empId ) {
  Employee emp = new Employee();
  emp.setEmpId(empId);
  return emp;
}
```

The preceding method defined in JShell is shown below:

```
jshell> public Employee newEmployee(String empId) {
   ...>     Employee emp = new Employee();
   ...>     emp.setEmpId(empId);
   ...>     return emp;
   ...> }
```

7. We will use the method defined in the previous step to create a statement declaring an `Employee` variable:

```
Employee e = newEmployee("1234");
```

The above statement and its output when executed from within JShell are shown below. The snippet `e.get + Tab` key generates auto-completion as supported by the IDEs.

```
jshell> Employee e = newEmployee("1234");
e ==> Employee@5ec0a365

jshell> e.get
getClass()   getEmpId()

jshell> e.getEmpId();
$18 ==> "1234"
```

There's more...

We can invoke an undefined method. Take a look at the following example:

```
public void newMethod(){
   System.out.println("New  Method");
   undefinedMethod();
}
```

```
jshell> public void newMethod(){
   ...>   System.out.println("New Method");
   ...>   undefinedMethod();
   ...> }
|  created method newMethod(), however, it cannot be invoked until method undefinedMethod() is declared

jshell> newMethod()
|  attempted to call method newMethod() which cannot be invoked until method undefinedMethod() is declared
```

However, the method cannot be invoked before the method being used has been defined:

```
public void undefinedMethod(){
   System.out.println("Now defined");
}
```

```
jshell> public void undefinedMethod(){
   ...>   System.out.println("Now defined");
   ...> }
|  created method undefinedMethod()

jshell> newMethod()
New Method
Now defined
```

We can invoke `newMethod()` only after we have defined `undefinedMethod()`.

Object-oriented programming in JShell

In this recipe, we will make use of predefined Java class definition files and import them into JShell. Then, we will play around with those classes in the JShell.

How to do it...

1. The class definition files we will use in this recipe are available at the location, `chp13/4_oo_programming`, in the code downloads for this book.
2. There are three class definition files: `Engine.java`, `Dimensions.java`, and `Car.java`.
3. Navigate to the directory where these three class definition files are available.
4. The `/open` command allows loading the code from within a file.
5. We will load the `Engine` class definition and create an `Engine` object.

```
jshell> /open Engine.java

jshell> Engine e = new Engine("Petrol", 4, 1400)
e ==> Engine@28d25987
```

6. Next, we will load the `Dimensions` class definition and create a `Dimensions` object:

```
jshell> /open Dimensions.java

jshell> Dimensions d = new Dimensions(
Dimensions(

jshell> Dimensions d = new Dimensions(4370, 1720, 1455)
d ==> Dimensions@61832929
```

7. Finally, we will load the `Car` class definition and create a `Car` object:

```
shell> /open Car.java

shell> Car c = new Car(e, d, "Kia", "Rio", 2017)
 ==> Car@26653222
```

Saving and restoring the JShell command history

We will want to try out some code snippets in `jshell` as a means to explain Java programming to someone who is new to it. Moreover, some form of record of what code snippets were executed will be useful for the person who is learning the language.

In this recipe, we will execute a few code snippets and save them into a file. We will then load the code snippets from the saved file.

How to do it...

1. Let's execute a series of code snippets, as follows:

```
"Hello World"
String msg = "Hello, %s. Good Morning"
System.out.println(String.format(msg, "Friend"))
int someInt = 10
boolean someBool = false
if ( someBool ) {
  System.out.println("True block executed");
}
if ( someBool ) {
  System.out.println("True block executed");
}else{
  System.out.println("False block executed");
}
for ( int i = 0; i < 10; i++ ){
  System.out.println("I is : " + i );
}
```

```
jshell> "Hello World"
$1 ==> "Hello World"

jshell> String msg = "Hello, %s. Good Morning"
msg ==> "Hello, %s. Good Morning"

jshell> System.out.println(String.format(msg, "Friend"))
Hello, Friend. Good Morning

jshell> int someInt = 10
someInt ==> 10

jshell> boolean someBool = false
someBool ==> false

jshell> if ( someBool ) {
   ...> System.out.println("True block executed");
   ...> }

jshell> if ( someBool ) {
   ...> System.out.println("True block executed");
   ...> }else{
   ...> System.out.println("False block executed");
   ...> }
False block executed

jshell> for ( int i = 0; i < 10; i++){
   ...> System.out.println("I is : " + i);
   ...> }
I is : 0
I is : 1
I is : 2
I is : 3
I is : 4
I is : 5
I is : 6
I is : 7
I is : 8
I is : 9
```

2. Save the code snippets executed into a file called `history` using the `/save history` command.
3. Exit the shell using `/exit` and list the files in the directory by using `dir` or `ls`, depending on the OS. There will be a `history` file in the listing.
4. Open `jshell` and check for the history of code snippets executed using `/list`. You will see that there are no code snippets executed.
5. Load the `history` file using `/open history` and then check for the history of the code snippets executed using `/list`. You will see all the previous code snippets being executed and added to the history:

```
jshell> /list

jshell> /open history
Hello, Friend. Good Morning
False block executed
I is : 0
I is : 1
I is : 2
I is : 3
I is : 4
I is : 5
I is : 6
I is : 7
I is : 8
I is : 9

jshell> /list

   1 : "Hello World"
   2 : String msg = "Hello, %s. Good Morning";
   3 : System.out.println(String.format(msg, "Friend"));
   4 : int someInt = 10;
   5 : boolean someBool = false;
   6 : if ( someBool ) {
       System.out.println("True block executed");
       }
   7 : if ( someBool ) {
       System.out.println("True block executed");
       }else{
       System.out.println("False block executed");
       }
   8 : for ( int i = 0; i < 10; i++){
       System.out.println("I is : " + i);
       }
```

Using the JShell Java API

JDK 9 provides the Java API for creating tools such as `jshell` for evaluating Java code snippets. This Java API is present in the `jdk.jshell` module (http://cr.openjdk.java.net/~rfield/arch/doc/jdk/jshell/package-summary.html). So, if you want to use the API in your application, then you need to declare a dependency on the `jdk.jshell` module.

In this recipe, we will use the JShell JDK API to evaluate simple code snippets, and you'll also see different APIs to get the state of JShell. The idea is not to recreate JShell but to show how to make use of its JDK API.



For this recipe, we will not be using JShell; instead, we will follow the usual way of compiling using `javac` and running using `java`.

How to do it...

1. Our module will depend on the `jdk.jshell` module. So, the module definition will look like the following:

```
module jshell{
  requires jdk.jshell;
}
```

2. Let's create an instance of the `jdk.jshell.JShell` class by using its `create()` method or the builder API in `jdk.jshell.JShell.Builder`:

```
JShell myShell = JShell.create();
```

3. Let's read the code snippet from `System.in` using `java.util.Scanner`:

```
try(Scanner reader = new Scanner(System.in)){
  while(true){
    String snippet = reader.nextLine();
    if ( "EXIT".equals(snippet)){
      break;
    }
    //TODO: Code here for evaluating the snippet using JShell API
  }
}
```

4. We will use the `jdk.jshell.JShell#eval(String snippet)` method to evaluate the input. Evaluation will result in a list of `jdk.jshell.SnippetEvent`, which contains the status and output of evaluation. The TODO in the preceding code snippet will be replaced by the following lines:

```
List<SnippetEvent> events = myShell.eval(snippet);
events.stream().forEach(se -> {
  System.out.print("Evaluation status: " + se.status());
  System.out.println(" Evaluation result: " + se.value());
});
```

5. After the evaluation is completed, we will print the snippets processed by using the `jdk.jshell.JShell.snippets()` method, which will return `Stream` of `Snippet` processed.

```
System.out.println("Snippets processed: ");
myShell.snippets().forEach(s -> {
```

```
        String msg = String.format("%s -> %s", s.kind(), s.source());
        System.out.println(msg);
      });
```

6. Similarly, we can print the active method and variables as follows:

```
      System.out.println("Methods: ");
      myShell.methods().forEach(m ->
        System.out.println(m.name() + " " + m.signature()));

      System.out.println("Variables: ");
      myShell.variables().forEach(v ->
        System.out.println(v.typeName() + " " + v.name()));
```

7. Before the application exits, we close the `JShell` instance by invoking its `close()` method:

```
      myShell.close();
```

The code for this recipe can be found at the location, `chp13/6_jshell_api`. You can run the sample by using the `run.bat` or `run.sh` scripts available in the same directory. The sample execution and output is shown here:

```
Welcome to JShell Java API Demo
Please Enter a Snippet. Enter EXIT to exit:
int i = 10;
Evaluation status: VALID Evaluation result: 10
void test() { System.out.println("Test called"); }
Evaluation status: VALID Evaluation result: null
int sum(int a, int b){ return a + b; }
Evaluation status: VALID Evaluation result: null
sum(4,5)
Evaluation status: VALID Evaluation result: 9
EXIT
Snippets processed:
VAR -> int i = 10;
METHOD -> void test() { System.out.println("Test called"); }
METHOD -> int sum(int a, int b){ return a + b; }
VAR -> sum(4,5)
Methods:
test ()void
sum (int,int)int
Variables:
int i
int $1
'Bye!!'
```

How it works...

The central class in the API is the `jdk.jshell.JShell` class. This class is the evaluation state engine, whose state is modified with every evaluation of the snippet. As we saw earlier, the snippets are evaluated using the `eval(String snippet)` method. We can even drop the previously evaluated snippet using the `drop(Snippet snippet)` method. Both these methods result in a change of the internal state maintained by `jdk.jshell.JShell`.

The code snippets passed to the `JShell` evaluation engine are categorized as follows:

- **Erroneous**: Syntactically incorrect input
- **Expressions**: An input which might or might not result in some output
- **Import**: An import statement
- **Method**: A method declaration
- **Statement**: A statement
- **Type declaration**: A type, that is, class/interface declaration
- **Variable declaration**: A variable declaration

All these categories are captured in the `jdk.jshell.Snippet.Kind` enum.

We also saw different APIs to get the evaluated snippets, created methods, variable declarations, and other specific snippet types executed. Each snippet type is backed by a class extending the `jdk.jshell.Snippet` class.

Scripting Using Oracle Nashorn

In this chapter, we will cover the following recipes:

- Using the jjs command-line tool
- Embedding the Oracle Nashorn engine
- Invoking Java from Oracle Nashorn
- Using the ES6 features implemented in Oracle Nashorn

Introduction

Oracle Nashorn is the JavaScript engine developed for the Java platform. This was introduced in Java 8. Prior to Nashorn, the JavaScript engine for the Java platform was based on the Mozilla Rhino JavaScript engine. The Oracle Nashorn engine leverages the `invokedynamic` support introduced in Java 8 for better runtime performance, and also provides better compliance with the ECMAScript specification.

Oracle Nashorn supports JavaScript code execution in a standalone mode using the `jjs` tool, as well as embedded in Java using its embedded scripting engine. In this chapter, we will look at executing the JavaScript code from Java and invoking the JavaScript function from Java, and vice versa, including accessing Java types from JavaScript. We will also look at using the command-line tool, `jjs`, for executing the JavaScript code.

Throughout the rest of the chapter, we will use the term ES6 to refer to ECMAScript 6.

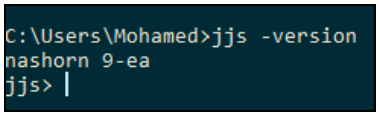
Using the jjs command-line tool

The `jjs` command-line tool supports the execution of JavaScript code files as well as an interactive execution of JavaScript code snippets, as supported by other JavaScript shells, such as `node.js`. It uses Oracle Nashorn, a next generation JavaScript engine for JVM to provide this support. In addition to the JavaScript code, `jjs` supports the execution of shell commands, thereby allowing us to create shell script utilities in JavaScript.

In this recipe, we will look at executing JavaScript code files via `jjs`, as well as the interactive execution of code snippets.

Getting ready

First, verify whether the `jjs` tool is available by issuing the command, `jjs -version`. This will print the version as `nashorn 9-ea` and enter it into the shell, as shown in the following image:



We can even get more specific version information using `jjs -fv`, which prints the version as `nashorn full version 9-ea+169`.

The JavaScript code files used in this recipe are available at the location, `chp14/1_jjs_demo`.

How to do it...

1. Let's use `jjs` to execute the script, `$ jjs hellojjs.js`, which gives this output: `Hello via JJS using Nashorn.`
2. Let's now try this with ECMAScript 6 features of using `Set`, `Map`, and template strings. A template string supports building a `String` with placeholders for dynamic values. A placeholder is identified by `${variable}` and the complete `String` is embedded within ``. We run this script using the `jjs --language=es6 using_map_set_demo.js` command. By default, `jjs` runs in `es5` mode and we enable it to run in `es6` by giving this option, as shown in the following screenshot:

```
G:\java9\java9-samples\chp14\1_jjs_demo>jjs --language=es6 using_map_set_demo.js
Set size is 3
Set Elements:
item1
item2
item3
Set has item1? true
Set has item4? false
Map has 3 entries
```

3. Now, let's use the `jjs` tool interactively. Run `$ jjs --language=es6` on the Command Prompt to launch the shell and execute a few JavaScript code snippets, as follows:

```
var numbers = [1,2,3,4,5];
var twiceTheValue = numbers.map(v => v * 2 ) ;
print(twiceTheValue);
var name = "Sanaulla";
print(`Hello Mr. ${name}`);
const pi = 3.14
pi = 4
for ( var n of twiceTheValue ) {
  print(n);
}
```

The following will be printed on the screen:

```
G:\java9\java9-samples\chp14\1_jjs_demo>jjs --language=es6
jjs> var numbers = [1,2,3,4,5];
jjs> var twiceTheValue = numbers.map(v => v * 2 ) ;
jjs> print(twiceTheValue)
2,4,6,8,10
jjs> var name = "Sanaulla";
jjs> print(`Hello Mr. ${name}`)
Hello Mr. Sanaulla
jjs> const pi = 3.14
jjs> pi = 4
<shell>:1 TypeError: Assignment to constant "pi"
jjs> for ( var n of twiceTheValue ) {
...> print(n);
...> }
2
4
6
8
10
jjs>
```

There's more...

The shell scripting mode can be enabled in `jjs` using the `-scripting` command. So, one can embed Shell/Batch commands within the JavaScript code, as shown here:

```
var files = $EXEC("dir").split("\n");
for( let file of files){
  print(file);
}
```

If you are using ES5 as the language for `jjs`, then you can replace `$EXEC("dir")` with `` `dir` ``. But in ES6, `` `` `` are used for representing template strings. The preceding script can be executed using `jjs`, as shown here:

```
$ jjs -scripting=true --language=es6 embedded_shell_command.js
embedded_shell_command.js  hellojjs.js  using_map_set_demo.js
```

There are two more variables available, `$ARG` and `$ENV`, which can be used to access the arguments passed to the script and the environment variables, respectively.

Embedding the Oracle Nashorn engine

In this recipe, we will look at embedding the Nashorn JavaScript engine in the Java code and execute different JavaScript code snippets, functions, and JavaScript source files.

Getting ready

You should have JDK 9 installed, as we will be using a few ES6 JavaScript language features with the Nashorn engine.

How to do it...

1. First, we get an instance of `ScriptEngine` with the ES6 language features enabled:

```
NashornScriptEngineFactory factory =
       new NashornScriptEngineFactory();
ScriptEngine engine = factory.getScriptEngine("--language=es6");
```

2. Let's define a JavaScript function to find the sum of two numbers:

```
engine.eval("function sum(a, b) { return a + b; }");
```

3. Let's invoke the function defined in the previous step:

```
System.out.println(engine.eval("sum(1, 2);"));
```

4. Then we will look at the template string support:

```
engine.eval("let name = 'Sanaulla'");
System.out.println(engine.eval("print(`Hello Mr. ${name}`)"));
```

5. We will use the new `Set` construct in ES6 and the new `for` loop to print the `Set` elements:

```
engine.eval("var s = new Set();
s.add(1).add(2).add(3).add(4).add(5).add(6);");
System.out.println("Set elements");
engine.eval("for (let e of s) { print(e); }");
```

6. Finally, we will look at loading the JavaScript source file and executing the methods defined in it:

```
engine.eval(new FileReader("src/embedded.nashorn/com/packt
                           /embeddable.js"));
int difference = (int)engine.eval("difference(1, 2);");
System.out.println("Difference between 1, 2 is: " + difference);
```

The complete code for this can be found at the location, `chp14/2_embedded_nashorn`.

The output after executing the sample will be as follows:

```
3
Hello Mr. Sanaulla
null
Set elements
1
2
3
4
5
6
Difference between 1, 2 is: -1
```

Invoking Java from Oracle Nashorn

In this recipe, we will look at calling Java APIs from the JavaScript code, including using the Java types, and dealing with package and class imports. There is a greater potential in combining the vastness of the Java API and the dynamic nature of JavaScript leveraged by the Oracle Nashorn JavaScript engine. We will look at creating a purely JavaScript code, which uses the Java APIs, and we'll use the `jjs` tool to execute this.

We will also look at creating a Swing-based application purely in JavaScript.

How to do it...

1. Let's create a `List` of numbers using the `Arrays.asList` API:

```
var numbers = java.util.Arrays.asList(12,4,5,67,34,567,32);
```

2. Now, compute the maximum number in the list:

```
var max = java.util.Collections.max(numbers);
```

3. We can print `max` using the JavaScript `print()` method, and we can use template strings:

```
print(`Max of ${numbers} is ${max}`);
```

4. Let's run the script created using `jjs`:

```
jjs --language=es6 java_from_javascript.js
```

5. Now, let's import the `java.util` package:

```
var javaUtils = new JavaImporter(java.util);
```

6. Let's use the imported package to print today's date:

```
with(javaUtils){
  var date = new Date();
  print(`Todays date is ${date}`);
}
```

7. Let's create an alias for a Java type using the `Java.type` API:

```
var jSet = Java.type('java.util.HashSet');
```

8. Use the alias to create a set, add a few elements, and print it:

```
var mySet = new jSet();
mySet.add(1);
mySet.add(4);
print(`My set is ${mySet}`);
```

We get the following output:

```
Max of [12, 4, 5, 67, 34, 567, 32] is 567
Todays date is Wed May 24 2017 00:43:35 GMT+0300 (AST)
My set is [1, 4]
```

The code for this script file can be found at the location, `chp14/3_java_from_nashorn/java_from_javascript.js`.

How it works...

The Java types and their APIs can be accessed from the JavaScript code by using their fully qualified name, as we saw in the previous section while creating the list of numbers using `java.util.Arrays.asList()` and finding the maximum using `java.util.Collections.max()`.

If we want to skip specifying the package name along with the class name, we can make use of `JavaImporter` to import the packages and use the `with` clause to wrap the code, which uses the classes from the imported package within it, as follows:

```
var javaUtils = new JavaImporter(java.util);
with(javaUtils){
  var date = new Date();
  print(`Todays date is ${date}`);
}
```

The other feature we saw was creating a type alias for the Java type by using `Java.type(<fully qualified class name>)`, as done in the example:

```
var jSet = Java.type('java.util.HashSet');
```

The type has to be an implementation class if you are creating objects using the alias:

```
var mySet = new jSet();
```

There's more...

Let's create a script to create a simple Swing GUI with a button and an event handler for the button. We will also look at how we leverage imports and implement interfaces using an anonymous inner class approach.

First, we'll create a new `JavaImporter` object with the required Java packages:

```
var javaGui = new JavaImporter(javax.swing, java.awt, java.awt.event);
```

We use the `with(obj){}` clause to wrap all the statements using the required imports:

```
with(javaGui){
  //other statements
}
```

Next, we create `JButton` and provide `ActionListener` to listen to its click events:

```
var button = new JButton("My Button");
button.addActionListener(new ActionListener({
  actionPerformed: function(e){
    print("Button clicked");
  }
}));
```

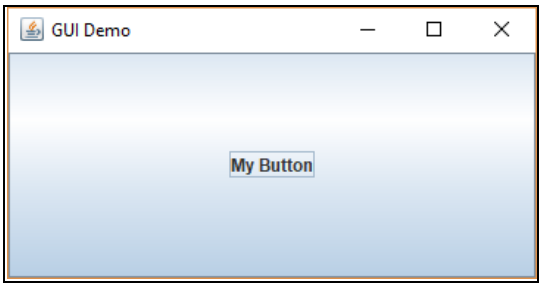
Then, we create `JFrame` to render the GUI with its components:

```
var frame = new JFrame("GUI Demo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(button, BorderLayout.CENTER);
frame.pack();
frame.setVisible(true);
```

The complete code for this can be found in `chp14/3_java_from_nashorn/gui_from_javascript.js`. Let's run the script using `jjs`:

```
jjs --language=es6 java_from_javascript.js
```

We can see a small GUI, as shown in the following screenshot:



To exit the program, we will have to use *Ctrl* + *C* to stop the process because `setDefaultCloseOperation` works when running in Java. The other option is to override the JFrame's `close` operation to exit from the program.

Using the ES6 features implemented in Oracle Nashorn

In this recipe, we will look at some of the ES6 features implemented in the Oracle Nashorn JavaScript engine. For this, we will create a JavaScript file and use `jjs` to execute the same. Remember to use `jjs` in the ES6 mode, which can be enabled by passing the `--language=es6` command-line option.

How to do it...

1. Template strings are strings with variable placeholders, thereby allowing the creation of dynamic texts. The strings have to be embedded within the symbols ``:

```
var name = "Sanaulla";
print(`My name is ${name}`);
```

2. Any variable in JavaScript has a global scope. ES6 introduced the block scope, which can be declared using the `let` keyword. Constants can now be defined using the `const` keyword, as shown in the following code snippet:

```
const pi = 3.14;
var language = "Java";
function hello(){
  let name = "Mohamed";
  language = "Javascript";
  print(`From hello(). Hello ${name}`);
  print(`From hello(). Language is ${language}`);
}
print(`Before hello(). Language is ${language}`);
hello();
print(`After hello(). Language is ${language}`);
print(`After hello(). Hello ${name}`);
//pi = 4.5;
//above will be error because pi is defined as a constant
```

3. Let's look at the new iteration construct, `for ... of`:

```
let numbers = [2,4,6,8,10,12];
for ( const number of numbers ){
  print(number);
}
```

4. Let's look at creating sets and maps using the new `Set` and `Map` classes:

```
var set = new Set();
set.add("elem 1").add("elem 2").add("elem 3").add("elem 1");
print(`Set ${set} has ${set.size} elements`);
var map = new Map();
map.set(1, "elem 1");
map.set(2, "elem 2");
map.set(3, "elem 3");
print(`Map has 1? ${map.has(1)}`);
```

5. Let's look at arrow functions. These are similar to the lambda expression concept that we have from Java 8 onward. Arrow functions are of the form, `(parameters ) => {function body }`:

```
numbers = [1,2,3,4,5,6,7,8,9,10];
var evenNumbers = numbers.filter(n => n % 2 == 0);
print(`Even numbers: ${evenNumbers}`);
```

The complete code for this can be found in the file, `chp14/4_es6_features/es6_features.js`. The output after executing the complete script using the `jjs --language=es6 es6_features.js` command is as follows:

```
******* Template strings *******
My name is Sanaulla
******* let, const and block scope *******
Before hello(). Language is Java
From hello(). Hello Mohamed
From hello(). Language is Javascript
After hello(). Language is Javascript
After hello(). Hello Sanaulla
******* Using for ... of loops *******
2
4
6
8
10
12
******* Map, Set *******
Set [object Set] has 3 elements
Map has 1? true
******* Arrow functions *******
Even numbers: 2,4,6,8,10
```

Testing

This chapter shows how to unit test your APIs before they are integrated with other components. Sometimes, we would have to stub dependencies with some dummy data, and this can be done by mocking the dependencies. We will show you how to do this using a mocking library. We will also show you how to write fixtures to populate test data and then how you can test the behavior of your application by integrating different APIs and testing them together. We will cover the following recipes:

- Unit testing of an API using JUnit
- Unit testing by mocking dependencies
- Using fixtures to populate data for testing
- Behavioral testing

Introduction

Well-tested code provides a peace of mind to the developer. If you get a feeling that writing a test for the new method you are developing is too much of an overhead, then you usually don't get it right the first time.. This is because you have to test your method anyway. Doing this in the context of the application only requires time to set things up (especially if you are trying to test all the possible input and conditions). Then, if the method changes, you need to redo the setup again. And you do it manually. You can avoid this by creating an automated test at the same time you were developing the new method (we assume not-too complex code, of course; setters and getters do not count). This would save you time in the long run.

Why do we get this feeling of overhead sometimes? It's probably because we are not prepared psychologically. When we think about how long it would take to add the new functionality, we often forget to include the time needed for writing the test. It is even worse when we forget this while providing an estimate to a manager. Often, we shy away from giving a higher estimate because we do not want to look not perception we have of not being very knowledgeable or skilled enough. Whatever the reason, it happens. Only after years of experience, we learn to include tests in our estimates and earn enough respect and clout to be able to assert publicly that doing things right requires more time up front, but saves much more time in the long run. Besides, doing it right leads to a better quality of the result with far less stress, which means a better quality of life overall.

If you are still not convinced, make note of the date when you read this and check back every year until this advice becomes obvious to you. Then, please share your experiences with others. This is how humanity makes progress, by passing knowledge from one generation to the next.

However, if you would like to learn how to write tests that will help you produce good quality Java code, this chapter is for you. Methodologically, this is applicable to other languages and professions too. This chapter is

written primarily for Java developers and assumes the authorship of tested code. Another assumption is that testing happens during the early stages of code writing so that code weaknesses discovered during the testing can be fixed immediately. Writing automated tests early is the best time to do it for two other reasons:

- You can easily restructure your code to make it more testable
- It saves you time by eliminating guesswork, which in turn makes your development process more productive

Another good moment to write a test (or enhance the existing one) is when a defect is discovered in production. It helps your investigation of the root cause if you recreate the problem and demonstrate it in a failed test and then show how the issue disappears (and the test does not fail anymore) in the new version of the code.

Unit testing of an API using JUnit

According to Wikipedia, *A research survey performed in 2013 across 10,000 Java projects hosted on GitHub found that JUnit, (in a tie with slf4j-api), was the most commonly included external library. Each library was used by 30.7% of projects.* JUnit is a testing framework--one of a family of unit testing frameworks collectively known as xUnit that originated with SUnit. It is linked as a JAR at compile time and resides (since JUnit 4) in the `org.junit` package.

Here's an excerpt from another article on Wikipedia, *In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.* We've found the last part--a unit as an individual method--the most useful in practice. It will be the basis for the examples of the recipes of this chapter.

Getting ready

At the time of this writing, the latest stable version of JUnit is 4.12, which can be used by adding the following Maven dependency to the `pom.xml` project level:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

After this, you can write your first JUnit test. Let's assume you have the `Vehicle` class created in the `src/main/java/com/packt/cookbook.ch02_oop.a_classes` folder (this is exactly the code we discussed in `Chapter 2`, *Fast Track to OOP - Classes and Interfaces*, of this book):

```
package com.packt.cookbook.ch02_oop.a_classes;
public class Vehicle {
  private int weightPounds;
  private Engine engine;
  public Vehicle(int weightPounds, Engine engine) {
    this.weightPounds = weightPounds;
    if(engine == null){
      throw new RuntimeException("Engine value is not set.");
    }
    this.engine = engine;
  }
  protected double getSpeedMph(double timeSec){
    double v = 2.0*this.engine.getHorsePower()*746;
    v = v*timeSec*32.174/this.weightPounds;
    return Math.round(Math.sqrt(v)*0.68);
  }
}
```

Now you can create the `src/test/java/com/packt/cookbook.ch02_oop.a_classes` folder (notice the new folder tree that starts with `test`, created in parallel to the tree of `main`) and create a new file in it called `VehicleTest.java` that contains the `VehicleTest` class:

```
package com.packt.cookbook.ch02_oop.a_classes;
import org.junit.Test;
public class VehicleTest {
  @Test
  public void testGetSpeedMph(){
    System.out.println("Hello!" + " I am your first test method!");
  }
}
```

Run it using your favorite IDE or just with the `mvn test` command. You will see an output that will include the following:



Congratulations! You have created your first test class. It does not test anything yet, but it is an important setup--the overhead that is necessary for doing things the right way. In the next section, we will start with the actual testing.

How to do it...

Let's look at the `Vehicle` class closer. Testing the getters would be of little value, but we can still do it, making sure that the value passed to the constructor is returned by the corresponding getter. The exception in the constructor belongs to the must-test features as well as the `getSpeedMph()` method. There is also an object of the `Engine` class that has the `getHorsePower()` method. Can it return `null`? We should look in the `Engine` class too:

```
public class Engine {
  private int horsePower;
  public int getHorsePower() {
    return horsePower;
  }
  public void setHorsePower(int horsePower) {
    this.horsePower = horsePower;
  }
}
```

There is not much behavior in this class to test and it cannot return `null`. But returning a negative value is a definite possibility, which in turn can cause problems for the `Math.sqrt()` function of the `getSpeedMph()` method. Should we make sure that the horsepower value will never be negative? It depends on how limited is the method's usage and the source of the input data for it.

Similar considerations are applicable to the value of the `weightPounds` property of the `Vehicle` class. It can stop the application with `ArithmeticException` caused by the division by zero in the `getSpeedMph()` method.

However, in practice, there is little chance that the values of an engine's horsepower and vehicle weight will be negative or close to zero, so we will assume this and will not add these checks to the code.

Such analysis is the daily routine and the background thoughts of every developer, and that is the first step in the right direction. The second step is to capture all these thoughts and doubts in the unit tests and verify the assumptions.

Let's go back to the test class we have created and enhance it. As you may have probably noticed, the `@Test` annotation makes a certain method a test method. This means it will be run by your IDE or Maven every time you issue a command to run tests. The method can be named any way you like, but a best practice advises to indicate which method (of the `Vehicle` class, in this case) you are testing. So, the format usually looks like `test<methodname><scenario>`, where `scenario` indicates a particular test case: a happy path, a failure, or some other condition you would like to test. In our example, we do not use suffix as an indication that we are going to test the main functionality that is working successfully (without any errors or edge cases). Later, we will show examples of methods that test other scenarios.

In such a test method, you can call the application method you are testing, provide it with the data, and assert the result. You can create your own assertions (methods to compare the actual results with the expected ones) or you can use assertions provided by JUnit. To do the latter, just add `static` import:

```
import static org.junit.Assert.assertEquals;
```

If you use a modern IDE, you can type `import static org.junit.Assert` and see how many different assertions are available (or go to JUnit's API documentation and see it there). There is a dozen or more overloaded methods available: `assertArrayEquals()`, `assertEquals()`, `assertNotEquals()`, `assertNull()`, `assertNotNull()`, `assertSame()`, `assertNotSame()`, `assertFalse()`, `assertTrue()`, `assertThat()`, and `fail()`. It would be helpful if you spend a few minutes reading what these methods do. You can also guess their purpose by name. Here is an example of the usage of the `assertEquals()` method:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class VehicleTest {
  @Test
  public void testGetSpeedMph(){
    System.out.println("Hello!" + " I am your first test method!");
    assertEquals(4, "Hello".length());
  }
}
```

We compare the actual length of the word `Hello` and the expected length, that is, `4`. We know that the correct number would be `5`, but we would like the test to fail to demonstrate the failing behavior and advise how to read the failing test results (you don't need to read happy results, do you?). If you run this test, you'll get the following result:

```
Running com.packt.cookbook.ch02_oop.a_classes.VehicleTest
Hello! I am your first test method!
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.04 sec <<< FAILURE!
testGetSpeedMph(com.packt.cookbook.ch02_oop.a_classes.VehicleTest)  Time elapsed: 0.006 sec
java.lang.AssertionError: expected:<4> but was:<5>
```

You can see that the expected value was `4`, while the actual is `5`. Say, you switch the order like this:

```
assertEquals("Assert Hello length:","Hello".length(), 4);
```

The result of this will be as follows:

```
Running com.packt.cookbook.ch02_oop.a_classes.VehicleTest
Hello! I am your first test method!
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.041 sec <<< FAILURE!
testGetSpeedMph(com.packt.cookbook.ch02_oop.a_classes.VehicleTest)  Time elapsed: 0.007 sec
java.lang.AssertionError: expected:<5> but was:<4>
```

This is incorrect because `5` is the actual result, while `4` is the expected (although erroneous, for demonstration purposes only).



It is important to remember that in each of the asserting methods, the parameter with the expected value is located (in the signature of an assertion) ***before*** *the actual one.*

After the test is written, you will do something else, and months later, you will probably forget what each assertion actually evaluated. But it may well be that one day the test will fail (because you or somebody will change the application code). You will see the test method name, expected value, and the actual value, but you will have to dig through the code to figure which of the assertion failed (there are often several of them in each test method). You will probably be forced to add a debug statement and run the test several times in order to figure it out. To help you avoid this extra digging,

each of the JUnit assertions allows you to add a message that describes the particular assertion. For example, run this version of the test:

```
public class VehicleTest {
  @Test
  public void testGetSpeedMph(){
    System.out.println("Hello!" + " I am your first test method!");
    assertEquals("Assert Hello length:", 4, "Hello".length());
  }
}
```

If you do this, the result will be much more readable:

```
Running com.packt.cookbook.ch02_oop.a_classes.VehicleTest
Hello! I am your first test method!
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.04 sec <<< FAILURE!
testGetSpeedMph(com.packt.cookbook.ch02_oop.a_classes.VehicleTest)  Time elapsed: 0.006 sec
java.lang.AssertionError: Assert Hello length: expected:<4> but was:<5>
```

To complete this demonstration, we change the expected value to `5`:

```
assertEquals("Assert Hello length:", 5, "Hello".length());
```

This will be your test output:

```
Running com.packt.cookbook.ch02_oop.a_classes.VehicleTest
Hello! I am your first test method!
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.038 sec
```

How it works...

Equipped with some knowledge of the JUnit framework, we can now write a real test method for the main case: the calculation of the speed of a vehicle of certain weight with an engine of certain horsepower to determine where it can reach in a certain period of time. We take the formula we used for writing the code (provided originally by the domain expert) and calculate the expected value. For example, if the vehicle has an engine of 246 hp and weight of 4,000 lb, then in 10 sec, its speed can reach 117 mph. Since the speed is of the type `double`, we will use this assertion:

```
void assertEquals(String message, double expected,
                        double actual, double delta)
```

Here, `delta` is allowable precision (we decided that 1 percent is good enough). The resulting implementation of the `test` method will look as follows:

```
@Test
public void testGetSpeedMph(){
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;

  Engine engine = new Engine();
  engine.setHorsePower(engineHorsePower);

  Vehicle vehicle = new Vehicle(vehicleWeightPounds, engine);
  double speed = vehicle.getSpeedMph(timeSec);
  assertEquals("Assert vehicle (" + engineHorsePower
            + " hp, " + vehicleWeightPounds + " lb) speed in "
            + timeSec + " sec: ", 117, speed, 0.001 * speed);
}
```

If we run this test, the output will be as follows:

```
Running com.packt.cookbook.ch02_oop.a_classes.VehicleTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.081 sec
```

To make sure the test is working, we set the expected value to 119 mph (more than 1 percent different) and run the test again. The result will be as follows:

```
Running com.packt.cookbook.ch02_oop.a_classes.VehicleTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.075 sec <<< FAILURE!
testGetSpeedMph(com.packt.cookbook.ch02_oop.a_classes.VehicleTest)  Time elapsed: 0.045 sec  <<< FAILURE!
java.lang.AssertionError: Assert vehicle (246 hp, 4000 lb) speed in 10.0 sec:  expected:<119.0> but was:<117.0>
```

We change the expected value back to 117 and continue writing other test cases we discussed while analyzing the code.

Let's make sure that the exception is thrown when expected. Let's add another import:

```
import static org.junit.Assert.fail;
```

Then, write the following test:

```
@Test
public void testGetSpeedMphException(){
  int vehicleWeightPounds = 4000;
  Engine engine = null;
  try {
    Vehicle vehicle = new Vehicle(vehicleWeightPounds, engine);
    fail("Exception was not thrown");
  } catch (RuntimeException ex) {}
}
```

This test runs successfully too. To make sure that the test works correctly, we temporarily assign it with the following:

```
Engine engine = new Engine();
```

Then, we observe the output:

```
Running com.packt.cookbook.ch02_oop.a_classes.VehicleTest
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.074 sec <<< FAILURE!
testGetSpeedMphException(com.packt.cookbook.ch02_oop.a_classes.VehicleTest)  Time elapsed: 0.004 sec
java.lang.AssertionError: Exception was not thrown
```

This way, we get a level of confidence that we did not code something that is going to be correct always, no matter the code changes.

As you can see, the best way to write these tests is in the process of writing application code, so you can test the code as it grows in complexity. Otherwise, especially in more complex code, you might have problems debugging it after all of the code is written already.

There are quite a few other annotations and JUnit features that can be helpful to you, so please refer to the JUnit documentation for more in-depth understanding of all the framework capabilities.

See also

Refer to the following recipes in this chapter:

- Unit testing by mocking dependencies
- Using fixtures to populate data for testing
- Behavioral testing

Unit testing by mocking dependencies

Writing a unit test requires unit isolation. In case a method uses several other methods from different objects, there arises a need to limit the depth of testing so that each layer can be tested in isolation as a unit. This is when the need for mocking the lower level comes into focus.

Mocking can not only be done vertically, but also horizontally: at the same level, but already isolated from the underlying functionality. If a method is long and complicated, you might consider breaking it into several smaller methods so you can test only one of them while mocking the others. This is another advantage of unit testing code along with its development; it is easier to redesign code for better testability before it is hardened.

Getting ready

Mocking other methods and classes is straightforward. Coding to an interface (as described in `Chapter 2`, *Fast Track to OOP - Classes and Interfaces*) makes it much easier, although there are mocking frameworks that allow you to mock classes that do not implement any interface (we will see examples of such framework usage in the next section of this recipe). Also, using object and method factories helps you create test-specific implementations of such factories so they can generate objects with methods that return the expected hardcoded values.

For example, in `Chapter 4`, *Going Functional*, we introduced `FactoryTraffic`, which produced one or many objects of `TrafficUnit`. In a real system, this factory would draw data from some external system that has data about traffic for a certain geographic location and time of the day. Using the real system as the source would complicate the code setup for you when you run the examples. To get around this problem, we have mocked the data by generating them according to the distribution that somewhat resembles the real one: a bit more cars than trucks, the weight of the vehicle depending on the type of the car, the number of passengers and weight of the payload, and similar. What is important for such a simulation is that the range of values (min and max) should reflect those coming from the real system, so the application would be tested on the full range of possible real data.

The important constraint for mocking code is that it should not be too complicated. Otherwise, its maintenance would require an overhead that would either decrease the team productivity or decrease (if not abandon completely) the test coverage.

How to do it...

Here is how the mock of `FactoryTraffic` will look like:

```
public class FactoryTraffic {
  public static List<TrafficUnit> generateTraffic(int
    trafficUnitsNumber, Month month, DayOfWeek dayOfWeek,
    int hour, String country, String city, String trafficLight){
    List<TrafficUnit> tms = new ArrayList();
    for (int i = 0; i < trafficUnitsNumber; i++) {
      TrafficUnit trafficUnit =
        FactoryTraffic.getOneUnit(month, dayOfWeek,  hour, country,
                                  city, trafficLight);
        tms.add(trafficUnit);
    }
    return tms;
  }
}
```

It assembles a collection of `TrafficUnit` objects. In a real system, these objects would be created from the rows of the result of some database query, for example. But in our case, we just mock the result:

```
public static TrafficUnit getOneUnit(Month month,
              DayOfWeek dayOfWeek, int hour, String country,
              String city, String trafficLight) {
  double r0 = Math.random();
  VehicleType vehicleType = r0 < 0.4 ? VehicleType.CAR :
  (r0 > 0.6 ? VehicleType.TRUCK : VehicleType.CAB_CREW);
  double r1 = Math.random();
  double r2 = Math.random();
  double r3 = Math.random();
  return new TrafficModelImpl(vehicleType, gen(4,1),
             gen(3300,1000), gen(246,100), gen(4000,2000),
             (r1 > 0.5 ? RoadCondition.WET : RoadCondition.DRY),
             (r2 > 0.5 ? TireCondition.WORN : TireCondition.NEW),
             r1 > 0.5 ? ( r3 > 0.5 ? 63 : 50 ) : 63 );
}
```

As you can see, we used a random number generator to pick up the value from a range for each of the parameters. The range is in line with the ranges of the real data. This code is very simple and it does not require much maintenance, but it provides the application with the flow of data similar to the real one.

You can use another technique for testing the method that has some dependencies that you would like to isolate in order to get the predictable result. For example, let's revisit the `VechicleTest` class. Instead of creating a real `Engine` object, we can mock it using one of the mocking frameworks. In this case, we use Mockito. Here is the Maven dependency for it:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>2.7.13</version>
  <scope>test</scope>
</dependency>
```

The test method now looks like this (the two lines that were changed are highlighted):

```
@Test
public void testGetSpeedMph(){
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;

  Engine engine = Mockito.mock(Engine.class);
  Mockito.when(engine.getHorsePower()).thenReturn(engineHorsePower);

  Vehicle vehicle =  new Vehicle(vehicleWeightPounds, engine);
  double speed = vehicle.getSpeedMph(timeSec);
  assertEquals("Assert vehicle (" + engineHorsePower
              + " hp, " + vehicleWeightPounds + " lb) speed in "
              + timeSec + " sec: ", 117, speed, 0.001 * speed);
}
```

As you can see, we instruct the `mock` object to return a fixed value when the `getHorsePower()` method is called. We can even go as far as creating a mock object for the method we want to test:

```
Vehicle vehicleMock = Mockito.mock(Vehicle.class);
Mockito.when(vehicleMock.getSpeedMph(10)).thenReturn(30d);

double speed = vehicleMock.getSpeedMph(10);
System.out.println(speed);
```

So, it always returns the same value:

```
30.0
```

However, this would defeat the purpose of testing.

You can use another technique while testing a stream's pipeline methods. Let's assume we need to test the `trafficByLane()` method in the `TrafficDensity1` class (we are going to have `TrafficDensity2` and `TrafficDensity3`, too):

```
public class TrafficDensity1 {
  public Integer[] trafficByLane(Stream<TrafficUnit> stream,
  int trafficUnitsNumber, double timeSec,
  SpeedModel speedModel, double[] speedLimitByLane) {

    int lanesCount = speedLimitByLane.length;

    Map<Integer, Integer> trafficByLane = stream
      .limit(trafficUnitsNumber)
      .map(TrafficUnitWrapper::new)
      .map(tuw -> tuw.setSpeedModel(speedModel))
      .map(tuw -> tuw.calcSpeed(timeSec))
      .map(speed ->  countByLane(lanesCount, speedLimitByLane, speed))
      .collect(Collectors.groupingBy(CountByLane::getLane,
               Collectors.summingInt(CountByLane::getCount)));

    for(int i = 1; i <= lanesCount; i++){
      trafficByLane.putIfAbsent(i, 0);
    }
    return trafficByLane.values()
      .toArray(new Integer[lanesCount]);
  }

  private CountByLane countByLane(int lanesCount,
                 double[] speedLimit, double speed) {
    for(int i = 1; i <= lanesCount; i++){
      if(speed <= speedLimit[i - 1]){
        return new CountByLane(1, i);
      }
    }
    return new CountByLane(1, lanesCount);
  }
}
```

It uses two support classes:

```
private class CountByLane{
  int count, lane;
  private CountByLane(int count, int lane){
    this.count = count;
    this.lane = lane;
  }
  public int getLane() { return lane; }
```

```
  public int getCount() { return count; }
}
```

It also uses the following:

```
private static class TrafficUnitWrapper {
  private Vehicle vehicle;
  private TrafficUnit trafficUnit;
  public TrafficUnitWrapper(TrafficUnit trafficUnit){
    this.vehicle = FactoryVehicle.build(trafficUnit);
    this.trafficUnit = trafficUnit;
  }
  public TrafficUnitWrapper setSpeedModel(SpeedModel speedModel) {
    this.vehicle.setSpeedModel(speedModel);
    return this;
  }
  public double calcSpeed(double timeSec) {
    double speed = this.vehicle.getSpeedMph(timeSec);
    return Math.round(speed * this.trafficUnit.getTraction());
  }
}
```

We demonstrated the use of such support classes in `Chapter 3`, *Modular Programming*, while talking about streams. Now we realize that testing this class might not be easy.

Because the `SpeedModel` object is an input parameter for the `trafficByLane()` method, we could test its `getSpeedMph()` method in isolation:

```
@Test
public void testSpeedModel(){
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;
  double speed = getSpeedModel().getSpeedMph(timeSec,
                 vehicleWeightPounds, engineHorsePower);
  assertEquals("Assert vehicle (" + engineHorsePower
              + " hp, " + vehicleWeightPounds + " lb) speed in "
              + timeSec + " sec: ", 117, speed, 0.001 * speed);
}

private SpeedModel getSpeedModel(){
  //FactorySpeedModel possibly
}
```

Refer to the following code:

```
public class FactorySpeedModel {
  public static SpeedModel generateSpeedModel(TrafficUnit trafficUnit){
    return new SpeedModelImpl(trafficUnit);
  }
  private static class SpeedModelImpl implements SpeedModel{
```

```
    private TrafficUnit trafficUnit;
    private SpeedModelImpl(TrafficUnit trafficUnit){
      this.trafficUnit = trafficUnit;
    }
    public double getSpeedMph(double timeSec,
                              int weightPounds, int horsePower) {
      double traction = trafficUnit.getTraction();
      double v = 2.0 * horsePower * 746
                 * timeSec * 32.174 / weightPounds;
      return Math.round(Math.sqrt(v) * 0.68 * traction);
    }
  }
```

As you can see, unfortunately, the current implementation of `FactorySpeedModel` requires the `TrafficUnit` object (in order to get the traction value). We need to modify it to extract `SpeedModel` without any dependency in `TrafficUnit` because we will now apply traction in the `calcSpeed()` method. The new version of `FactorySpeedModel` will now look like this:

```
public class FactorySpeedModel {
  public static SpeedModel generateSpeedModel(TrafficUnit trafficUnit) {
    return new SpeedModelImpl(trafficUnit);
  }
  public static SpeedModel getSpeedModel(){
    return SpeedModelImpl.getSpeedModel();
  }
  private static class SpeedModelImpl implements SpeedModel{
    private TrafficUnit trafficUnit;
    private SpeedModelImpl(TrafficUnit trafficUnit){
      this.trafficUnit = trafficUnit;
    }
    public double getSpeedMph(double timeSec,
                      int weightPounds, int horsePower) {
      double speed = getSpeedModel()
             .getSpeedMph(timeSec, weightPounds, horsePower);
      return Math.round(speed *trafficUnit.getTraction());
    }
    public static SpeedModel getSpeedModel(){
      return  (t, wp, hp) -> {
        double weightPower = 2.0 * hp * 746 * 32.174 / wp;
        return Math.round(Math.sqrt(t * weightPower) * 0.68);
      };
    }
  }
}
```

The test method could now be implemented as follows:

```
@Test
public void testSpeedModel(){
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;
  double speed = FactorySpeedModel.generateSpeedModel()
```

```
                .getSpeedMph(timeSec, vehicleWeightPounds,
                             engineHorsePower);
  assertEquals("Assert vehicle (" + engineHorsePower
               + " hp, " + vehicleWeightPounds + " lb) speed in "
               + timeSec + " sec: ", 117, speed, 0.001 * speed);
}
```

However, the `calcSpeed()` method in `TrafficUnitWrapper` remains untested.

We could test the `trafficByLane()` method as a whole:

```
@Test
public void testTrafficByLane() {
  TrafficDensity1 trafficDensity = new TrafficDensity1();
  double timeSec = 10.0;
  int trafficUnitsNumber = 120;
  double[] speedLimitByLane = {30, 50, 65};
  Integer[] expectedCountByLane = {30, 30, 60};
  Integer[] trafficByLane =
    trafficDensity.trafficByLane(getTrafficUnitStream2(
      trafficUnitsNumber), trafficUnitsNumber, timeSec,
      FactorySpeedModel.getSpeedModel(),speedLimitByLane);
    assertArrayEquals("Assert count of "
              + trafficUnitsNumber + " vehicles by "
              + speedLimitByLane.length +" lanes with speed limit "
              + Arrays.stream(speedLimitByLane)
                      .mapToObj(Double::toString)
                      .collect(Collectors.joining(", ")),
                      expectedCountByLane, trafficByLane);
}
```

However, this would require you to create a stream of objects of `TrafficUnit` with fixed data:

```
TrafficUnit getTrafficUnit(int engineHorsePower,
                           int vehicleWeightPounds) {
  return new TrafficUnit() {
    @Override
    public Vehicle.VehicleType getVehicleType() {
      return Vehicle.VehicleType.TRUCK;
    }
    @Override
    public int getHorsePower() {return engineHorsePower;}
    @Override
    public int getWeightPounds() { return vehicleWeightPounds; }
    @Override
    public int getPayloadPounds() { return 0; }
    @Override
    public int getPassengersCount() { return 0; }
    @Override
    public double getSpeedLimitMph() { return 55; }
    @Override
    public double getTraction() { return 0.2; }
    @Override
    public SpeedModel.RoadCondition getRoadCondition() {return null; }
```

```
    @Override
    public SpeedModel.TireCondition getTireCondition() { return null; }
    @Override
    public int getTemperature() { return 0; }
  };
}
```

It is not clear how the data in the `TrafficUnit` object resulted in a different speed value. Besides, we would need to add a variety of test data--for different vehicle types and other parameters--and that is a lot of code to write and maintain.

This means we need to revisit the design of the `trafficByLane()` method. To get confidence that the method will work correctly, we need to test each step of the calculations inside the method in isolation so that each test would require little input data and allow you to have a clear understanding of the expected results.

How it works...

If you look closely at the `trafficByLane()` method, you will notice that the problem is caused by the location of the calculation--inside the private class `TrafficUnitWrapper`. We can move it from there and create a new method of the `TrafficDensity` class:

```
double calcSpeed(double timeSec) {
  double speed = this.vehicle.getSpeedMph(timeSec);
  return Math.round(speed * this.trafficUnit.getTraction());
}
```

Then, we can change its signature to this:

```
double calcSpeed(Vehicle vehicle, double traction, double timeSec) {
  double speed = vehicle.getSpeedMph(timeSec);
  return Math.round(speed * traction);
}
```

Add these two methods to the `TrafficUnitWrapper` class:

```
public Vehicle getVehicle() { return vehicle; }
public double getTraction() { return trafficUnit.getTraction(); }
```

We can now rewrite the stream pipeline like this (the line changed is in bold font):

```
Map<Integer, Integer> trafficByLane = stream
  .limit(trafficUnitsNumber)
  .map(TrafficUnitWrapper::new)
  .map(tuw -> tuw.setSpeedModel(speedModel))
  .map(tuw -> calcSpeed(tuw.getVehicle(), tuw.getTraction(), timeSec))
  .map(speed -> countByLane(lanesCount, speedLimitByLane, speed))
      .collect(Collectors.groupingBy(CountByLane::getLane,
            Collectors.summingInt(CountByLane::getCount)));
```

By making the `calcSpeed()` method protected and assuming that the `Vehicle` class is tested in its own test class `VehicleTest`, we can now write `testCalcSpeed()`:

```
@Test
public void testCalcSpeed(){
  double timeSec = 10.0;
  TrafficDensity2 trafficDensity = new TrafficDensity2();
```

```
  Vehicle vehicle = Mockito.mock(Vehicle.class);
  Mockito.when(vehicle.getSpeedMph(timeSec)).thenReturn(100d);
  double traction = 0.2;
  double speed = trafficDensity.calcSpeed(vehicle, traction, timeSec);
  assertEquals("Assert speed (traction=" + traction + ") in "
              + timeSec + " sec: ",20,speed,0.001 *speed);
}
```

The remaining functionality can now be tested by mocking the `calcSpeed()` method:

```
@Test
public void testCountByLane() {
  int[] count ={0};
  double[] speeds =
                 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
  TrafficDensity2 trafficDensity = new TrafficDensity2() {
    @Override
    protected double calcSpeed(Vehicle vehicle,
                     double traction, double timeSec) {
      return speeds[count[0]++];
    }
  };
  double timeSec = 10.0;
  int trafficUnitsNumber = speeds.length;

  double[] speedLimitByLane = {4.5, 8.5, 12.5};
  Integer[] expectedCountByLane = {4, 4, 4};

  Integer[] trafficByLane = trafficDensity.trafficByLane(
    getTrafficUnitStream(trafficUnitsNumber),
    trafficUnitsNumber, timeSec, FactorySpeedModel.getSpeedModel(),
    speedLimitByLane );
  assertArrayEquals("Assert count of " + speeds.length
          + " vehicles by " + speedLimitByLane.length
          + " lanes with speed limit "
          + Arrays.stream(speedLimitByLane)
                  .mapToObj(Double::toString).collect(Collectors
                  .joining(", ")), expectedCountByLane, trafficByLane);
}
```

There's more...

This experience has made us aware that using an inner private class can make the functionality untestable in isolation. Let's try and get rid of the `private` class `CountByLane`. This leads us to the third version of the `TrafficDensity3` class (we have shown the code that has changed in bold):

```
Integer[] trafficByLane(Stream<TrafficUnit> stream,
int trafficUnitsNumber, double timeSec,
SpeedModel speedModel, double[] speedLimitByLane) {
  int lanesCount = speedLimitByLane.length;
  Map<Integer, Integer> trafficByLane = new HashMap<>();
  for(int i = 1; i <= lanesCount; i++){
    trafficByLane.put(i, 0);
  }
  stream.limit(trafficUnitsNumber)
    .map(TrafficUnitWrapper::new)
    .map(tuw -> tuw.setSpeedModel(speedModel))
    .map(tuw -> calcSpeed(tuw.getVehicle(),
                          tuw.getTraction(), timeSec))
    .forEach(speed -> trafficByLane.computeIfPresent(
      calcLaneNumber(lanesCount,
                     speedLimitByLane, speed), (k, v) -> ++v));
    return trafficByLane.values().toArray(new Integer[lanesCount]);
}
protected int calcLaneNumber(int lanesCount,
  double[] speedLimitByLane, double speed) {
    for(int i = 1; i <= lanesCount; i++){
      if(speed <= speedLimitByLane[i - 1]){
        return i;
      }
    }
    return lanesCount;
}
```

This change allows us to extend the class in the test:

```
private class TrafficDensityTestCalcLaneNumber
extends TrafficDensity3 {
  protected int calcLaneNumber(int lanesCount,
    double[] speedLimitByLane, double speed){
    return super.calcLaneNumber(lanesCount,
    speedLimitByLane, speed);
  }
}
```

It also allows us to change the test method `calcLaneNumber()` in isolation:

```
@Test
public void testCalcLaneNumber() {
  double[] speeds = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
  double[] speedLimitByLane = {4.5, 8.5, 12.5};
  int[] expectedLaneNumber = {1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3};

  TrafficDensityTestCalcLaneNumber trafficDensity =
              new TrafficDensityTestCalcLaneNumber();
  for(int i = 0; i < speeds.length; i++){
    int ln = trafficDensity.calcLaneNumber(
              speedLimitByLane.length,
              speedLimitByLane, speeds[i]);
    assertEquals("Assert lane number of speed "
               + speeds + " with speed limit "
               + Arrays.stream(speedLimitByLane)
                       .mapToObj(Double::toString).collect(
                             Collectors.joining(", ")),
               expectedLaneNumber[i], ln);
  }
}
```

See also

Refer to the following recipes in this chapter:

- Using fixtures to populate data for testing
- Behavioral testing

Using fixtures to populate data for testing

In more complex applications (which use a database, for example), there is often a need to set up the same data before each test and clean it up after each test is run. Some parts of the data need to be set before each test method and cleaned afterwards. You'd also need to have another setup configured before you run the test class, and it should be cleaned up afterwards.

How to do it...

To accomplish this, you can dedicate a method that does the setup by writing a `@Before` annotation in front of it. The corresponding cleaning method is identified by the `@After` annotation. Similar class-level methods are annotated by `@BeforeClass` and `@AfterClass`. Here is a quick demo of this. Add the following methods:

```
public class DatabaseRelatedTest {
  @BeforeClass
  public static void setupForTheClass(){
    System.out.println("setupForTheClass() is called");
  }
  @AfterClass
  public static void cleanUpAfterTheClass(){
    System.out.println("cleanAfterClass() is called");
  }
  @Before
  public void setupForEachMethod(){
    System.out.println("setupForEachMethod() is called");
  }
  @After
  public void cleanUpAfterEachMethod(){
    System.out.println("cleanAfterEachMethod() is called");
  }
  @Test
  public void testMethodOne(){
    System.out.println("testMethodOne() is called");
  }
  @Test
  public void testMethodTwo(){
    System.out.println("testMethodTwo() is called");
  }
}
```

If you run the tests now, you'll get the following result:

```
Running com.packt.cookbook.ch06_db.DatabaseRelatedTest
setupForTheClass() is called
setupForEachMethod() is called
testMethodOne() is called
cleanAfterEachMethod() is called
setupForEachMethod() is called
testMethodTwo() is called
cleanAfterEachMethod() is called
cleanAfterClass() is called
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec
```

Such methods that "fix" the test context are called **fixtures**. Notice that they have to be public, and the class-level setup/cleanup fixtures have to be static. The upcoming JUnit 5 plans to lift these constraints, though.

How it works...

A typical example of such a usage would be creating necessary tables before the first test method is run and removing them after the last method of the test class is finished. The setup/cleanup methods can also be used to create/close a database connection unless your code does it in the try-with-resources construct (refer to `Chapter 12`, *Memory Management and Debugging*).

Here is an example of the usage of fixtures (refer to `Chapter 6`, *Database Programming* on how to set up a database for running it). Let's assume we need to test the `DbRelatedMethods` class:

```
class DbRelatedMethods{
  public void updateAllTextRecordsTo(String text){
    executeUpdate("update text set text = ?", text);
  }
  private void executeUpdate(String sql, String text){
    try (Connection conn = getDbConnection();
      PreparedStatement st = conn.prepareStatement(sql)){
        st.setString(1, text);
        st.executeUpdate();
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
    private Connection getDbConnection(){
       ...
    }
}
```

We would like to make sure that this method always updates all the records of the `text` table with the provided value. Our first test is to update all the existing records:

```
@Test
public void updateAllTextRecordsTo1(){
  System.out.println("updateAllTextRecordsTo1() is called");
  String testString = "Whatever";
  System.out.println("  Update all records to " + testString);
  dbRelatedMethods.updateAllTextRecordsTo(testString);
  int count = countRecordsWithText(testString);
  assertEquals("Assert number of records with "
               + testString + ": ", 1, count);
```

```
  System.out.println("All records are updated to " + testString);
}
```

This means that the table has to exist in the test database and should have a record in it.

Our second test makes sure that all the records are updated even if there is more than one record, and each record contains a different value:

```
@Test
public void updateAllTextRecordsTo2(){
  System.out.println("updateAllTextRecordsTo2() is called");
  String testString = "Whatever";
  System.out.println("  Update all records to Unexpected");
  dbRelatedMethods.updateAllTextRecordsTo("Unexpected");
  executeUpdate("insert into text(id, text) values(2, ?)",
                                                "Text 01");
  System.out.println("Update all records to "  + testString);
  dbRelatedMethods.updateAllTextRecordsTo(testString);
  int count = countRecordsWithText(testString);
  assertEquals("Assert number of records with "
              + testString + ": ", 2, count);
  System.out.println("  " + count + " records are updated to " +
                     testString);
}
```

Both the tests use the same table, that is, `text`. Therefore, there is no need to drop it after each test. This is why we create and drop it at the class level:

```
@BeforeClass
public static void setupForTheClass(){
  System.out.println("setupForTheClass() is called");
  execute("create table text (id integer not null,
           text character varying not null)");
}
@AfterClass
public static void cleanUpAfterTheClass(){
  System.out.println("cleanAfterClass() is called");
  execute("drop table text");
}
```

This means that all we need to do is populate the table before each test and clean it up:

```
@Before
public void setupForEachMethod(){
  System.out.println("setupForEachMethod() is called");
  executeUpdate("insert into text(id, text) values(1,?)", "Text 01");
}
@After
public void cleanUpAfterEachMethod(){
```

```
    System.out.println("cleanAfterEachMethod() is called");
    execute("delete from text");
}
```

Also, since we can use the same object for all the tests, let's create it on the class level too (as the test class's property):

```
private DbRelatedMethods dbRelatedMethods = new DbRelatedMethods();
```

If we run all the tests of the `test` class now, the output would look like this:

```
Running com.packt.cookbook.ch06_db.DbRelatedMethodsTest
setupForTheClass() is called
  create table text (id integer not null, text character varying not null)
setupForEachMethod() is called
  insert into text(id, text) values(1, ?), params=Text 01
updateAllTextRecordsTo1() is called
  Update all records to Whatever
  1 record is updated to Whatever
cleanAfterEachMethod() is called
  delete from text
setupForEachMethod() is called
  insert into text(id, text) values(1, ?), params=Text 01
updateAllTextRecordsTo2() is called
  Update all records to Unexpected
  insert into text(id, text) values(2, ?), params=Text 01
  Update all records to Whatever
  2 records are updated to Whatever
cleanAfterEachMethod() is called
  delete from text
cleanAfterClass() is called
  drop table text
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.36 sec
```

The printed messages allow you to trace the sequence of all the method calls and see that they are executed as expected.

See also

Refer to the following recipe in this chapter:

- Behavioral testing

Behavioral testing

If you have read all the chapters and have looked at the code examples, you may have probably noticed that by now, we have discussed and built all the components necessary for a typical distributed application. Now is the time to put all the components together and see whether they cooperate as expected. This process is called integration.

While doing this, we will look closely at assessing whether the application behaves according to the requirements. In cases where functional requirements are presented in an executable form (using the Cucumber framework, for example), we can run them and check whether all the checks pass. Many software companies follow a behavior-driven development process and perform testing very early, sometimes even before any substantial amount of code is written (such tests fail, of course, but succeed as soon as the expected functionality is implemented). As mentioned already, early testing can be very helpful for writing focused, clear, and well-testable code.

However, even without strict adherence to the *test-first* process, the integration phase naturally includes some kind of behavioral testing. In this recipe, we will see several possible approaches and specific examples related to this.

Getting ready

You might have noticed that in the course of this book, we have built several classes that compose an application that analyzes and models traffic. For your convenience, we have included all of them in the `com.packt.cookbook.ch15_testing` package. You're already familiar with (from Chapters 2, 4, 5, and 7) the five interfaces in the `api` folder: `Car`, `SpeedModel`, `TrafficUnit`, `Truck`, and `Vehicle`. Their implementations are encapsulated inside classes called *factories* in the folder with the same name (used in Chapters 2, 4, 5, and 7): `FactorySpeedModel`, `FactoryTraffic`, and `FactoryVehicle`. These factories produced input for the functionality of the `AverageSpeed` classes (`Chapter 7`, *Concurrent and Multithreaded Programming*) and `TrafficDensity` (based on `Chapter 5`, *Stream Operations and Pipelines* but created and discussed in the current chapter)--the core classes of our demo application. They produce the values that motivated the development of this particular application in the first place.

The main functionality of the application is straightforward. For a given number of lanes and speed limit for each lane, `AverageSpeed` calculates (estimates) the actual speed of each lane (assuming all the drivers are behaving rationally, taking the lane according to their speed), while `TrafficDensity` calculates the number of vehicles in each lane after 10 sec (assuming all the cars start at the same time after the traffic light). The calculations are done based on the data from `numberOfTrafficUnits` vehicles collected at a certain location and time of the year. It does not mean that all the thousand vehicles were racing at the same time. These 1,000 measuring points have been collected over 50 years for approximately 20 vehicles that drove at the specified intersection during the specified hour (which means one vehicle every three minutes on average).

The overall infrastructure of the application is supported by the classes in the `process` folder: `Dispatcher`, `Processor`, and `Subscription` (we discussed their functionality and demonstrated them in `Chapter 7`, *Concurrent and Multithreaded Programming*). These classes allow distributed processing.

The `Dispatcher` class sends a request for processing to the population of `Processors` in a pool, using the `Subscription` class. Each `Processor` class performs the task according to the request (using the `AverageSpeed` and `TraffciDensity` classes) and stores the results in the database (using the `DbUtil` class in the `utils` folder, based on the functionality discussed in `Chapter 6`, *Database Programming*).

We have tested most of these classes as units. Now we are going to integrate them and test the application as a whole for correct behavior.

The requirements were made up just for demo purposes. The goal was to have something well motivated (resembling real data) and at the same time simple enough to understand without special knowledge of traffic analysis and modeling.

How to do it...

There are several levels of integration. We need to integrate the classes and subsystems of the application and also integrate our application with the external system (the source of the traffic data developed and maintained by a third party). Here is an example of class-level integration (see the `demo1_class_level_integration()` method in the `Chapter15Testing` class):

```
String result = IntStream.rangeClosed(1,
  speedLimitByLane.length).mapToDouble(i -> {
    AverageSpeed averageSpeed =
      new AverageSpeed(trafficUnitsNumber, timeSec,
                       dateLocation, speedLimitByLane, i,100);
    ForkJoinPool commonPool = ForkJoinPool.commonPool();
    return commonPool.invoke(averageSpeed);
}).mapToObj(Double::toString).collect(Collectors.joining(", "));
System.out.println("Average speed = " + result);

TrafficDensity trafficDensity = new TrafficDensity();
Integer[] trafficByLane =
     trafficDensity.trafficByLane(trafficUnitsNumber,
                    timeSec, dateLocation, speedLimitByLane );
System.out.println("Traffic density = " + Arrays.stream(trafficByLane)
                                    .map(Object::toString)
                                    .collect(Collectors.joining(", ")));
```

In this example, we integrated each of the two main classes, namely `AverageSpeed` and `TrafficDensity`, with factories and implementation of their API interfaces.

The results are as follows:

```
Average speed = 8.5, 23.5, 44.0
Traffic density = 345, 315, 340
```

Notice that the results are slightly different from one run to another. This is because the data produced by `FactoryTraffic` varies from one request to another. But, at this stage, we just have to make sure that everything works together and produce some more or less accurate-looking results. We have tested the code by units and have a level of confidence that it is doing what

it is supposed to do. We will get back to the results' validation during the actual integration *testing* process, not during integration.

After finishing the integration at the class level, see how the subsystems work together (see the `demo1_subsystem_level_integration()` method in the `Chapter15Testing` class):

```
DbUtil.createResultTable();
Dispatcher.dispatch(trafficUnitsNumber, timeSec, dateLocation,
                    speedLimitByLane);
try { Thread.sleep(2000L); }
catch (InterruptedException ex) {}
Arrays.stream(Process.values()).forEach(v -> {
  System.out.println("Result " + v.name() + ": "
                     + DbUtil.selectResult(v.name()));
});
```

In this code, you can see that we used `DBUtil` to create the necessary table that holds the input data and the results (produced and recorded by `Processor`). The `Dispatcher` class sends a request and inputs data to the objects of the `Processor` class, as shown here:

```
void dispatch(int trafficUnitsNumber, double timeSec,
         DateLocation dateLocation, double[] speedLimitByLane) {
  ExecutorService execService =  ForkJoinPool.commonPool();
  try (SubmissionPublisher<Integer> publisher =
                              new SubmissionPublisher<>()){
    subscribe(publisher, execService,Process.AVERAGE_SPEED,
              timeSec, dateLocation, speedLimitByLane);
   subscribe(publisher,execService,Process.TRAFFIC_DENSITY,
             timeSec, dateLocation, speedLimitByLane);
    publisher.submit(trafficUnitsNumber);
  } finally {
    try {
      execService.shutdown();
      execService.awaitTermination(1, TimeUnit.SECONDS);
    } catch (Exception ex) {
      System.out.println(ex.getClass().getName());
    } finally {
      execService.shutdownNow();
    }
  }
}
```

The `Subscription` class is used to send/get the message (refer to `Chapter 7`, *Concurrent and Multithreaded Programming* for a description of this functionality):

```
void subscribe(SubmissionPublisher<Integer> publisher,
               ExecutorService execService, Process process,
               double timeSec, DateLocation dateLocation,
               double[] speedLimitByLane) {
  Processor<Integer> subscriber =  new Processor<>(process, timeSec,
                                  dateLocation, speedLimitByLane);
  Subscription subscription = new Subscription(subscriber, execService);
  subscriber.onSubscribe(subscription);
  publisher.subscribe(subscriber);
}
```

The processors are doing their job; we just need to wait for a few seconds (you might adjust this time if the computer you are using requires more time to finish the job) before we get the results (using DBUtil for reading the recorded results from the database):

```
AVERAGE_SPEED: 8.5, 23.0, 43.0
TRAFFIC_DENSITY: 370, 314, 316
```

The names of the `Process` enum class point to the corresponding records in the `result` table in the database. Again, at this stage, we are primarily looking for getting any results at all, not at how correct the values are.

After the successful integration between the subsystems of our application (based on the generated data from `FactoryTraffic`), we can try and connect to the external system that provides real traffic data. Inside `FactoryTraffic`, we would now switch from generating `TrafficUnit` objects to getting data from a real system:

```
public class FactoryTraffic {
  private static boolean switchToRealData = true;
  public static Stream<TrafficUnit>
  getTrafficUnitStream(DateLocation dl, int trafficUnitsNumber){
    if(switchToRealData){
      return getRealData(dL,  trafficUnitsNumber);
    } else {
      return IntStream.range(0, trafficUnitsNumber)
      .mapToObj(i -> generateOneUnit());
    }
  }

  private static Stream<TrafficUnit>
  getRealData(DateLocation dl, int trafficUnitsNumber) {
    //connect to the source of the real data
    // and request the flow or collection of data
    return new ArrayList<TrafficUnit>().stream();
  }
}
```

The switch can be implemented as a Boolean property in the class (as seen in the preceding code) or the project configuration property. We leave out the details of the connection to a particular source of real traffic data as this is not relevant to the purpose of this book.

The main focus at this stage has to be the performance and having a smooth data flow between the external source of real data and our application. After we have made sure that everything works and produces results (that look realistic) with satisfactory performance, we can turn to integration *testing* (with the actual results' assertion).

How it works...

For testing, we need to set the expected values, which we can then can compare with the (actual) values, produced by the application that processes real data. But real data change slightly from run to run, and an attempt to predict the resulting values either makes the test fragile or forces the introduction of a huge margin of error, which may effectively defeat the purpose of testing.

We cannot even mock the generated data (as we did in the case of unit testing) because we are at the integration stage and have to use the real data.

One possible solution would be to store the incoming real data and the result our application produced along with them in the database. Then, a domain specialist can walk through each record and assert whether the results are as expected.

To accomplish this, we introduced a boolean switch in the `TrafficDensity` class, so it records the input along with each unit of the calculations:

```
public class TrafficDensity {
  public static Connection conn;
  public static boolean recordData = false;
  //...
  private double calcSpeed(TrafficUnitWrapper tuw, double timeSec){
    double speed = calcSpeed(tuw.getVehicle(),
    tuw.getTrafficUnit().getTraction(), timeSec);
    if(recordData) {
      DbUtil.recordData(conn, tuw.getTrafficUnit(), speed);
    }
    return speed;
  }
  //...
}
```

We have also introduced a static property to keep the same database connection across all the class instances. Otherwise, the connection pool should be very big because, as you may recall from Chapter 7, *Concurrent*

and Multithreaded Programming, the number of workers that execute the task in parallel grows with the growth of the amount of work to do.

If you look at `DbUtils`, you will see a new method that creates the `data` table (designed to hold `TrafficUnits` coming from `FactoryTraffic`) and the `data_common` table that keeps the main parameters used for data requests and calculations: requested traffic units' number, the date and geolocation of the traffic data, time in seconds (the point when the speed is calculated), and the speed limit for each lane (its size defines how many lanes we plan to use while modeling the traffic). Here is the code that we configure to do the recording:

```
private static void demo3_prepare_for_integration_testing(){
  DbUtil.createResultTable();
  DbUtil.createDataTables();
  TrafficDensity.recordData = true;
  try(Connection conn = DbUtil.getDbConnection()){
    TrafficDensity.conn = conn;
    Dispatcher.dispatch(trafficUnitsNumber, timeSec,
                        dateLocation, speedLimitByLane);
  } catch (SQLException ex){
    ex.printStackTrace();
  }
}
```

After the recording is completed, we can turn the data over to a domain specialist who can assert the correctness of the application behavior.

The verified data can now be used for integration testing. We can add another switch to `FactoryTrafficUnit` and force it to read the recorded data instead of the unpredictable real or generated ones:

```
public class FactoryTraffic {
  public static boolean readDataFromDb = false;
  private static boolean switchToRealData = false;
  public static Stream<TrafficUnit>
                getTrafficUnitStream(DateLocation dl,
                                     int trafficUnitsNumber){
    if(readDataFromDb){
      if(!DbUtil.isEnoughData(trafficUnitsNumber)){
        System.out.println("Not enough data");
        return new ArrayList<TrafficUnit>().stream();
      }
      return readDataFromDb(trafficUnitsNumber);
    }
    //....
}
```

As you may have noticed, we have also added method `isEnoughData()` that checks whether there is enough recorded data:

```
public static boolean isEnoughData(int trafficUnitsNumber){
  try (Connection conn = getDbConnection();
  PreparedStatement st =
      conn.prepareStatement("select count(*) from data")){
    ResultSet rs = st.executeQuery();
    if(rs.next()){
      int count = rs.getInt(1);
      return count >= trafficUnitsNumber;
    }
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  return false;
}
```

This will help avoid the unnecessary frustration of debugging the test problem, especially in the case of testing a more complex system.

Now we can predict not only the input data, but also the expected results that we can use to assert the application behavior. Both are now included in the `TrafficUnit` object. To be able to do this, we took advantage of the new Java interface feature discussed in Chapter 2, *Fast Track to OOP - Classes and Interfaces*, which is the default method:

```
public interface TrafficUnit {
  VehicleType getVehicleType();
  int getHorsePower();
  int getWeightPounds();
  int getPayloadPounds();
  int getPassengersCount();
  double getSpeedLimitMph();
  double getTraction();
  RoadCondition getRoadCondition();
  TireCondition getTireCondition();
  int getTemperature();
  default double getSpeed(){ return 0.0; }
}
```

So, we can attach the result to the input data. See the following method:

```
List<TrafficUnit> selectData(int trafficUnitsNumber){...}
```

We can attach the result to the `DbUtil` class and the `TrafficUnitImpl` class inside `DbUtil` too:

```
class TrafficUnitImpl implements TrafficUnit{
  private int horsePower, weightPounds, payloadPounds,
              passengersCount, temperature;
  private Vehicle.VehicleType vehicleType;
  private double speedLimitMph, traction, speed;
  private RoadCondition roadCondition;
  private TireCondition tireCondition;
  ...
  public double getSpeed() { return speed; }
}
```

We can attach it inside the `DbUtil` class too.

Now we can write an integration test. First, we will test the speed model using the recorded data:

```
void demo1_test_speed_model_with_real_data(){
  double timeSec = DbUtil.getTimeSecFromDataCommon();
  FactoryTraffic.readDataFromDb = true;
  TrafficDensity trafficDensity = new TrafficDensity();
  FactoryTraffic.getTrafficUnitStream(dateLocation, 1000).forEach(tu -> {
    Vehicle vehicle = FactoryVehicle.build(tu);
    vehicle.setSpeedModel(FactorySpeedModel.getSpeedModel());
    double speed = trafficDensity.calcSpeed(vehicle,
                                tu.getTraction(), timeSec);
    assertEquals("Assert vehicle (" + tu.getHorsePower()
                 + " hp, " + tu.getWeightPounds() + " lb) speed in "
                 + timeSec + " sec: ", tu.getSpeed(), speed,
                 speed * 0.001);
  });
}
```

A similar test can be written for testing the speed calculation of the `AverageSpeed` class with real data.

Then, we can write an integration test for the class level:

```
private static void demo2_class_level_integration_test() {
  FactoryTraffic.readDataFromDb = true;
  String result = IntStream.rangeClosed(1,
              speedLimitByLane.length).mapToDouble(i -> {
    AverageSpeed averageSpeed = new AverageSpeed(trafficUnitsNumber,
                timeSec, dateLocation, speedLimitByLane, i,100);
    ForkJoinPool commonPool = ForkJoinPool.commonPool();
    return commonPool.invoke(averageSpeed);
  }).mapToObj(Double::toString).collect(Collectors.joining(", "));
  String expectedResult = "7.0, 23.0, 41.0";
  String limits = Arrays.stream(speedLimitByLane)
                        .mapToObj(Double::toString)
                        .collect(Collectors.joining(", "));
  assertEquals("Assert average speeds by "
                + speedLimitByLane.length
```

```
                + " lanes with speed limit "
                + limits, expectedResult, result);
```

Similar code can be written for the class level testing of class `TrafficDensity` too:

```
TrafficDensity trafficDensity = new TrafficDensity();
String result = Arrays.stream(trafficDensity.
       trafficByLane(trafficUnitsNumber, timeSec,
                     dateLocation, speedLimitByLane))
       .map(Object::toString)
       .collect(Collectors.joining(", "));
expectedResult = "354, 335, 311";
assertEquals("Assert vehicle count by " + speedLimitByLane.length +
         " lanes with speed limit " + limits, expectedResult, result);
```

Finally, we can write the integration test for the subsystem level as well:

```
void demo3_subsystem_level_integration_test() {
  FactoryTraffic.readDataFromDb = true;
  DbUtil.createResultTable();
  Dispatcher.dispatch(trafficUnitsNumber, 10, dateLocation,
                      speedLimitByLane);
  try { Thread.sleep(3000l); }
  catch (InterruptedException ex) {}
  String result = DbUtil.selectResult(Process.AVERAGE_SPEED.name());
  String expectedResult = "7.0, 23.0, 41.0";
  String limits = Arrays.stream(speedLimitByLane)
                        .mapToObj(Double::toString)
                        .collect(Collectors.joining(", "));
  assertEquals("Assert average speeds by " + speedLimitByLane.length
        + " lanes with speed limit " + limits, expectedResult, result);
  result = DbUtil.selectResult(Process.TRAFFIC_DENSITY.name());
  expectedResult = "354, 335, 311";
  assertEquals("Assert vehicle count by " + speedLimitByLane.length
        + " lanes with speed limit " + limits, expectedResult, result);
}
```

All of them can be run successfully now and may be used for application regression testing any time later.

An automated integration test between our application and the source of the real traffic data can be created only if the latter has a test mode from where the same flow of data can be sent our way so we can use them in the same manner we use recorded data (which is essentially the same thing).

One parting thought. All of this integration testing is possible when the amount of processing data is statistically significant. This is because we do not have full control over the number of workers and how the JVM decides

to split the load. It is quite possible that on a particular occasion, the provided code would not work. In such a case, try to increase the number of requested traffic units. This will ensure more space for the load-distributing logic.