Exploring Java 9

Build Modularized Applications in Java

Fu Cheng

Apress®

Exploring Java 9

Build Modularized Applications in Java

Fu Cheng

Apress®

Exploring Java 9

Fu Cheng
Auckland, New Zealand

ISBN-13 (pbk): 978-1-4842-3329-0 ISBN-13 (electronic): 978-1-4842-3330-6
https://doi.org/10.1007/978-1-4842-3330-6

Library of Congress Control Number: 2017962328

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Technical Reviewer: Massimo Nardone
Coordinating Editor: Jessica Vakili
Copy Editor: Rebecca Rider
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3329-0. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my wife Andrea and my daughters Olivia and Erica

Contents

About the Author xiii

About the Technical Reviewer xv

Chapter 1: Introduction 1

Installation 1

IDE 2

IntelIij IDEA 2

Eclipse 2

Build Tools 4

Gradle 4

Apache Maven 4

javac and java 4

Docker 5

CI Builds 6

Summary 6

Chapter 2: The Module System 7

Module Introduction 8

Sample Application 8

Module Declaration 9

requires and exports 9

Transitive Dependencies 9

Static Dependencies 12

Services 12

Qualified Exports 14
Opening Modules and Packages 14
Working with Existing Code 15
Unnamed Modules 15
Automatic Modules 16
JDK Tools 17
Module Paths 17
Module Version 18
The Main Module 18
Root Modules 18
Limiting the Observable Modules 19
Upgrading the Module Path 19
Increasing Readability and Breaking Encapsulation 19
javac 20
jlink 21
java 24
jdeps 24
Module Java API 27
ModuleFinder 27
ModuleDescriptor 28
Configuration 31
The Module Layers 34
Class Loaders 39
Class 42
Reflection 43
Automatic Module Names 43
Module Artifacts 46
JAR Files 46
JMOD Files 47
JDK Modules 49

Common Issues 49
Migration in Action 51
Building the Project Using Java 9 51
The Migration Path 51
BioJava 52
Summary 56
Chapter 3: jshell 57
Code Completion 58
Classes 58
Methods 59
Commands 59
/list 59
/edit 60
/drop 61
/save 61
/open 61
/imports 62
/vars 62
/types 62
/methods 63
/history 63
/env 63
/set 64
/reset 64
/reload 64
/! 65
/<id> 65
/-<n> 65
/exit 65
Summary 65

Chapter 4: Collections, Stream, and Optional 67

Factory Methods for Collections 67

The List.of() Method 67

The Set.of() Method 67

The Map.of() and Map.ofEntries() Methods 68

Arrays 68

Mismatch() Methods 68

Compare() Methods 69

Equals() Methods 69

Stream 69

The ofNullable() Method 69

The dropWhile() Method 70

The takeWhile() Method 70

The iterate() Method 71

IntStream, LongStream, and DoubleStream 71

Collectors 71

The filtering() Method 71

The flatMapping() Method 72

Optional 73

The ifPresentOrElse() Method 73

The Optional.or() Method 73

The stream() Method 74

Summary 74

Chapter 5: The Process API 75

The ProcessHandle Interface 75

Process 77

Managing Long-Running Processes 78

Summary 79

Chapter 6: The Platform Logging API and Service 81
Default LoggerFinder Implementation 82
Creating Custom LoggerFinder Implementations 83
Summary 86
Chapter 7: Reactive Streams 87
Core Interfaces 87
Flow.Publisher<T> 87
Flow.Subscriber<T> 87
Flow.Subscription 88
Flow.Processor<T,R> 88
SubmissionPublisher 88
Third-Party Libraries 95
RxJava 2 95
Reactor 96
Interoperability 97
Summary 97
Chapter 8: Variable Handles 99
Creating Variable Handles 99
findStaticVarHandle 99
findVarHandle 99
unreflectVarHandle 100
Access Modes 100
Memory Ordering 100
VarHandle Methods 101
Arrays 105
byte[] and ByteBuffer Views 106
Memory Fence 107
Summary 107

Chapter 9: Enhanced Method Handles 109
arrayConstructor 109
arrayLength 109
varHandleInvoker and varHandleExactInvoker 110
zero 110
empty 111
Loops 111
loop 111
countedLoop 113
iteratedLoop 114
whileLoop and doWhileLoop 115
Try-finally 116
Summary 117
Chapter 10: Concurrency 119
CompletableFuture 119
Async 119
Timeout 119
Utilities 120
TimeUnit and ChronoUnit 120
Queues 121
Atomic Classes 122
Thread.onSpinWait 123
Summary 124
Chapter 11: Nashorn 125
Getting the Nashorn Engine 125
ECMAScript 6 Features 126
Template Strings 126
Binary and Octal Literals 126
Iterators and for..of Loops 126
Functions 127

Parser API 128
Basic Parsing 128
Parsing Error 129
Analyzing Function Complexity 130
Summary 131
Chapter 12: I/O 133
InputStream 133
The ObjectInputStream Filter 134
Summary 137
Chapter 13: Security 139
SHA-3 Hash Algorithms 139
SecureRandom 139
Using PKCS12 as the Default Keystore 141
Summary 141
Chapter 14: User Interface 143
Desktop 143
Application Events 143
About Window 144
Preferences Window 144
Open Files 145
Print Files 146
Open URI 146
Application Exit 146
Other Functionalities 148
Multiresolution Images 148
TIFF Image Format 150
Deprecating the Applet API 150
Summary 150

Chapter 15: JVM 151

Unified Logging 151

Tags, Levels, Decorations, and Output 151

Logging Configuration 152

The Diagnostic Command VM.log 153

Remove GC Combinations 154

Making G1 the Default Garbage Collector 154

Deprecating the Concurrent Mark Sweep (CMS) Garbage Collector 154

Removing Launch-Time JRE Version Selection 154

More Diagnostic Commands 155

Removal of the JVM TI hprof Agent 157

Removal of the jhat Tool 157

Removal of Demos and Samples 157

Javadoc 157

Summary 159

Chapter 16: Miscellaneous 161

Small Language Changes 161

Private Interface Methods 161

Resource References in try-with-resources 161

Other Changes 162

The Stack-Walking API 162

Objects 164

Unicode 8.0 165

UTF-8 Property Resource Bundles 166

Enhanced Deprecation 166

NetworkInterface 167

Summary 168

Index 169

About the Author

Fu Cheng is a full-stack software developer working in a healthcare start-up in Auckland, New Zealand. During his many years of experiences, he has worked in different companies to build large-scale enterprise systems, government projects, and SaaS products. He is an experienced JavaScript and Java developer and always wants to learn new things. He enjoys sharing knowledge by writing blog posts, technical articles, and books.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experiences in Security, Web/Mobile development, and Cloud and IT Architecture. His true IT passions are Security and Android.

He has been programming and teaching others how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

Massimo holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has held the positions of Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

Massimo's technical skills include Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web and Mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, and more.

He currently works as the Chief Information Security Office (CISO) for Cargotec Oyj.

He has also been a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing company and he is the coauthor of *Pro Android Games* (Apress, 2015).

He dedicates this book to Antti Jalonen and his family who are always there when Massimo needs them.

CHAPTER 1

Introduction

Java SE 9 was released on September 21, 2017. It's the first major release of the Java platform since Java SE 8 was released on March 18, 2014. The Java community has been waiting for this release for quite a long time. The release of Java 9 was delayed several times. In this release, the Java Platform Module System (JPMS), or Project Jigsaw, introduces a module system to the Java platform. Now we can create modular Java applications using Java 9. The contents of Java SE 9 can be found in JSR 379: Java™ SE 9 Release Contents (https://www.jcp.org/en/jsr/detail?id=379). When I'm talking about Java 9 in this book, I'm referring to Java SE 9 and JDK 9.

Installation

You can download the JDK 9 General Availability (GA) release builds from Oracle (http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html). You can choose the build for your platform and install it on your local machine.

After the installation, you can run `java -version` to check the version. Listing 1-1 shows the version information of JDK 9 on macOS.

Listing 1-1. JDK 9 Version on macOS

```
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

Note Java 9 uses a new schema for version strings. The version strings before Java 9 start with `1`, for example, `1.8.0_60` for Java 8. In Java 9, the leading `1` of the original version strings has been removed. The version strings for Java 9 start with `9`. Java 9 also provides the class `Runtime.Version` to represent and parse the version strings.

Installing Java 9 GA builds makes them the default JVM on your local machine, which may break some Java applications that are not yet compatible with Java 9. You can use a tool like jEnv (http://www.jenv.be/) to manage multiple JDK installations or update the environment variable `JAVA_HOME` manually to point to your old JDK installations. Some of these applications may have options for configuring the JRE for use. For these applications, you can use the options to point to the old JDK installations.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_1

IDE

IDEs are very useful for Java developers. I'll discuss Java 9 support of two popular IDEs: Intellij IDEA and Eclipse.

Intellij IDEA

Intellij IDEA (`https://www.jetbrains.com/idea/`) 2017.1 already supports Java 9. You can simply install Java 9 GA builds and add JDK 9 as the project SDK. Don't forget to change the language level to 9; see Figure 1-1.

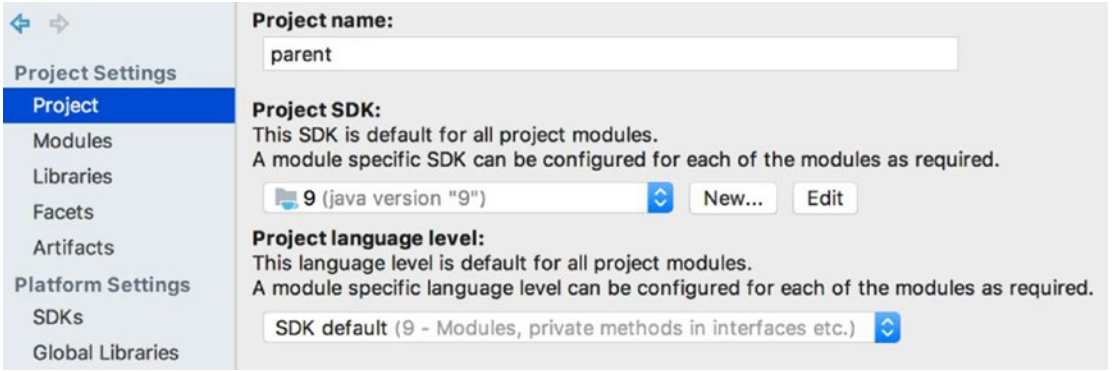

Figure 1-1. *Intellij IDEA Java 9 setup*

All the code in this book is written and tested using IDEA 2017.2.

Eclipse

At the time of writing, when compared to using Intellij IDEA, using Eclipse for Java 9 development requires more manual setup steps. To run Java 9 you need at least Eclipse Oxygen (4.7) with Eclipse Marketplace Client installed. When you're ready to get started, you first need to modify the file `eclipse.ini` to specify the JVM and add extra arguments. If Java 9 is not the default JVM, you need to use the option `-vm` to specify the JDK 9 installation path, for example, `-vm /Library/Java/JavaVirtualMachines/jdk-9.0.1.jdk/Contents/Home/bin/javaw`. You also need to add the JVM argument `--add-modules=ALL-SYSTEM`, which should be placed after `-vmargs`. After you've done this, you can start Eclipse using JDK 9. Finally, you need to open **Eclipse Marketplace** in the **Help** menu, search for "Java 9," and then install the **Java 9 Support (BETA) for Oxygen 4.7**; see Figure 1-2. After you restarting Eclipse, you can add a new Java project with Java SE 9 as the execution environment and the compiler compliance level set to `9`.

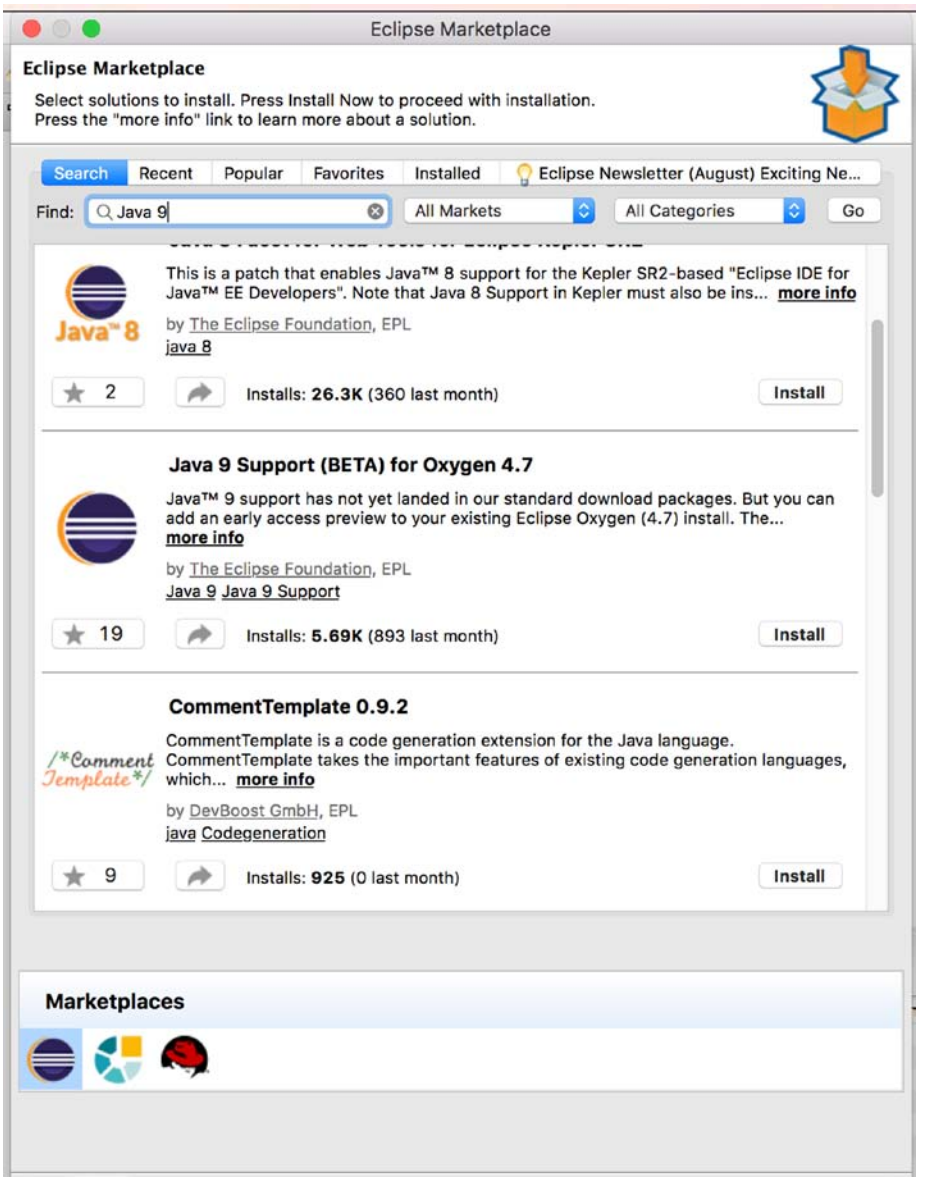

Figure 1-2. *Install Eclipse support for Java 9*

Eclipse continues to improve its support for Java 9. You should always refer to the latest official guide for its Java 9 support.

Build Tools

Typically, you need to use some kinds of build tools when you're developing nontrivial Java applications. The popular choices are Gradle (https://gradle.org/) and Apache Maven (https://maven.apache.org/).

Gradle

The support of Gradle for Java 9 is tracked in this GitHub issue (https://github.com/gradle/gradle/issues/719). According to the latest status (https://github.com/gradle/gradle/issues/719#issuecomment-312225355), you should use at least Gradle version `4.1-milestone-1` to run with Java 9.

> *For the most part, version 4.1-milestone-1 requires no special environment settings to get Gradle running on a simple Java project. However, there are still many, commonly-used plugins for which at least --permit-illegal-access will be required to run.*

The option `--permit-illegal-access` has been renamed to `--illegal-access` and already has the default value `permit`, so it's no longer required to add this option. At the time of writing, Gradle 4.2 doesn't provide first-class support of modules. You can experiment with Gradle's module support using this guide (https://guides.gradle.org/building-java-9-modules/).

Apache Maven

The Maven Compiler plugin (https://maven.apache.org/plugins/maven-compiler-plugin/) supports compiling Java 9 modules. Listing 1-2 shows how to configure this plugin in Maven projects.

Listing 1-2. Maven Compiler Plugin Configuration

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.7.0</version>
    <configuration>
        <source>9</source>
        <target>9</target>
        <fork>true</fork>
    </configuration>
</plugin>
```

javac and java

If you just want to run some quick tests to see how Java 9 works, you can skip the setup with Gradle or Maven and use `javac` and `java` directly. Java code can be compiled using `javac` and run using `java.` `javac` and `java` commands both have new options, which I cover in later chapters.

You should only use `javac` and `java` for small projects. For large projects, these command line arguments are very hard to manage. Use Gradle or Maven for nontrivial projects.

Docker

If you prefer to use Docker for local development, the official openjdk (https://hub.docker.com/_/openjdk/) Docker image already has Java 9 builds. If you also use Apache Maven to build projects, then the official Apache Maven (https://hub.docker.com/r/library/maven/) Docker image is better than the openjdk image. I use the tag `3.5.0-jdk-9` to get Maven 3.5.0 with JDK 9.

After pulling down the image, you can run `java -version` to check the version; see Listing 1-3.

Listing 1-3. Checking JDK 9 Version in the Docker Image

```
$ docker run -it --rm --name maven-java9 \
  maven:3.5-jdk-9 java -version
```

As shown in Listing 1-4, the JDK 9 build in this image is `9.0.1`.

Listing 1-4. JDK 9 Version in the Docker Image

```
openjdk version "9.0.1"
OpenJDK Runtime Environment (build 9.0.1+11-Debian-1)
OpenJDK 64-Bit Server VM (build 9.0.1+11-Debian-1, mixed mode)
```

Now you can use `javac` to compile Java files; see Listing 1-5. Java code files are in the current directory. Compiled class files are in the directory `classes`.

Listing 1-5. Running `javac` Using Docker

```
$ docker run -it --rm --name maven-java9 \
  -v "$PWD":/usr/src/java9 \
  -w /usr/src/java9 maven:3.5-jdk-9 \
  javac demo/Main.java -d classes
```

After the compilation, you can use `java` to run the compiled code; see Listing 1-6.

Listing 1-6. Running `java` Using Docker

```
$ docker run -it --rm --name maven-java9 \
  -v "$PWD":/usr/src/java9 \
  -w /usr/src/java9 maven:3.5-jdk-9 \
  java -cp classes demo.Main
```

■ **Note** If you want to use Docker as your primary approach for playing with Java 9, it's worth reading the article "Executable Images—How to Dockerize Your Development Machine" (https://www.infoq.com/articles/docker-executable-images). It covers important techniques for utilizing Docker in local development.

CI Builds

We run CI builds for the source code using CircleCI (https://circleci.com). Listing 1-7 shows the `circle.yml` of CircleCI. Here we use the Docker image `maven:3.5.0-jdk-9` to run the Maven builds.

Listing 1-7. CircleCI Configuration File

```
version: 2
jobs:
  build:
    working_directory: ~/circleci-feature9
    docker:
      - image: maven:3.5.0-jdk-9

    steps:
      - checkout

      - restore_cache:
          key: circleci-feature9-{{ checksum "pom.xml" }}

      - run: mvn test

      - save_cache:
          paths:
            - ~/.m2
          key: circleci-feature9-{{ checksum "pom.xml" }}

      - store_test_results:
          path: target/surefire-reports
```

If you're using other CI services, check their guides for how to build Maven projects using Java 9.

Summary

This chapter covers the basic setup required to develop applications using Java 9, including JDK 9 installation, IDE setup, build tools configuration, Docker setup, and CI builds configuration. In the next chapter, we'll discuss the most important feature in Java 9—the Java Platform Module System.

CHAPTER 2

The Module System

When we're talking about Java 9, the most important topic is Project Jigsaw (http://openjdk.java.net/projects/jigsaw/) or the Java Platform Module System (JPMS), which introduces the module system into the Java platform. Project Jigsaw was supposed to be added in Java 8, but the changes were too big, so it was delayed to release with Java 9. Project Jigsaw brings significant changes to the Java platform, not only to the JDK itself, but also to Java applications running on it.

The Java SE platform and JDK are organized in modules in Java 9 so they can be customized to scale down to run on small devices. Before Java 9, the installation of JRE was all-or-nothing. The JRE contains tools, libraries, and classes that can satisfy requirements for running different applications. But some of these tools, libraries, and classes may not be required for a particular application. For example, a REST API proxy running on the JRE may never use the desktop AWT/Swing library. JPMS makes it possible to strip unnecessary libraries from the JRE to build customized images that are suitable for every unique application. This can dramatically reduce the package size making deployment faster.

The Java community has always wanted a way to build modular Java applications. OSGi (https://en.wikipedia.org/wiki/OSGi) is a good choice for doing this at the moment. JPMS also allows developers to create modular Java libraries and applications. Compared to using OSGi, a solution from Java platform itself is more promising.

JPMS is complicated with one JSR—JSR 376: Java™ Platform Module System (https://jcp.org/en/jsr/detail?id=376)—and six related JEPs.

- 200: The Modular JDK
- 201: Modular Source Code
- 220: Modular Run-Time Images
- 260: Encapsulate Most Internal APIs
- 261: Module System
- 282: `jlink`: The Java Linker

This chapter covers the most important concepts of JPMS.

F. Cheng, *Exploring Java 9,* https://doi.org/10.1007/978-1-4842-3330-6_2

Module Introduction

Take a look at this quote from a document (http://openjdk.java.net/projects/jigsaw/spec/sotms/) by Mark Reinhold (http://mreinhold.org/), the chief architect of the Java Platform Group at Oracle:

> *A module is a named, self-describing collection of code and data. Its code is organized as a set of packages containing types, i.e., Java classes and interfaces; its data includes resources and other kinds of static information.*
>
> Mark Reinhold

From this definition, we get that a *module* is just a set of compiled Java code and supplementary resources that are organized in a predefined structure. If you already use Maven multiple modules (https://maven.apache.org/guides/mini/guide-multiple-modules.html) or Gradle multiproject builds (https://docs.gradle.org/current/userguide/multi_project_builds.html) to organize your code, then you can easily upgrade each of these Maven modules or Gradle projects to a JPMS module.

Each JPMS module should have a name that follows the same naming convention as Java packages; that is, it should use the *reverse-domain-name* pattern—for example, `com.mycompany.mymodule`. A JPMS module is described using the file `module-info.java` in the root source directory, which is compiled to `module-info.class`. In this file, you use the new keyword `module` to declare a module with a name. Listing 2-1 shows the content of the file `module-info.java` of the module `com.mycompany.mymodule` with minimal information.

Listing 2-1. Minimal Module Description

```
module com.mycompany.mymodule {

}
```

Now you have successfully created a new JPMS module.

Sample Application

Let me use a sample application to demonstrate the usage of the module system. This is a simple e-commerce application with very limited features. The main objective is to demonstrate how the module system works, so the actual implementations of these modules don't really matter. The sample application is a Maven project with modules shown in Table 2-1. The namespace of this application is `io.vividcode.store`, so the module name of `common` in Table 2-1 is actually `io.vividcode.store.common`.

Table 2-1. *Modules of the Sample Application*

Name	Description
`common`	Common API
`common.persistence`	Common persistence API
`filestore`	File-based persistence implementation
`product`	Product API
`product.persistence`	Product persistence implementation
`runtime`	Application bootstrap

Module Declaration

The module declaration file `module-info.java` is the first step to understanding how module system works.

`requires` and `exports`

After the introduction of modules in Java 9, you should organize Java applications as modules. A module can declare its dependencies upon other modules using the keyword `requires`. Requiring a module doesn't mean that you have access to its public and protected types automatically. A module can declare which packages are accessible to other modules. Only a module's exported packages are accessible to other modules, and by default, no packages are exported. The module declaration in Listing 2-1 exports nothing. The keyword `exports` is used to export packages. Public and protected types in exported packages and their public and protected members can be accessed by other modules.

Listing 2-2 shows the file `module-info.java` of the module `io.vividcode.store.common.persistence`. It uses two `requires` declarations to declare its dependencies upon modules `slf4j.api` and `io.vividcode.store.common`. The module `slf4j.api` comes from the third-party library SLF4J (https://www.slf4j.org/), while `io.vividcode.store.common` is another module in the sample application. The module `io.vividcode.store.common.persistence` exports its package `io.vividcode.store.common.persistence` to other modules.

Listing 2-2. Module Declaration of `io.vividcode.store.common.persistence`

```
module io.vividcode.store.common.persistence {
  requires slf4j.api;
  requires io.vividcode.store.common;
  exports io.vividcode.store.common.persistence;
}
```

Please note, when you export a package, you only export types in this package but not types in its subpackages. For example, the declaration `exports com.mycompany.mymodule` only exports types like `com.mycompany.mymodule.A` or `com.mycompany.mymodule.B`, but not types like `com.mycompany.mymodule.impl.C` or `com.mycompany.mymodule.test.demo.D`. To export those subpackages, you need to use `exports` to explicitly declare them in the module declaration.

If a type is not accessible across module boundaries, this type is treated like a private method or field in the module. Any attempt to use this type will cause an error in the compile time, a `java.lang.IllegalAccessError` thrown by the JVM in the runtime, or a `java.lang.IllegalAccessException` thrown by the Java reflection API when you are using reflection to access this type.

■ **Note** All modules, except for the module `java.base` itself, have implicit and mandatory dependency upon `java.base`. You don't need to explicitly declare this dependency.

Transitive Dependencies

When module A requires module B, module A can read public and protected types exported in module B. Here we say module A *reads* module B. If module B also reads module C, module B can have methods that return types exported in module C.

Listing 2-3 shows the file `module-info.java` of module C. Module C exports the package `ctest`.

Listing 2-3. Module Declaration of C

```
module C {
  exports ctest;
}
```

Listing 2-4 shows the class `ctest.MyC` in module C. It only has one method, `sayHi()`, that prints out a message to the console.

Listing 2-4. Class `ctest.MyC` in Module C

```
package ctest;

public class MyC {
  public void sayHi() {
    System.out.println("Hi from module C!");
  }
}
```

Listing 2-5 shows the file `module-info.java` of module B. Module B requires module C and exports the package `btest`.

Listing 2-5. Module Declaration of B

```
module B {
  requires C;
  exports btest;
}
```

The method `getC()` of the class `btest.MyB` in Listing 2-6 returns a new instance of the class `ctest.MyC`.

Listing 2-6. Class `btest.MyB` in Module B

```
package btest;

import ctest.MyC;

public class MyB {
  public MyC getC() {
    return new MyC();
  }
}
```

The module A only requires module B in its file `module-info.java`; see Listing 2-7.

Listing 2-7. Module Declaration of B

```
module A {
  requires B;
}
```

The class `atest.MyA` in module A tries to use the class `MyC`; see Listing 2-8.

Listing 2-8. Class `atest.MyA` in Module A

```
package atest;

import btest.MyB;

public class MyA {
  public static void main(String[] args) {
    new MyB().getC().sayHi();
  }
}
```

Although the code in Listing 2-8 looks quite reasonable, it cannot be compiled due to the error that module A doesn't read module C; see the error message in Listing 2-9. The module readability relationship is not transitive by default. Module B reads module C, module A reads module B, but module A doesn't read module C.

Listing 2-9. Compile Error of Module A

```
/<code_path>/A/atest/MyA.java:7: error: MyC.sayHi() in package ctest is not
 accessible
  new MyB().getC().sayHi();
          ^
  (package ctest is declared in module C, but module A does not read it)
1 error
```

To make the code in Listing 2-8 compile, you need to add `requires C` to the file `module-info.java` of module A in Listing 2-7. This can be a tedious task when many modules depend on each other. Since this is a common usage scenario, Java 9 provides built-in support for it. The `requires` declaration can be extended to add the modifier `transitive` to declare the dependency as transitive. The transitive modules that a module depends on are readable by any module that depends upon this module. This is called *implicit readability*.

After the declaration of module B is changed to use the modifier `transitive` in Listing 2-10, module A can be compiled successfully. The transitive module C that module B depends on is readable by module A, which depends upon module B. Module A can now read module C.

Listing 2-10. Updating the Module Declaration of B to Use `transitive`

```
module B {
  requires transitive C;
  exports btest;
}
```

In general, if one module exports a package containing a type whose signature refers to another package in a second module, then the first module should use `requires transitive` to declare the dependency upon the second module. As in module B, the method `getC()` of class `MyB` references the class `MyC` from module C, so module B should use `requires transitive C` instead of `requires C`.

Static Dependencies

You can use `requires static` to specify that a module dependency is required in the compile time, but optional in the runtime; see Listing 2-11.

Listing 2-11. Example of `requires static`

```
module demo {
  requires static A;
}
```

Static dependencies are useful for frameworks and libraries. Suppose that you are building a library to work with different kinds of databases. The library module can use static dependencies to require different kinds of JDBC drivers. At compile time, the library's code can access types defined in those drivers. At runtime, users of the library can add only the drivers they want to use. If the dependencies are not static, users of the library have to add all supported drivers to pass the module resolution checks.

Services

Java has its own service interfaces and providers mechanism using the class `java.util.ServiceLoader`. This service mechanism has been primarily used by JDK itself and third-party frameworks and libraries. A typical example of a service provider is a JDBC driver. Each JDBC driver should provide the implementation of the service interface `java.sql.Driver`. The driver's JAR file should have the provider configuration file `java.sql.Driver` in the directory `META-INF/services`. For example, the file `java.sql.Driver` in the JAR file of Apache Derby (`https://db.apache.org/derby/`) has this content:

```
org.apache.derby.jdbc.AutoloadedDriver
```

`org.apache.derby.jdbc.AutoloadedDriver` is the name of the implementation class of the service interface `java.sql.Driver`.

Before Java 9, `ServiceLoader` scanned the class path to locate provider implementations for a given service interface. In Java 9, the module descriptor `module-info.java` has specific declarations for service consumers and providers.

Listing 2-12 shows the service interface `PersistenceService` in the module `io.vividcode.store.common`. The interface `PersistenceService` has a single method `save()` to save `Persistable` objects.

Listing 2-12. Service Interface `PersistenceService`

```
package io.vividcode.store.common;

public interface PersistenceService {
  void save(final Persistable persistable) throws PersistenceException;
}
```

The module `io.vividcode.store.common.persistence` consumes this service interface. In its file `module-info.java` in Listing 2-13, the new keyword `uses` is used to declare the consumption of the service interface `io.vividcode.store.common.PersistenceService`.

Listing 2-13. Module Declaration of `io.vividcode.store.common.persistence`

```
module io.vividcode.store.common.persistence {
  requires slf4j.api;
  requires transitive io.vividcode.store.common;
  exports io.vividcode.store.common.persistence;
  uses io.vividcode.store.common.PersistenceService;
}
```

Now you can use `ServiceLoader` to look up providers of this service interface. In Listing 2-14, the method `ServiceLoader.load()` creates a new service loader for the service type `PersistenceService`, then you use the method `findFirst()` to get the first available service provider. If a service provider is found, use its method `save()` to save `Persistable` objects.

Listing 2-14. Using `ServiceLoader` to Look Up Providers

```
public class DataStore<T extends Persistable> {

  private final Optional<PersistenceService> persistenceServices;

  public DataStore() {
    this.persistenceServices = ServiceLoader
        .load(PersistenceService.class)
        .findFirst();
  }

  public void save(final T object) throws PersistenceException {
    if (this.persistenceServices.isPresent()) {
      this.persistenceServices.get().save(object);
    }
  }
}
```

The provider of service interface `PersistenceService` is in the module `io.vividcode.store.filestore`. In this module's declaration of Listing 2-15, provides `io.vividcode.store.common.PersistenceService` with `io.vividcode.store.filestore.FileStore` means this module provides the implementation of the service interface `PersistenceService` using the class `io.vividcode.store.filestore.FileStore`. The implementation of `FileStore` is quite simple and you can check the source code for its implementation.

Listing 2-15. Module Declaration of `io.vividcode.store.filestore`

```
module io.vividcode.store.filestore {
  requires io.vividcode.store.common.persistence;
  requires slf4j.api;
  provides io.vividcode.store.common.PersistenceService
      with io.vividcode.store.filestore.FileStore;
}
```

Qualified Exports

When you are using `exports` to export a package in the module declaration, this package is visible to all modules that use `requires` to require it. Sometimes you may want to limit the visibility of certain packages to some modules only. Consider this example: a package was initially designed to be public to other modules, but this package was deprecated in later versions. Legacy code that uses the old version of this package should continue to work after migrating to Java 9, while new code should use the new versions. The package should only be visible to modules of the legacy code that still use the old version. This is done by using the `to` clause in `exports` to specify the names of modules that should have access.

Listing 2-16 shows the module declaration of the JDK module `java.rmi`. You can see that package `com.sun.rmi.rmid` is only visible to module `java.base` and package `sun.rmi.server` is only visible to modules `jdk.management.agent`, `jdk.jconsole`, and `java.management.rmi`.

Listing 2-16. Module Declaration of JDK Module `java.rmi`

```
module java.rmi {
  requires java.logging;

  exports java.rmi.activation;
  exports com.sun.rmi.rmid to java.base;
  exports sun.rmi.server to jdk.management.agent,
      jdk.jconsole, java.management.rmi;
  exports javax.rmi.ssl;
  exports java.rmi.dgc;
  exports sun.rmi.transport to jdk.management.agent,
      jdk.jconsole, java.management.rmi;
  exports java.rmi.server;
  exports sun.rmi.registry to jdk.management.agent;
  exports java.rmi.registry;
  exports java.rmi;

  uses java.rmi.server.RMIClassLoaderSpi;
}
```

Opening Modules and Packages

In the module declaration, you can add the modifier `open` before `module` to declare it as an open module. An open module grants compile time access to explicitly exported packages only, but it grants access to types in all its packages at runtime. It also grants reflective access to all types in all packages. All types include private types and their private members. If you use the reflection API and suppress Java language access checks—using the method `setAccessible()` of `AccessibleObject`, for example—you can access private types and members in open modules.

You can also use `opens` to open packages to other modules. You can access open packages using the reflection API at runtime. Just like open modules, all types in an open package and all their members can be reflected by the reflection API. You can also qualify open packages using `to`, which has a similar meaning in qualified exports.

The declaration of module E in Listing 2-17 marks it as an open module.

Listing 2-17. Declaration of an Open Module

```
open module E {
  exports etest;
}
```

The declaration of module F in Listing 2-18 opens two packages. It's possible to open packages that don't exist in the module. It's also possible to open packages to nonexistent modules. The compiler generates warnings in these cases, which is the same as exporting packages to non-existent modules.

Listing 2-18. Module Declaration That Uses Open Packages

```
module F {
  opens ftest1;
  opens ftest2 to G;
}
```

Open modules and open packages are provided mainly for resolving backward compatibility issues. You may need to use them when migrating legacy code that relies on reflections to work.

Working with Existing Code

For developing new projects in Java 9, the concepts in the module declaration are enough. But if you need to work with existing code written before Java 9, you need to understand unnamed modules and automatic modules.

Unnamed Modules

From previous sections, you can see that Java 9 has a strict constraint on how types can be accessed across module boundaries. If you are creating a brand-new application targeting Java 9, then you should use the new module system. However, Java 9 still supports running all applications written prior to Java 9. This is done with the help of the *unnamed modules*.

When the module system needs to load a type whose package is not defined in any module, it will try to load it from the class path. If the type is loaded successfully, then this type is considered to be a member of a special module called the *unnamed module*. The unnamed module is special because it reads all other named modules and exports all of its packages.

When a type is loaded from the class path, it can access exported types of all other named modules, including built-in platform modules. For Java 8 applications, all types of this application are loaded from the class path, so they are all in the same unnamed module and have no issues accessing each other. The application can also access platform modules. That's why Java 8 applications can run on Java 9 without changes.

The unnamed module exports all of its packages, but code in other named modules cannot access types in the unnamed module, and you cannot use `requires` to declare the dependency—there is no name for you to use to reference it. This constraint is necessary; otherwise we lose all the benefits of the module system and go back to the dark old days of messy class path. The unnamed module is designed purely for backward compatibility. If a package is defined in both a named and unnamed module, the package in the unnamed module is ignored. Unexpected duplicate packages in the class path won't interfere with the code in other named modules.

Automatic Modules

Since Java 9 is backward compatible to run existing Java applications, it's not necessary to upgrade existing applications to use modules. However, it's recommended that you upgrade to take advantages of the new module system.

The recommended approach to migrate existing applications is to do it in a bottom-up way; that is, start with the modules that are in the bottom of the entire dependency tree. For example, in an application that has three modules/subprojects A, B, and C with a dependency tree like `A -> B -> C`, you start by migrating C to a module first, then B, and then A. After C is migrated to a module, A and B, which are currently in the unnamed module, can still access types in C because the unnamed module reads all named modules. Then you migrate B to a named module and declare it to require the migrated named module C. Finally, migrate A to a named module; at this point, the whole application has been migrated successfully.

It's not always possible to do the bottom-up migration. Some libraries may be maintained by third parties and you cannot control when these libraries will be migrated to modules. But you still want to migrate modules that depend on those third-party libraries. You cannot just migrate those modules, while leaving third-party libraries in the class path, however, because named modules cannot read the unnamed module. What you can do is put those library JAR files in the module path and turn them into *automatic modules*.

Other than explicitly created named modules, automatic modules are created implicitly from normal JAR files. There is no module declaration in these JAR files. The name of an automatic module comes from the attribute `Automatic-Module-Name` in the manifest file `MANIFEST.MF` of the JAR file, or it is derived from the name of the JAR file. Other named modules can declare dependency upon this JAR file using the name of the automatic module. It's recommended that you add the attribute `Automatic-Module-Name` in the manifest, because it is more reliable than the derived module names from JAR file names.

Automatic modules are special in many ways:

- An automatic module reads all other named modules.
- An automatic module exports all of its packages.
- An automatic module reads the unnamed modules.
- An automatic module grants transitive readability to all other automatic modules.

Automatic modules are the bridge between the class path and explicitly named modules. The goal is to migrate all existing modules/subprojects/libraries to Java 9 named modules. However, during the migrating process, you can always add them to the module path to use them as automatic modules.

In Listing 2-19, the class `MyD` use the class `com.google.common.collect.Lists` from Guava (`https://github.com/google/guava`) version 21.0.

Listing 2-19. Class `MyD` Using the Guava Library

```
package dtest;

import com.google.common.collect.Lists;

public class MyD {
  public static void main(String[] args) {
    System.out.println(Lists.newArrayList("Hello", "World"));
  }
}
```

At the time of writing, Guava has not yet been migrated as a Java 9 module. In the module declaration of D in Listing 2-20, you can use `requires guava` to declare dependency upon it. `guava` is the name of the automatic module for Guava, which is derived from the JAR file name.

Listing 2-20. Declaring Dependency Upon an Automatic Module

```
module D {
  requires guava;
}
```

When compiling the code in Listing 2-20, you can use the command-line option `--module-path` in `javac` to add Guava's JAR file `guava-21.0.jar` in the directory `~/libs` to the module path.

```
$ javac --module-path ~/libs <src_path>/*.java <src_path>/dtest/*.java
```

JDK Tools

After you finish the source code of a project's modules, you need to compile and run these modules. Most of time, you'll use IDEs for development and testing, so you can leave the compilation and execution of the project to the IDE. However, you can still use the JDK tools `javac` and `java` directly to compile and run the code, respectively. Understanding details about these JDK tools can help you to understand the whole life cycle of modules. However, it takes time for IDEs and build tools to improve their support for JDK 9. The process may be slow, so you may still need to use these JDK tools for some tasks. When migrating to Java 9, you may encounter various problems related to the module system. If you have a deep understanding of these tools, you can easily find the root cause and solve these problems.

Some of these JDK tools have been upgraded to support module-related options, while others are new in Java 9. These tools support some common command-line options related to common concepts in the module system. These tools can be used in different phases:

- *Compile time*: Use `javac` to compile Java source code into class files.
- *Link time*: The new optional phase introduced in Java 9. Use `jlink` to assemble and optimize a set of modules and their transitive dependencies to create a custom runtime image.
- *Runtime*: Use `java` to launch the JVM and execute the byte code.

Other utility tools have no special phases to work with.

Module Paths

The module system uses module paths to locate modules defined in different artifacts. A *module path* is a sequence of paths to a module definition or directories containing module definitions. A *module definition* can be either a module artifact or a directory that contains exploded contents of a module. Module paths are searched based on their order in a sequence to find the first definition that defines a particular module. The module system also uses module paths to resolve dependencies. If the module system cannot find the definition for a dependent module, or if it finds two definitions in the same directory that define modules with the same name, it signals an error and exits. Module paths are separated with platform-dependent path separators: a semicolon (`;`) for Windows and a colon (`:`) for macOS and Linux.

Different types of module paths are used in different phases; see Table 2-2. As shown in this table, different command-line options for module paths can be applied to multiple phases. The order for each module path defines the search sequence when multiple module paths exist. For example, at compile time,

using the `javac`, for example, all four types of module paths can be used. The module system checks module paths specified in `--module-source-path` first, then it checks `--upgrade-module-path`, then `--system`, and finally `--module-path` or `-p`.

Table 2-2. *Different Types of Module Paths*

Order	Name	Command-Line Option	Applied Phase	Description
1	Compilation module path	`--module-source-path`	Compile time	Source code of modules
2	Upgrade module path	`--upgrade-module-path`	Compile time and runtime	Contains compiled modules used to replace upgradeable modules in the environment
3	System modules	`--system`	Compile time and runtime	Compiled modules in the environment
4	Application module path	`--module-path` or `-p`	All phases	Compiled application modules

The system modules and modules definitions found on the module paths are referred to as the set of *observable modules*, which are very important in the module resolution process. If a module to resolve doesn't exist in the set of observable modules, the module system signals an error and exists.

Module Version

Although there is no version-related configuration in the module declaration, it's still possible to record version information for a module. It's recommended that you record the module version following the scheme of semantic versioning (`http://semver.org/`). Build tools like Maven or Gradle should record the version information automatically, so you don't need to worry about it unless you're using `javac` or `jar` tools directly. The important thing to know is that the module system ignores version information when searching for modules. If a module path contains multiple definitions of modules with the same name but different versions, the module system still treats this as an error. Module name is the only thing that matters when resolving modules.

You can specify module version using the option `--module-version`.

The Main Module

The main module can be specified using the option `--module` or `-m`. At runtime, the main module contains the main class to run. If the main class is recorded in the module's declaration, then specifying only the module name is enough. Otherwise, you need to use `<module>/<mainclass>` to specify the module and main class, for example, `com.mycompany.mymodule/com.mycompany.mymodule.Main`.

At compile time, `--module` or `-m` specifies the only module to compile.

Root Modules

The set of observable modules defines all the possible modules that can be resolved. However, not all observable modules are required at runtime. The module system starts the resolution process from a set of root modules, and it constructs a module graph by resolving the transitive closure of the dependencies of these root modules against the set of observable modules. It's possible that not all observable modules are resolved, and only the observable modules are resolvable.

The module system has some rules it uses to select the default root modules. When you're compiling or running code in the unnamed module, that is, code that predates Java 9, the default set of root modules for the an unnamed module includes JDK system modules and application modules. If the `java.se` module exists, it will be the only JDK system module to include. Otherwise, every `java.*` module that unconditionally exports at least one package is a root module. Every non-`java.*` module that unconditionally exports at least one package is also a root module.

When you're compiling or running Java 9 code, the default set of root modules depends on the phase:

- At compile time, it's the set of modules to compile.
- At link time, it's empty.
- At runtime, it's the application's main module.

The set of root modules can be extended to include extra modules using the option `--add-modules`. The value of this option is a comma-separated list of module names. There are also three special values of this option.

- `ALL-DEFAULT`: Add the default set of root modules for the unnamed module.
- `ALL-SYSTEM`: Add all system modules.
- `ALL-MODULE-PATH`: Add all observable modules found on the module paths.

Limiting the Observable Modules

It's possible to limit the observable modules using the option `--limit-modules`. After you use this option, the set of observable modules is the transitive closure of those specified modules plus the main module and any modules specified via the option `--add-modules`. The value of this option is also a comma-separated list of module names.

Upgrading the Module Path

The option `--upgrade-module-path` specifies the upgrade module path. This path contains modules that you can use to upgrade modules that are built-in into the environment. This module path supersedes the existing extension mechanism (http://docs.oracle.com/javase/8/docs/technotes/guides/extensions/index.html).

Whether a system module is upgradable is clearly documented in the file `module-info.java`. For example, modules `java.xml.bind` and `java.xml.ws` are upgradable.

Increasing Readability and Breaking Encapsulation

The module system is all about encapsulation. But sometimes you'll still want to break encapsulation when you're dealing with legacy code or running tests. You can use several command-line options to break encapsulation.

- `--add-reads module=target-module(,target-module)*` updates the source module to read the target module. The target module can be `ALL-UNNAMED` to read all unnamed modules.
- `--add-exports module/package=target-module(,target-module)*` updates the source module to export the package to the target module. This will add a qualified export of the package from the source module to the target module. The target module can be `ALL-UNNAMED` to export to all unnamed modules.

- `--add-opens module/package=target-module(,target-module)*` updates the sources module to open the package to the target module. This will add a qualified open of the package from the source module to the target module.
- `--patch-module module=file(;file)*` overrides or augments a module with classes and resources in JAR files or directories. `--patch-module` is useful when you're running tests that may need to temporarily replace the contents of a module.

Now that I've introduced the basic concepts, let's discuss these JDK tools.

javac

`javac` supports the following options related to modules. Meanings of these options have been explained in earlier sections of this chapter.

- `--module` or `-m`
- `--module-path` or `-p`
- `--module-source-path`
- `--upgrade-module-path`
- `--system`
- `--module-version`
- `--add-modules`
- `--limit-modules`
- `--add-exports`
- `--add-reads`
- `--patch-module`

I'll now use the modules discussed in the "Transitive Dependencies" as examples of how to use `javac`. Let's say you have a directory that contains the source code of these three modules. Each module has its own subdirectory. You can compile a single module as shown here. Module C has no dependencies, so it can be compiled directly.

```
$ javac -d ~/Downloads/modules_output/C C/**/*.java
```

To compile module B, you can use `-p` to provide the compiled module C since module B requires module C. You should also use `-p` when third-party libraries are required.

```
$ javac -d ~/Downloads/modules_output/B -p ~/Downloads/modules_output B/**/*.java
```

Since you have source code for all the modules, you can simply compile them all.

```
$ javac -d ~/Downloads/modules_output --module-source-path . **/*.java
```

You can also compile a single module using `-m. --module-source-path` is required to specify the module source path when you're using `-m`.

```
$ javac -d ~/Downloads/modules_output --module-source-path . -m B
```

jlink

To run Java applications, you need to have JRE or JDK installed first. As I mentioned earlier, before Java 9, there was no easy way to customize the JRE or JDK to include only necessary contents. Even a simple "Hello World" application requires the full-sized JRE to run. After JDK is modularized, however, it's possible to create your own Java runtime image that only contains the system modules required by the application, which can reduce the size of the JRE image.

You can use `jlink` to build a custom image. If you're given a module `demo.simple` that has a class `test.Main`, you can use it to print out `Hello World!`. To build your image, create the modular JAR file `demo.simple-1.0.0.jar` and set the main class. Listing 2-21 shows the command you can use to create a custom image using `jlink`. The path `<module_dir>` contains the artifact of the module `demo.simple. <JDK_PATH>/jmods` is the path to JDK modules.

Listing 2-21. Using `jlink` to Create Custom Images

```
$ jlink -p <module_dir>:<JDK_PATH>/jmods \
  --add-modules demo.simple \
  --output <output_dir> \
  --launcher simple=demo.simple
```

The `jlink` tool creates a new runtime image in the output directory. You can run the executable file `simple` in the `bin` directory to run your application. The size of this customized runtime image on macOS is only 36.5MB. Figure 2-1 shows the content of the custom runtime image.

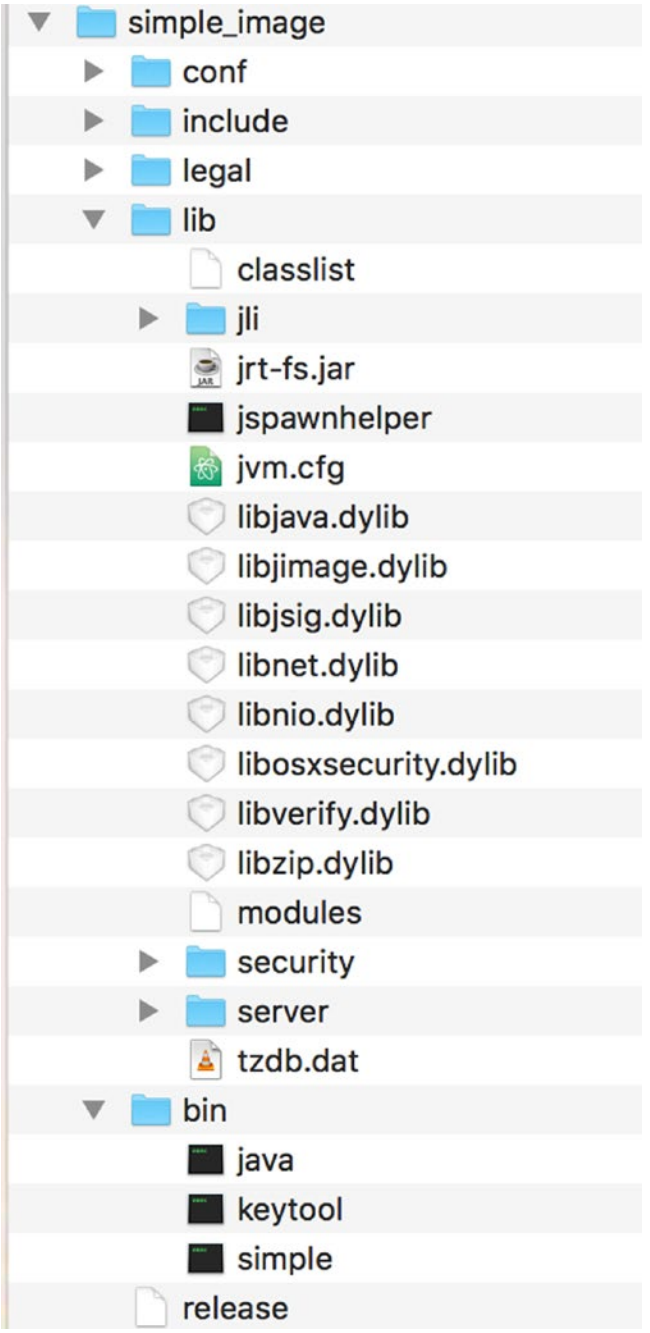

Figure 2-1. *Contents of the custom runtime image*

Table 2-3 shows the options of `jlink`.

Table 2-3. *Options of* `jlink`

Option	Description
`--module-path` or `-p`	See "*Module Paths*".
`--add-modules`	See "*Root Modules*".
`--output`	The output directory.
`--launcher`	The launcher command to run a module or a main class in a module. If the module is created with `--main-class` to specify the main class, then using the module name is enough. Otherwise, the main class can be specified using `<module>/<mainclass>`.
`--limit-modules`	See "Limiting the Observable Modules".
`--bind-services`	Performs the full service binding.
`--compress=<0|1|2>` or `-c`	Enables compression. Possible values are `0`, `1`, and `2`. `0` means no compression, `1` means constant string sharing, and `2` means using ZIP compression.
`--endian`	Byte order of the generated image. Possible values can be `little` or `big`. The default value is `native`.
`--no-header-files`	Excludes header files in the image.
`--no-man-pages`	Excludes man pages.
`-G` or `--strip-debug`	Strip debug information.
`--ignore-signing-information`	Suppresses the fatal error when linking signed module JAR files. The signature-related files of the signed module JAR files are not copied to the generated image.
`--suggest-providers [name, ...]`	Suggests providers that implement the given service types.
`--include-locales=langtag[,langtag]*`	Includes the list of locales. The module `jdk.localedata` must be added when using this option.
`--exclude-files=pattern-list`	Excludes specified files.
`--verbose` or `-v`	Enable verbose tracing output.

A major limitation of `jlink` is that it doesn't support automatic modules, which means third-party libraries cannot be linked using `jlink`. For example, when you're trying to use `jlink` to create an image for the sample application, it gives following error.

```
Error: module-info.class not found for slf4j.api module
```

From this error message, you know that it requires the `module-info.class` for the module `slf4j.api`, but the SLF4J library has not been migrated to Java 9 yet, so you cannot include it in the image.

java

The java command supports the following options related to modules:

- `--module-path` or `-p`
- `--upgrade-module-path`
- `--add-modules`
- `--limit-modules`
- `--list-modules`: Lists the observable modules.
- `--describe-module` or `-d`: Describe the module.
- `--validate-modules`: Validates all modules. Can be used to find conflicts and errors.
- `--show-module-resolution`: Shows module resolution results during start.
- `--add-exports`
- `--add-reads`
- `--add-opens`
- `--patch-module`

For the sample application, after using the Maven assembly plugin to copy all libraries and module artifacts into one directory, you can start the application using following command:

```
$ java -p <path> -m io.vividcode.store.runtime/io.vividcode.store.runtime.Main
```

If you record the main class in the module artifact, you can simply use `java -p <path> -m io.vividcode.store.runtime` to run the application.

The `--list-modules` option is very useful when you're debugging module resolution issues, because it can list all the observable modules. For example, you can use `java --list-modules` to list all JDK system modules. If you use `-p` to add module paths, the output also includes modules found in the module paths. The option `--show-module-resolution` is also very useful for solving module resolution issues.

jdeps

You can use the `jdeps` tool to analyze the dependencies of specified modules. The following command prints out the module descriptor of `io.vividcode.store.runtime`, the resulting module dependencies after analysis, and the graph after transition reduction; see Listing 2-22. It also identifies any unused qualified exports.

```
$ jdeps --module-path <path> --check io.vividcode.store.runtime
```

Listing 2-22. Output of `jdeps` Using `--check`

```
io.vividcode.store.runtime (file:///<path>/runtime-1.0.0-SNAPSHOT.jar)
  [Module descriptor]
  requires io.vividcode.store.filestore;
  requires io.vividcode.store.product.persistence;
  requires mandated java.base (@9);
  requires slf4j.simple;
```

```
  [Suggested module descriptor for io.vividcode.store.runtime]
  requires io.vividcode.store.common;
  requires io.vividcode.store.product;
  requires io.vividcode.store.product.persistence;
  requires mandated java.base;
  [Transitive reduced graph for io.vividcode.store.runtime]
  requires io.vividcode.store.product.persistence;
  requires mandated java.base;
```

To use the `--check` option, you need to know the module name first. If you only have a JAR file, you can use the `--list-deps` option or `--list-reduced-deps`.

```
$ jdeps --module-path <path> --list-deps <path>/runtime-1.0.0-SNAPSHOT.jar
```

Listing 2-23 shows the output of this command.

Listing 2-23. Output of `jdeps` Using `--list-deps`

```
io.vividcode.store.common
io.vividcode.store.product
io.vividcode.store.product.persistence
java.base
```

The difference between `--list-deps` and `--list-reduced-deps` is that the result of using the `--list-reduced-deps` option doesn't include implicit read edges in the module graph.

```
$ jdeps --module-path <path> --list-reduced-deps <path>/runtime-1.0.0-SNAPSHOT.jar
```

Listing 2-24 shows the output of this command.

Listing 2-24. Output of `jdeps` Using `--list-reduced-deps`

```
io.vividcode.store.product.persistence
```

Another handy feature of `jdeps` is that it can generate graphviz (http://graphviz.org/) DOT files to visualize module dependencies in graphs.

```
$ jdeps --module-path <module_path> \
    --dot-output <dot_output_path> -m io.vividcode.store.runtime
```

The output directory contains a DOT file `summary.dot` for all modules and a DOT file `io.vividcode.store.runtime.dot` for the module. Then you can turn DOT files into PNG files using the following command. Make sure you have graphviz installed first.

```
$ dot -Tpng <dot_output_path>/summary.dot -o <dot_png_output_path>/summary.png
```

You can also use `jdeps` to process multiple JAR files to generate the module dependencies diagram of the whole project. The generated DOT file `summary.dot` contains all the modules. For the sample application, use Maven to copy all module artifacts and third-party libraries into different directories. You can do this easily using Maven's dependency plugin. Then you can generate the DOT files using following

command. (<third_party-libs-path> is the path of third-party libraries, while <modules_path> is the path of module artifacts.)

```
$jdeps --module-path <third_party-libs-path> \
    --dot-output <dot_output_path> \
  <modules_path>/*.jar
```

You can then turn the file summary.dot into a PNG file, see Figure 2-2 for the generated module dependencies diagram.

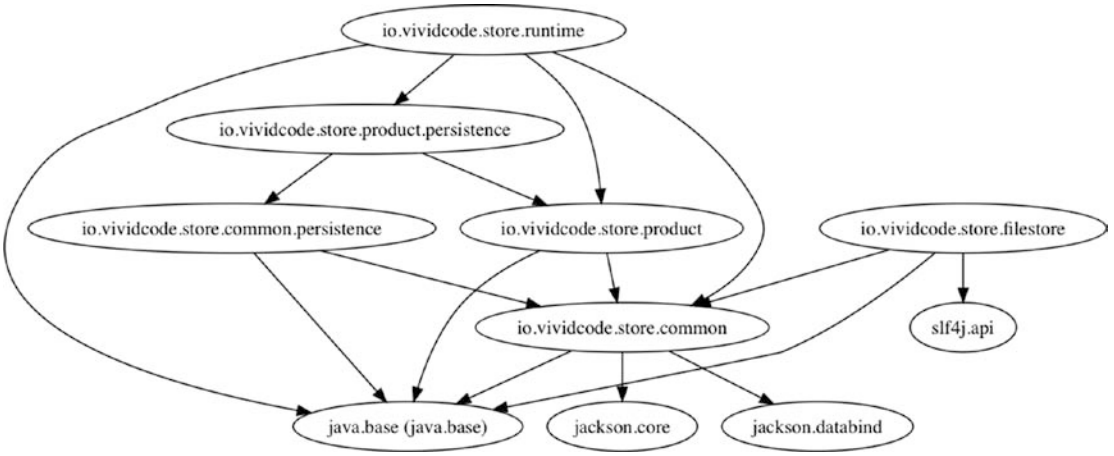

Figure 2-2. *Generated module dependencies diagram*

jdeps can also generate the module declaration file module-info.java from JAR files. The options --generate-module-info and --generate-open-module can be used for normal modules and open modules, respectively. For example, you can use the following command to generate the file module-info.java for jackson-core-2.8.7.jar.

```
$ jdeps --generate-module-info <output_dir> <path>/jackson-core-2.8.7.jar
```

The generated module-info.java is shown in Listing 2-25. It simply exports all packages and adds service providers. You can use the generated module-info.java file as a starting point when migrating legacy JAR files to Java 9 modules.

Listing 2-25. Generated module-info.java of jackson-core-2.8.7.jar

```
module jackson.core {
  exports com.fasterxml.jackson.core;
  exports com.fasterxml.jackson.core.base;
  exports com.fasterxml.jackson.core.filter;
  exports com.fasterxml.jackson.core.format;
  exports com.fasterxml.jackson.core.io;
  exports com.fasterxml.jackson.core.json;
  exports com.fasterxml.jackson.core.sym;
```

```
  exports com.fasterxml.jackson.core.type;
  exports com.fasterxml.jackson.core.util;

  provides com.fasterxml.jackson.core.JsonFactory with
    com.fasterxml.jackson.core.JsonFactory;

}
```

Module Java API

When the JVM starts, it resolves the application's main module against the observable modules. The result of this resolution process is a *readability graph*, which is a directed graph with nodes representing resolved modules and edges representing the readability among the modules. Then the JVM creates a module layer, which consists of runtime modules defined from those resolved modules. The module system has Java API for applications to interact with it.

ModuleFinder

Let's start with the module resolution process. To resolve a module, you first need to find the module definition. The interface `ModuleFinder` is responsible for locating modules. There are three different ways to create instances of `ModuleFinder` with static methods in `ModuleFinder`; see Table 2-4.

Table 2-4. *Static Methods of `ModuleFinder`*

Method	Description
`ofSystem()`	Creates a `ModuleFinder` that locates system modules
`of(Path… entries)`	Creates a `ModuleFinder` that locates modules on the file system by searching a sequence of paths specified by `entries`
`compose(ModuleFinder... finders)`	Creates a `ModuleFinder` by composing a sequence of zero or more `ModuleFinders`

Once a `ModuleFinder` is created, you can use its method `Optional<ModuleReference> find(String name)` to find a module by name. The class `ModuleReference` is a reference to the module's contents. The method `Set<ModuleReference> findAll()` returns a set that contains all `ModuleReferences` that can be located by this `ModuleFinder` object. Normally you don't need to use `ModuleFinders` directly, but you should use them to create configurations for readability graphs.

The modules located by `ModuleFinders` are represented as `ModuleReference`. Table 2-5 shows the methods of `ModuleReference`.

Table 2-5. *Methods of `ModuleReference`*

Method	Description
`ModuleDescriptor descriptor()`	Returns the `ModuleDescriptor` that describes the module
`Optional<URI> location()`	Returns the location of the module's content as a `URI`
`ModuleReader open()`	Opens the module's content for reading

Listing 2-26 shows a helper class to create `ModuleFinder` and `ModuleReference` objects. The `ModuleFinder` object finds modules in the resource path `/modules`. This path contains module artifacts of the sample application. The method `getModuleReference()` uses the `ModuleFinder` to find a `ModuleReference` by name. Since I am sure that the module to find must exist, here I simply use the method `get()` of the returned `Optional<ModuleReference>` to retrieve the `ModuleReference` directly.

Listing 2-26. Support Class for Module Tests

```
public class ModuleTestSupport {

  public static final String MODULE_NAME = "io.vividcode.store.common";

  public static ModuleFinder getModuleFinder() throws URISyntaxException {
    return ModuleFinder.of(
      Paths.get(
        ModuleDescriptorTest.class.getResource("/modules").toURI()));
  }

  public static ModuleReference getModuleReference() throws URISyntaxException {
    return getModuleFinder().find(MODULE_NAME).get();
  }
}
```

Listing 2-27 is the test of `ModuleFinder`. Here I verify the results of methods `find()` and `findAll()`.

Listing 2-27. `ModuleFinder` Test

```
public class ModuleFinderTest {

  @Test
  public void testFindModule() throws Exception {
    final ModuleFinder moduleFinder = ModuleTestSupport.getModuleFinder();
    assertTrue(moduleFinder.find(ModuleTestSupport.MODULE_NAME).isPresent());
    final Set<ModuleReference> allModules = moduleFinder.findAll();
    assertEquals(4, allModules.size());
  }
}
```

ModuleDescriptor

From the `ModuleReference`, you can get the `ModuleDescriptor` that describes modules. Table 2-6 shows the methods of `ModuleDescriptor`.

Table 2-6. *Methods of* `ModuleDescriptor`

Method	Description
`Set<ModuleDescriptor.Exports> exports()`	Returns the set of `ModuleDescriptor.Exports` that represent the exported packages
`Set<ModuleDescriptor.Opens> opens()`	Returns the set of `ModuleDescriptor.Opens` that represent the open packages.
`Set<ModuleDescriptor.Requires> requires()`	Returns the set of `ModuleDescriptor.Requires` that represent the module's dependencies
`Set<ModuleDescriptor.Provides> provides()`	Returns the set of `ModuleDescriptor.Provides` that represent services provided by this module
`Set<String> uses()`	Returns the set of services this module uses
`String name()`	Returns the module's name
`Optional<String> mainClass()`	Returns the module's main class
`Set<ModuleDescriptor.Modifier> modifiers()`	Returns the set of `ModuleDescriptor.Modifier` that represent module modifiers
`boolean isAutomatic()`	Checks if the module is automatic
`boolean isOpen()`	Checks if the module is open
`Optional<ModuleDescriptor.Version> version()`	Returns the module's version

In Listing 2-28, I use a test to verify various information retrieved from the `ModuleDescriptor`.

Listing 2-28. `ModuleDescriptor` Test

```
public class ModuleDescriptorTest {

  @Test
  public void testModuleDescriptor() throws Exception {
    final ModuleReference reference = ModuleTestSupport.getModuleReference();
    assertNotNull(reference);
    final ModuleDescriptor descriptor = reference.descriptor();
    assertEquals(ModuleTestSupport.MODULE_NAME, descriptor.name());
    assertFalse(descriptor.isAutomatic());
    assertFalse(descriptor.isOpen());
    assertEquals(1, descriptor.exports().size());
    assertEquals(2, descriptor.packages().size());
    assertTrue(descriptor.requires().stream().map(Requires::name)
      .anyMatch(Predicate.isEqual("jackson.core")));
    assertTrue(descriptor.uses().isEmpty());
    assertTrue(descriptor.provides().isEmpty());
  }
}
```

The methods in Table 2-6 are used to query information of existing `ModuleDescriptor` objects. To create `ModuleDescriptor` objects, you can either use the builder class `ModuleDescriptor.Builder`, or read from the binary form of module descriptors. The static methods `newModule(String name)`, `newOpenModule(String name)`, and `newAutomaticModule(String name)` can create `ModuleDescriptor`.

Builder objects to build a normal, open, and automatic module, respectively. ModuleDescriptor.Builder has different methods to programmatically configure various components of a module descriptor. Listing 2-29 shows a simple test of ModuleDescriptor.Builder.

The static method read() has different overloads to read the binary form of a module declaration from an InputStream or a ByteBuffer. It's possible that the binary form doesn't include a set of packages in the module, if it doesn't, you can pass an extra argument of type Supplier<Set<String>> to provide the packages.

Listing 2-29. ModuleDescriptor.Builder Test

```
@Test
public void testModuleDescriptorBuilder() {
  final ModuleDescriptor descriptor = ModuleDescriptor.newModule("demo")
      .exports("demo.api")
      .exports("demo.common")
      .mainClass("demo.Main")
      .version("1.0.0")
      .build();
  assertEquals(2, descriptor.exports().size());
  assertTrue(descriptor.mainClass().isPresent());
  assertEquals("demo.Main", descriptor.mainClass().get());
  assertTrue(descriptor.version().isPresent());
  assertEquals("1.0.0", descriptor.version().get().toString());
}
```

ModuleReader returned by ModuleReference.open() reads resources in a module. A resource is identified by a path string separated by /, for example, com/mycompany/mymodule/Main.class. Table 2-7 shows the methods of ModuleReader.

Table 2-7. *Methods of ModuleReader*

Method	Description
Stream<String> list()	Lists the names of all resources in the module
Optional<URI> find(String name)	Finds a resource by its name
Optional<InputStream> open(String name)	Opens a resource for reading
Optional<ByteBuffer> read(String name)	Reads a resource's contents as a ByteBuffer
void release(ByteBuffer bb)	Releases a ByteBuffer
void close()	Closes this ModuleReader

In Listing 2-30, I use ModuleReader to read the content of module-info.class into a ByteBuffer and then create a new ModuleDescriptor from this ByteBuffer.

Listing 2-30. ModuleReader Test

```
public class ModuleReaderTest {

  @Test
  public void testModuleReader() throws Exception {
    final ModuleReference reference = ModuleTestSupport.getModuleReference();
```

```
    assertNotNull(reference);
    final ModuleReader reader = reference.open();
    final Optional<ByteBuffer> byteBuffer = reader.read("module-info.class");
    assertTrue(byteBuffer.isPresent());
    final ModuleDescriptor descriptor = ModuleDescriptor.read(byteBuffer.get());
    assertEquals(ModuleTestSupport.MODULE_NAME, descriptor.name());
  }
}
```

Configuration

The readability graph is represented as an object of the class `java.lang.module.Configuration`. A configuration may have parent configurations. Configurations can be organized in a hierarchy. `Configuration` has methods to create and query the readability graph.

`Configurations` are created using the methods `resolve()` and `resolveAndBind()`. Both methods return a new `Configuration` object representing the result of module resolution. `Configuration` also has static versions of these two methods. For static methods, you need to provide the parameter of type `List<Configuration>` as the parent configurations of the created `Configuration`. For instance methods, the current `Configuration` is the only parent configuration of the created `Configuration`. The following are the methods you need to create `Configurations`:

- `Configuration resolve(ModuleFinder before, ModuleFinder after, Collection<String> roots)`
- `Configuration resolveAndBind(ModuleFinder before, ModuleFinder after, Collection<String> roots)`
- `static Configuration resolve(ModuleFinder before, List<Configuration> parents, ModuleFinder after, Collection<String> roots)`
- `static Configuration resolveAndBind(ModuleFinder before, List<Configuration> parents, ModuleFinder after, Collection<String> roots)`

The method `resolveAndBind()` takes the same parameters as the method `resolve()`. The difference is that the resolved modules of `resolveAndBind()` also includes modules introduced by service dependencies.

The following are the possible parameters of the preceding methods:

- `roots`: The list of root module names.
- `parents`: The list of parent configurations.
- `before` and `after`: Two `ModuleFinders` to locate modules. The `ModuleFinder before` is used to locate root modules. If a module cannot be found, parent configurations are used to locate it using the method `findModule()`. If the module still cannot be found, then the `ModuleFinder after` is used to locate it. Transitive dependencies are located following the same search order.

When a root module or transitive dependency is located in a parent configuration, the resolution process ends for this module and the module is not included in the resulting configuration.

Listing 2-31 shows the example of creating a new `Configuration` by finding modules in the given path.

Listing 2-31. Create New `Configuration` Objects Using `resolve()`

```
public Configuration resolve(final Path path) {
  return Configuration.resolve(
    ModuleFinder.of(path),
    List.of(Configuration.empty()),
    ModuleFinder.ofSystem(),
    List.of("io.vividcode.store.runtime")
  );
}
```

The module resolution process may fail for various reasons. If a root module or a direct or transitive dependency, is not found, or any error occurs when trying to find a module, the methods `resolve()` and `resolveAndBind()` throw the `FindException`. After all modules have been found, the readability graph is computed, and the consistency of module exports and service use is checked. Consistency checks include checking for cyclic module dependencies, duplicate module reads, duplicate package exports, and invalid service use. `ResolutionException` is thrown when any consistency check fails.

Table 2-8 shows other methods of `Configuration`.

Table 2-8. *Methods of `Configuration`*

Method	Description
`static Configuration empty()`	Returns the empty configuration
`List<Configuration> parents()`	Returns the list of this configuration's parent configurations, in the search order
`Optional<ResolvedModule> findModule(String name)`	Finds a resolved module by name
`Set<ResolvedModule> modules()`	Returns the set of all resolved modules in this configuration

For each `ResolvedModule`, the `reads()` method returns a set of `ResolvedModules` that it reads; see Table 2-9 for other methods.

Table 2-9. *Methods of `ResolvedModule`*

Method	Description
`Configuration configuration()`	Returns the `Configuration` that this resolved module belongs to
`String name()`	Returns the module name
`Set<ResolvedModule> reads()`	Returns the set of `ResolvedModules` that this resolved module reads
`ModuleReference reference()`	Returns the `ModuleReference` to this module's content

Listing 2-32 shows how to print out the readability graph. The method `sortedModules()` sorts `ResolvedModules` by name. In the method `printLayer()`, I get all `ResolvedModules` in the configuration and print out their names. For each `ResolvedModule`, I also print out the `ResolvedModules` it reads.

Listing 2-32. Printing Out the Readability Graph

```
public class ConfigurationPrinter {

  public void printLayer(final Configuration configuration) {
    sortedModules(configuration.modules()).forEach(module -> {
      System.out.println(module.name());
      printModule(module);
      System.out.println("");
    });
  }

  private void printModule(final ResolvedModule module) {
    sortedModules(module.reads())
      .forEach(m -> System.out.println("|--" + m.name()));
  }

  private List<ResolvedModule> sortedModules(final Set<ResolvedModule> modules) {
    return modules
      .stream()
      .sorted(Comparator.comparing(ResolvedModule::name))
      .collect(Collectors.toList());
  }

  public static void main(final String[] args) {
    final Configuration configuration = Configuration.resolve(
      ModuleFinder.of(Paths.get(args[0])),
      List.of(Configuration.empty()),
      ModuleFinder.ofSystem(),
      List.of("io.vividcode.store.runtime")
    );
    new ConfigurationPrinter().printLayer(configuration);
  }
}
```

Listing 2-33 shows the output of Listing 2-32. It shows all modules of the sample application.

Listing 2-33. Output of the Readability Graph

```
io.vividcode.store.common
|--guava
|--jackson.annotations
|--jackson.core
|--jackson.databind
|--java.base
|--slf4j.api
|--slf4j.simple
```

```
io.vividcode.store.common.persistence
|--guava
|--io.vividcode.store.common
|--jackson.annotations
|--jackson.core
|--jackson.databind
|--java.base
|--slf4j.api
|--slf4j.simple

io.vividcode.store.filestore
|--guava
|--io.vividcode.store.common
|--io.vividcode.store.common.persistence
|--jackson.annotations
|--jackson.core
|--jackson.databind
|--java.base
|--slf4j.api
|--slf4j.simple

io.vividcode.store.product
|--io.vividcode.store.common
|--java.base

io.vividcode.store.product.persistence
|--io.vividcode.store.common
|--io.vividcode.store.common.persistence
|--io.vividcode.store.product
|--java.base

io.vividcode.store.runtime
|--guava
|--io.vividcode.store.common
|--io.vividcode.store.common.persistence
|--io.vividcode.store.filestore
|--io.vividcode.store.product
|--io.vividcode.store.product.persistence
|--jackson.annotations
|--jackson.core
|--jackson.databind
|--java.base
|--slf4j.api
|--slf4j.simple
```

The Module Layers

Module layers are used to organize modules. Module layers are represented as the class `java.lang.ModuleLayer`.

A layer is created from the readability graph in a `Configuration` object by mapping each resolved module to a class loader that is responsible for loading types defined in this module. The JVM has at least

one nonempty layer, the boot layer, which is created when the JVM starts. Most applications will only use this boot layer. Additional layers can be created to support advanced use cases. A layer can have multiple parent layers. Modules in a layer can also read modules in the layer's parent layers. When a layer is created, a `java.lang.Module` object is created for each `ResolvedModule` in the configuration.

The static method `boot()` of `ModuleLayer` returns the boot layer. There are three different ways to create a new layer from an existing `ModuleLayer`; see Table 2-10. The existing `ModuleLayer` will be the parent layer of the newly created layer.

Table 2-10. *Methods for Creating New `ModuleLayers`*

Method	Description
`defineModules(Configuration cf, Function<String, ClassLoader> clf)`	Creates a new `ModuleLayer` from the `Configuration`. For each module, the function returns the mapped `ClassLoader`.
`defineModulesWithOneLoader(Configuration cf, ClassLoader parentLoader)`	Creates a new `ModuleLayer` from the `Configuration`. All modules use the same `ClassLoader` with `parentLoader` as the parent class loader.
`defineModulesWithManyLoaders(Configuration cf, ClassLoader parentLoader)`	Creates a new `ModuleLayer` from the `Configuration`. Each module has its own `ClassLoader` with `parentLoader` as the parent class loader.

`ModuleLayer` also has the static methods `defineModules()`, `defineModulesWithOneLoader()`, and `defineModulesWithManyLoaders()` to create new layers. These methods require to provide the list of parent layers. Compared to those instance methods, these static methods return `ModuleLayer.Controller` instead of `ModuleLayer`. `ModuleLayer.Controller` has methods for modifying the modules in the layer; see Table 2-11.

Table 2-11. *Methods of `ModuleLayer.Controller`*

Method	Description
`addReads(Module source, Module target)`	Updates the module `source` to read the module `target`
`ModuleLayer.Controller addExports(Module source, String pn, Module target)`	Updates the module `source` to export the package `pn` to the module `target`
`ModuleLayer.Controller addOpens(Module source, String pn, Module target)`	Updates the module `source` to open the package `pn` to the module `target`
`ModuleLayer layer()`	Returns the `ModuleLayer` object

`ModuleLayer` also has other methods; see Table 2-12.

Table 2-12. *Methods of* `ModuleLayer`

Method	Description
`static ModuleLayer empty()`	Returns the empty layer.
`Configuration configuration()`	Returns the configuration for this layer.
`List<ModuleLayer> parents()`	Returns the list of this layer's parents.
`Set<Module> modules()`	Returns the set of modules in this layer.
`Optional<Module> findModule(String name)`	Returns the module with the given name. Parent layers are also searched recursively until the module is found.
`ClassLoader findLoader(String name)`	Returns the `ClassLoader` of the module with the given name. Parent layers are also searched in the same way as in `findModule()`.

The class `Module` represents a runtime module. It can be used to retrieve information about the module and modify it when required. Table 2-13 shows methods of `Module`.

Table 2-13. *Methods of* `Module`

Method	Description
`String getName()`	Returns the name of this module, `null` for an unnamed module
`Set<String> getPackages()`	Returns the set of package names in this module
`ModuleDescriptor getDescriptor()`	Returns the `ModuleDescriptor` object that describes this module, `null` for an unnamed module
`boolean isNamed()`	Checks if the module is named
`boolean isExported(String pn)`	Checks if the module exports a package `pn` unconditionally
`boolean isExported(String pn, Module other)`	Checks if the module exports a package `pn` to the given module `other`
`boolean isOpen(String pn)`	Checks if the module opens a package `pn` unconditionally
`boolean isOpen(String pn, Module other)`	Checks if the module opens a package `pn` to the given module `other`
`boolean canRead(Module other)`	Checks if the module reads the given module `other`
`boolean canUse(Class<?> service)`	Checks if the module uses the given service class
`Module addExports(String pn, Module other)`	Updates this module to export the given package `pn` to the given module `other`
`Module addOpens(String pn, Module other)`	Updates this module to open the given package `pn` to the given module `other`
`Module addUses(Class<?> service)`	Updates this module to add a service dependency on the given service type
`Module addReads(Module other)`	Updates this module to read the given module `other`
`ClassLoader getClassLoader()`	Gets the class loader of this module
`ModuleLayer getLayer()`	Returns the module layer that contains this module
`InputStream getResourceAsStream(String name)`	Returns an input stream to read a resource in the module

When using `getResourceAsStream()` to read the resources in a module, the resource may be encapsulated so that it cannot be located by code in other modules. Class files are not encapsulated. For other resources, the package name is derived from the resource name first. If the package name is in the module, that is, it is in the set of package names returned by `getPackages()`, then the caller code needs to be in a module that the package is open to.

Now I'll use an example to show you how to use `ModuleLayers` to have multiple versions of the same module coexist in the same application. In the module `demo`, I have a simple Java `Version` class that has the static method `getVersion()` to return the version string; see Listing 2-34. The version string is `1.0.0` for the module version `1.0.0` and is updated to `2.0.0` for the module version `2.0.0`.

Listing 2-34. `Version` Class in the Module `demo`

```
package io.vividcode.demo.version;

public class Version {
  public static String getVersion() {
    return "1.0.0";
  }
}
```

I have another `runtime` module that requires the module `demo`. Listing 2-35 shows the `Main` class that outputs the version string.

Listing 2-35. `Main` Class in the Module `runtime`

```
package io.vividcode.demo.runtime;

import io.vividcode.demo.version.Version;

public class Main {
  public static void main(final String[] args) {
    System.out.println(Version.getVersion());
  }
}
```

The module `runtime` and two versions of the module `demo` are packaged as JAR files. If you put these three JAR files in the same directory `dist` and run the module `runtime` using following command, you'll find out that the command cannot run.

```
$ java -p dist -m runtime
```

The command fails with following error. This is because two versions of the same module `demo` appear in the same directory of the module path.

```
Error occurred during initialization of boot layer
java.lang.module.FindException: Two versions of module demo found in dist (demo-2.0.0.jar
and demo-1.0.0.jar)
```

If you put `demo-1.0.0.jar` and `demo-2.0.0.jar` into separate directories and update the module path as shown here, the output is `1.0.0` because the `demo-1.0.0.jar` is found first.

```
$ java -p dist:dist/v1:dist/v2 -m runtime
```

If you change the order of `dist/v1` and `dist/v2` in the module path as shown here, the output is `2.0.0`, because `demo-2.0.0.jar` is found first.

```
$ java -p dist:dist/v2:dist/v1 -m runtime
```

If you want to have both versions running in the same JVM, you can use OSGi or create custom class loaders. In Java 9, you can use `ModuleLayers`. Listing 2-35 shows the updated version of the `Main` class using `ModuleLayers`. In the method `createLayer()`, `path` is the path of the directory that contains the artifact of module `demo`. In this code, I create a `Configuration` that uses the `ModuleFinder` to only search in this path. The parent `Configuration` comes from the boot layer. I then use `defineModulesWithOneLoader()` to create the module layer from the `Configuration` and use the system class loader as the parent class loader for the only `ClassLoader` object. The created `ModuleLayer` uses the boot layer as the parent.

Once I have created the module layer, the method `showVersion()` uses the method `findLoader()` to find the `ClassLoader` for the module and the use reflections API to load the class and invoke the method `getVersion()`; see Listing 2-36. If you run the class `Main`, you can see that both version strings are displayed.

Listing 2-36. Using `ModuleLayer` to Load Modules

```
import java.lang.module.Configuration;
import java.lang.module.ModuleFinder;
import java.lang.reflect.Method;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Set;

public class Main {
  private static final String MODULE_NAME = "demo";
  private static final String CLASS_NAME = "io.vividcode.demo.version.Version";
  private static final String METHOD_NAME = "getVersion";

  public static void main(final String[] args) {
    final Main main = new Main();
    main.load(Paths.get("dist", "v1"));
    main.load(Paths.get("dist", "v2"));
  }

  public void load(final Path path) {
    showVersion(createLayer(path));
  }

  private void showVersion(final ModuleLayer moduleLayer) {
    try {
      final Class<?> clazz = moduleLayer.findLoader(MODULE_NAME)
          .loadClass(CLASS_NAME);
      final Method method = clazz.getDeclaredMethod(METHOD_NAME);
      System.out.println(method.invoke(null));
    } catch (final Exception e) {
      e.printStackTrace();
    }
  }
```

```
  private ModuleLayer createLayer(final Path path) {
    final ModuleLayer parent = ModuleLayer.boot();
    final Configuration configuration = parent.configuration().resolve(
        ModuleFinder.of(path),
        ModuleFinder.of(),
        Set.of(MODULE_NAME)
    );
    return parent.defineModulesWithOneLoader(configuration,
        ClassLoader.getSystemClassLoader());
  }
}
```

Class Loaders

OSGi uses a complicated class loader hierarchy to allow different versions of the same bundle to coexist at runtime. JPMS uses a simple class loading strategy. A module has its own class loader that is responsible for loading all types in this module. A class loader can load types from one module or from many modules. All of the types in one module should be loaded by the same class loader.

Before Java 9, Java runtime has three built-in class loaders:

- *Bootstrap class loader*: JVM built-in class loader, typically represented as `null`. It has no parent.
- *Extension class loader*: Loads classes from the extension directory. It's a result of the extension mechanism introduced in JDK 1.2. Its parent is the bootstrap class loader.
- *System or application class loader*: Loads classes from the application's class path. Its parent is the extension class loader.

The code in Listing 2-37 outputs the class loader hierarchy.

Listing 2-37. Outputting the Class Loader Hierarchy

```
ClassLoader classLoader = ClassLoaderMain.class.getClassLoader();
while (classLoader != null) {
  System.out.println(classLoader);
  classLoader = classLoader.getParent();
}
```

When you're running the code in Listing 2-37 on Java 8, the following output is shown. `AppClassLoader` and `ExtClassLoader` are the classes of the system class loader and the extension class loader, respectively. The bootstrap class loader is represented as `null`, so it's not shown.

```
sun.misc.Launcher$AppClassLoader@18b4aac2
sun.misc.Launcher$ExtClassLoader@60e53b93
```

In Java 9, the extension mechanism has been superseded by the upgrade module path. The extension class loader is still kept for backward compatibility, but it is renamed to platform class loader. It can be retrieved using the method `getPlatformClassLoader()` of `ClassLoader`. The following are the built-in class loaders in Java 9:

- *Bootstrap class loader*: Defines core Java SE and JDK modules
- *Platform class loader*: Defines selected Java SE and JDK modules
- *System or application class loader*: Defines classes on the class path and modules in the module path

In Java 9, the platform class loader and system class loader are no longer instances of `URLClassLoader`. This affects a popular trick (https://stackoverflow.com/a/7884406) to dynamically add new entries to the search path of the system class loader. As shown in Listing 2-38, this hack converts the system class loader to `URLClassLoader` and invokes its method `addURL()`. This no longer works in Java 9 as the cast to `URLClassLoader` fails.

Listing 2-38. Adding a Path to `URLClassLoader`

```
public static void addPath(String s) throws Exception {
  File f = new File(s);
  URL u = f.toURL();
  URLClassLoader urlClassLoader = (URLClassLoader)
      ClassLoader.getSystemClassLoader();
  Class urlClass = URLClassLoader.class;
  Method method = urlClass.getDeclaredMethod("addURL", new Class[]{URL.class});
  method.setAccessible(true);
  method.invoke(urlClassLoader, new Object[]{u});
}
```

Every class loader has its own unnamed module that can be retrieved using the method `getUnnamedModule()` of `ClassLoader`. If a class loader loads a type that is not defined in a named module, this type is considered to be in the class loader's unnamed module. The unnamed module we discussed earlier is actually the unnamed module of the application class loader.

In Java 9, class loaders have names. The name is specified in the constructor and can be retrieved using the method `getName()`. The name of the platform class loader is `platform`, whereas the name of application class loader is `app`. The new class loader name can be useful when you're debugging class loader–related issues.

Listing 2-39. Testing the Class Loader Names

```
@Test
public void testClassLoaderName() {
  ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
  final List<String> names = Lists.newArrayList();
  while (classLoader != null) {
    names.add(classLoader.getName());
    classLoader = classLoader.getParent();
  }
  assertEquals(2, names.size());
  assertEquals("app", names.get(0));
  assertEquals("platform", names.get(1));
}
```

The new method `Class<?> findClass(String moduleName, String name)` finds the class with a given binary name in a module defined to this class loader. If the module name is `null`, it finds the class in the unnamed module of this class loader. If the class cannot be found, this method returns `null` instead of throwing a `ClassNotFoundException`. The new method `URL findResource(String moduleName, String name)` returns a URL to the resource with a given name in a module that is defined for this class loader. The method `Module.getResourceAsStream()` actually uses this method of the module's class loader to get the `URL` first and then open the input stream. The methods `findClass()` and `findResource()` are both protected and should be overridden by class loader implementations.

`ClassLoader` already has the method `Enumeration<URL> getResources(String name)` to find all resources with the given name. The new method `Stream<URL> resources(String name)` has the same functionality, but it returns a `Stream<URL>`.

Java 9 adds a new restriction when accessing the resources that matches the access control rules enforced by module declaration. Resources in the named modules are subject to the encapsulation rules specified in the method `Module.getResourceAsStream()`. For nonclass resources, the related methods in `ClassLoader` can only find resources in packages when the package is open unconditionally.

For example, the module `io.vividcode.store.common` has a properties file `application.properties` in the directory `config`. The package name of this resource is `config`. If the package `config` is not declared as `open` in the module declaration, it cannot be located using those resource-related methods.

In Listing 2-40, I get the `Module` object of the module `io.vividcode.store.common` first, then I get its class loader and use the method `resources()` to list all resources with name `config/application.properties`. Here the package `config` must be open, otherwise the method `resources()` returns an empty stream.

Listing 2-40. Testing `ClassLoader.resources()`

```
public class ResourceTest {

  @Test
  public void testResources() throws URISyntaxException {
    final Optional<Module> moduleOpt = ModuleTestSupport.getModule();
    assertTrue(moduleOpt.isPresent());
    final Module module = moduleOpt.get();
    assertTrue(module.isOpen("config"));
    assertTrue(module
        .getClassLoader()
        .resources("config/application.properties")
        .count() > 0
    );
  }
}
```

The new method `Package[] getDefinedPackages()` returns all of the `Packages` defined by this class loader, while `Package getDefinedPackage(String name)` returns the `Package` of the given name defined by this class loader. In Listing 2-41, the class loader to test is actually the system class loader, so it defines application packages, but not packages like `java.lang`.

Listing 2-41. Testing `ClassLoader.getDefinedPackages()`

```
@Test
public void testGetDefinedPackages() {
  final ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
  final Package[] packages = classLoader.getDefinedPackages();
```

```
  assertTrue(Stream.of(packages)
      .map(Package::getName)
      .noneMatch(Predicate.isEqual("java.lang")));
  assertTrue(Stream.of(packages)
      .map(Package::getName)
      .anyMatch(Predicate.isEqual("io.vividcode.feature9.module")));
  assertNull(classLoader.getDefinedPackage("java.lang"));
  assertNotNull(
    classLoader.getDefinedPackage("io.vividcode.feature9.module"));
}
```

Class

With the introduction of modules, `Class` has new methods related to modules. `Class<?> forName(Module module, String name)` returns the `Class` with the given binary name in the given module. This method delegates to the module's class loader for class loading. Similar to the method `findClass()` of `ClassLoader`, `forName()` returns `null` on failure rather than throwing a `ClassNotFoundException`. Since a class loader can define classes for multiple modules, it's possible that the defined class actually comes from a different module. In this case, the method also returns `null` after the class is loaded.

Since every type is now in a module, `Class` has a new method, `getModule()`, to return the `Module` object representing the module it belongs to. If this class represents an array type, then this method returns the `Module` object for the element type. If this class represents a primitive type or `void`, then the `Module` object for the module `java.base` is returned. If this class is in an unnamed module, then the result of `getUnnamedModule()` of the class loader for this class is returned. Listing 2-42 shows the test related to `Class.getModule()`.

Listing 2-42. Testing `Class.getModule()`

```
@Test
public void testGetModule() {
  assertEquals("java.sql", Driver.class.getModule().getName());
  assertEquals("java.base", String[].class.getModule().getName());
  assertEquals("java.base", int.class.getModule().getName());
  assertEquals("java.base", void.class.getModule().getName());
}
```

The new method `String getPackageName()` returns the fully qualified package name of a class. If this class represents an array type, then this method returns the package name of the element type. If this class represents a primitive type or `void`, then the package name `java.lang` is returned. Otherwise, the package name is derived from the class name returned by `Class.getName()` by keeping the characters before the last dot (`.`). Listing 2-43 shows the test related to `Class.getPackageName()`.

Listing 2-43. Testing `Class.getPackageName()`

```
@Test
public void testGetPackageName() {
  assertEquals("java.sql", Driver.class.getPackageName());
  assertEquals("java.lang", String[].class.getPackageName());
  assertEquals("java.lang", int.class.getPackageName());
  assertEquals("java.lang", void.class.getPackageName());
}
```

Reflection

In the previous section, I showed you how to use `ServiceLoader` to load provider classes of service interfaces. Many frameworks use the similar pattern and reflection API to dynamically load classes by name and instantiate them. A client can provide the class name to the framework through configuration. The framework uses `Class.forName()` to load the class and instantiate it using the method `newInstance()` of the `Class` object.

This kind of pattern may have issues in the module system. The actual class may be loaded from the unnamed module or another named module. If the actual class is loaded from the unnamed module, then the instantiation fails because the framework's module cannot read the unnamed module. If the actual class is loaded from another named module, it's not possible for the framework's module to know of the existence of client modules and declare dependencies upon them, which actually reverses the dependencies. To make this work, the reflection API has an implicit convention in which it assumes any code that reflects a type is in a module that reads the module that defines this type. The reflection API has the access to the type.

In Listing 2-44, I use the reflection API in a named module to load the Guava class `com.google.common.eventbus.EventBus` and instantiate a new instance. The Guava library is put into the class path, so it's in the unnamed module. The output of `clazz.getModule()` is something similar to `unnamed module @1623b78d`, which confirms that the class is in the unnamed module. The instantiation is successful and you can see output like `EventBus{default}`.

Listing 2-44. Using the Reflection API to Instantiate an Instance

```
try {
  final Class<?> clazz = Class.forName("com.google.common.eventbus.EventBus");
  System.out.println(clazz.getModule());
  final Object instance = clazz.getDeclaredConstructor().newInstance();
  System.out.println(instance);
} catch (final Exception e) {
  e.printStackTrace();
}
```

■ **Note** The method `Class.newInstance()` is deprecated in Java 9. In Listing 2-44, I use `clazz.getDeclaredConstructor().newInstance()` to get the constructor and then use it to instantiate new instances.

Automatic Module Names

I mentioned names of automatic modules earlier. If no attribute `Automatic-Module-Name` is found in the manifest of a JAR file, the module name is derived from the file name. The module name is determined using following steps.

1. Remove the suffix `.jar`.
2. Attempt to match the file name to the regular expression pattern `-(\\d+(\\.|$))`. If the pattern matches, then the substring before the matching position is treated as the candidate of module name, while the substring after the dash (`-`) is parsed as the version. If the version cannot be parsed, then it's ignored. If the pattern doesn't match, the whole file name is treated as the candidate of module name.

3. Clean up the candidate of module name to create a valid module name. All nonalphanumeric characters (`[^A-Za-z0-9]`) in the module name are replaced with a dot (`.`), all repeating dots are replaced with one dot, and all leading and trailing dots are removed.

If you are interested in knowing the actual implementation of these steps, check the method `ModuleDescriptor deriveModuleDescriptor(JarFile jf)` of the class `jdk.internal.module.ModulePath` in the module `java.base`. The parse of the version string is done using the static method `Version parse(String v)` of `ModuleDescriptor.Version`. The code in Listing 2-45 also implements the same algorithm.

Table 2-14 shows some examples of deriving automatic module names from the JAR file names.

Table 2-14. *Example of Deriving Automatic Module Names for JAR File Names*

File Name	Module Name	Version
`mylib.jar`	`mylib`	`null`
`slf4j-api-1.7.25.jar`	`slf4j.api`	`1.7.25`
`hibernate-jpa-2.1-api-1.0.0.Final.jar`	`hibernate.jpa`	`2.1-api-1.0.0.Final`
`spring-context-support-4.3.6.RELEASE.jar`	`spring.context.support`	`4.3.6.RELEASE`

Listing 2-45 shows the code to derive the automatic module names shown in Table 2-14.

Listing 2-45. Deriving Automatic Module Names

```
public class DeriveAutomaticModuleName {

  static final Pattern DASH_VERSION = Pattern.compile("-(\\d+(\\.|$))");
  static final Pattern NON_ALPHANUM = Pattern.compile("[^A-Za-z0-9]");
  static final Pattern REPEATING_DOTS = Pattern.compile("(\\.)(\\1)+");
  static final Pattern LEADING_DOTS = Pattern.compile("^\\.");
  static final Pattern TRAILING_DOTS = Pattern.compile("\\.$");

  public Tuple2<String, String> deriveModuleName(final String fileName) {
    Objects.requireNonNull(fileName);
    String name = fileName;
    String version = null;
    if (fileName.endsWith(".jar")) {
      name = fileName.substring(0, fileName.length() - 4);
    }

    final Matcher matcher = DASH_VERSION.matcher(name);
    if (matcher.find()) {
      final int start = matcher.start();

      try {
        final String tail = name.substring(start + 1);
        ModuleDescriptor.Version.parse(tail);
```

```
        version = tail;
      } catch (final IllegalArgumentException ignore) {
      }

      name = name.substring(0, start);
    }
    return Tuple.of(cleanModuleName(name), version);
  }

  public void displayModuleName(final String fileName) {
    final Tuple2<String, String> result = deriveModuleName(fileName);
    System.out.printf("%s => %s [%s]%n", fileName, result._1, result._2);
  }

  private String cleanModuleName(String mn) {
    // replace non-alphanumeric
    mn = NON_ALPHANUM.matcher(mn).replaceAll(".");

    // collapse repeating dots
    mn = REPEATING_DOTS.matcher(mn).replaceAll(".");

    // drop leading dots
    if (mn.length() > 0 && mn.charAt(0) == '.') {
      mn = LEADING_DOTS.matcher(mn).replaceAll("");
    }

    // drop trailing dots
    final int len = mn.length();
    if (len > 0 && mn.charAt(len - 1) == '.') {
      mn = TRAILING_DOTS.matcher(mn).replaceAll("");
    }

    return mn;
  }

  public static void main(final String[] args) {
    final DeriveAutomaticModuleName moduleName = new DeriveAutomaticModuleName();
    moduleName.displayModuleName("mylib.jar");
    moduleName.displayModuleName("slf4j-api-1.7.25.jar");
    moduleName.displayModuleName("hibernate-jpa-2.1-api-1.0.0.Final.jar");
    moduleName.displayModuleName("spring-context-support-4.3.6.RELEASE.jar");
  }
}
```

If the module name is specified using the manifest attribute `Automatic-Module-Name`, then the name is used as is. If the specified module name is invalid, then a `FindException` is thrown when the module is loaded.

Module Artifacts

Modules are packaged as module artifacts. There are two types of module artifacts, JAR files and JMOD files.

JAR Files

A modular JAR file is just a JAR file with the file `module-info.class` in its root directory, so you can use the existing `jar` tool to create modular JAR files. The `jar` tool also adds new options to insert additional information into module descriptors.

- `-e`, `--main-class=CLASSNAME` records the entry point class in the file `module-info.class`. This is actually an old option that records the main class in the manifest file.
- `--module-version=VERSION` records the `VERSION` in the `module-info.class` file as the module's version.
- `--hash-modules=PATTERN` records hashes of modules that depend upon this module in the `module-info.class` file. Hashes are only recorded for modules whose names match the regular expression specified with `PATTERN`.
- `-d`, `--describe-module` prints the module descriptor or the name of an automatic module.

You can use the command in Listing 2-46 to create a modular JAR file for the module `io.vividcode.store.runtime`. Here I specified the module version and main class.

Listing 2-46. Using `jar` to Create Module Artifacts

```
$ jar --create --file target/runtime-1.0.0.jar \
  --main-class io.vividcode.store.runtime.Main \
  --module-version 1.0.0 \
  -C target/classes .
```

Then you can print out the details of the created JAR file.

```
$ jar -d -f target/runtime.jar
```

Listing 2-47 shows the output of this command.

Listing 2-47. Output of `jar -d`

```
module io.vividcode.store.runtime@1.0.0 (module-info.class)
  requires io.vividcode.store.filestore
  requires io.vividcode.store.product.persistence
  requires mandated java.base
  requires slf4j.simple
  contains io.vividcode.store.runtime
  main-class io.vividcode.store.runtime.Main
```

To record module hashes using `--hash-modules`, you need to also provide the module path using `-p` or `--module-path` for the `jar` tool to locate modules.

JMOD Files

JMOD files are introduced in Java 9 to package JDK modules. Compared to JAR files, JMOD files can contain native code, configuration files, and other kinds of data. Figure 2-3 shows the JMOD files in the `jmods` directory of the JDK.

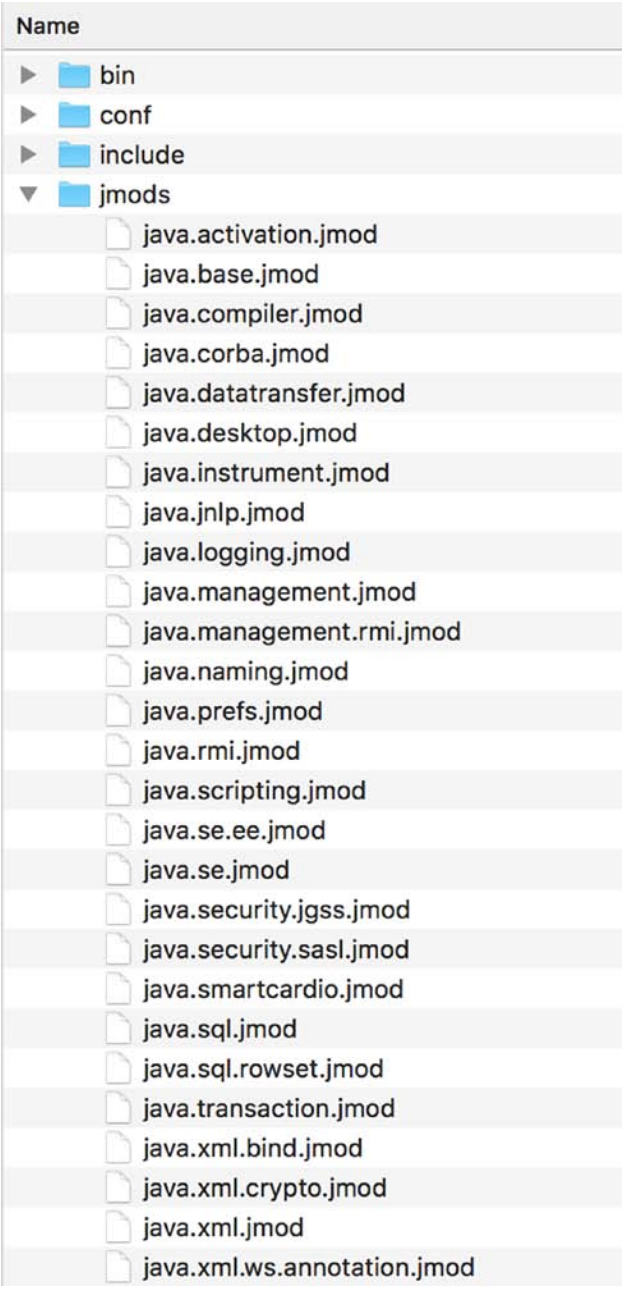

Figure 2-3. *JDK JMOD files*

Developers can also use JMOD files to package module files using the new `jmod` tool.

You can use the following `jmod list` command to list the names of all entries in the JMOD file of the module `java.base`.

```
$ jmod list <JDK_PATH>/jmods/java.base.jmod
```

This JMOD file contains directories listed in Table 2-15.

Table 2-15. *Directories in JMOD Files*

Directory Name	Files
`classes/`	Java class files
`conf/`	Configuration files
`include/`	Header files
`legal/`	Legal files
`bin/`	Executable binaries
`lib/`	Native libraries

Other modules, for example, `java.sql` and `java.xml`, may only contain `classes` and `legal` directories.

The command `jmod describe` prints out the module details, which are similar to `jar -d`.

```
$ jmod describe <JDK_PATH>/jmods/java.base.jmod
```

The command `jmod extract` extracts all the files to the target directory.

```
$ jmod extract <JDK_PATH>/jmods/java.sql.jmod --dir <output_dir>
```

The command `jmod create` creates JMOD files. You can use different options to provide the path for various kinds of files; see Table 2-16.

Table 2-16. *Options of jmod*

Option	Description
`--class-path`	JAR files or directories containing Java class files
`--cmds`	Path of native commands
`--config`	Path of configuration files
`--header-files`	Path of header files
`--legal-notices`	Path of legal files
`--libs`	Path of native libraries
`--man-pages`	Path of man pages

`jmod create` also supports the options `--main-class` and `--module-version`. Options `--os-arch` and `--os-name` can be used to specify the operating system architecture and name. Listing 2-48 shows how to use `jmod` to create a JMOD file.

Listing 2-48. Using `jmod` to Create Module Artifacts

```
$ jmod create --class-path target/classes \
  --main-class io.vividcode.store.runtime.Main \
  --module-version 1.0.0 \
  --os-arch x86_64 \
  --os-name "Mac OS X" \
  target/runtime-1.0.0.jmod
```

JDK Modules

In Java 9, JDK itself is organized as multiple modules. There are currently 94 modules in JDK. These modules have four different prefixes to indicate their groups; see Table 2-17.

Table 2-17. *Groups of JDK Modules*

Group	Description
`java.`	Standard Java modules, for example, `java.base`, `java.logging`, `java.sql` and `java.xml`
`javafx.`	JavaFX modules, for example, `javafx.base`, `javafx.deploy` and `javafx.media`
`jdk.`	Part of JDK implementation, for example, `jdk.compiler`, `jdk.charsets`, `jdk.accessibility` and `jdk.management`
`oracle.`	Oracle modules, for example, `oracle.desktop` and `oracle.net`

Common Issues

With the introduction of the module system, some common issues may occur. You've already seen a potential issue—that the platform and application class loaders are no longer instances of `URLClassLoader`. There are other potential issues you may need to be aware of.

The modifier `public` no longer guarantees that a program element is accessible everywhere. Accessibility now depends on various conditions related to the module system, including whether the package is exported or open and whether its module is readable by the module containing the code that is attempting to access it.

If a package is defined in both a named module and on the class path, then the package on the class path is ignored. For example, the package `javax.transaction` is defined in the module `java.transaction`. If the module `java.transaction` is in the module path, and the JAR file `javax.transaction-api-1.2.jar` that contains the same package `javax.transaction` is also on the class path, the classes in the JAR file are ignored. That's why you may get the error `java.lang.ClassNotFoundException: javax.transaction.SystemException` when you're running Spring JPA applications. The solution is to either remove the module `java.transaction` from the module path or put the JAR file in the upgrade module path to override built-in modules.

Modules that define Java EE APIs, or APIs primarily of interest to Java EE applications, have been deprecated and will be removed in a future release. These modules are not resolved by default for code on the class path. The default set of root modules for the unnamed module is based upon the `java.se` module. The `java.se` module only requires modules shown in Listing 2-49.

Listing 2-49. Modules Required by the `java.se` Module

```
java.compiler
java.datatransfer
java.desktop
java.instrument
java.logging
java.management
java.management.rmi
java.naming
java.prefs
java.rmi
java.scripting
java.security.jgss
java.security.sasl
java.sql
java.sql.rowset
java.xml
java.xml.crypto
```

The modules shown in Listing 2-50 are required in the `java.se.ee` module, so by default, code in the unnamed module doesn't have access to APIs defined in these modules.

Listing 2-50. Modules Required by the `java.se.ee` Module

```
java.activation
java.corba
java.transaction
java.xml.bind
java.xml.ws
java.xml.ws.annotation
```

If the project uses classes from the modules in Listing 2-50, the code cannot compile. You need to update the Java compiler option to add `--add-modules java.se.ee` to include the `java.se.ee` module.

Most of JDK's internal APIs are inaccessible by default at compile time. Some of these internal APIs, especially the `sun.misc` package, have been used by some popular libraries. In JDK 9, noncritical internal APIs, for example `sun.misc.BASE64Decoder`, are encapsulated and inaccessible by default. Critical internal APIs, for example, `sun.misc.Unsafe` and `sun.reflect.Reflection`, are not encapsulated. These APIs are defined in the JDK specific module `jdk.unsupported`. If your project requires you to use these packages, you can make your module require `jdk.unsupported`. For those encapsulated internal APIs, they can be made accessible using `--add-exports` and `--add-opens` when necessary.

The options `-Xbootclasspath` and `-Xbootclasspath/p` have been removed. The JDK-specific system property `sun.boot.class.path` has also been removed. This is because the bootstrap class path is empty by default, since bootstrap classes are loaded from modules. If the code depends on the bootstrap class path, it may not run using JDK 9. For example, the class `org.elasticsearch.monitor.jvm.JvmInfo` in Elasticsearch 1.7.1 tries to get the bootstrap class path from the JMX bean `java.lang.management.RuntimeMXBean` using the method `getBootClassPath()`. This method in Java 9 simply throws `UnsupportedOperationException`. This causes Elasticsearch 1.7.1 to fail to start when running on JDK 9. The correct implementation should check the method `boolean isBootClassPathSupported()` first before calling `getBootClassPath()`.

Migration in Action

Now that you understand all the important concepts in the Java 9 module system, you have a clear path to migrate existing applications to use modules.

Building the Project Using Java 9

Before migrating the code to Java 9, you should build and run the project on Java 9 first. This requires configuration changes to your existing build tools. For Maven, you need to upgrade the Maven compiler plugin to the latest version and set the source and target configuration to `9`. For Gradle, you need to update the properties `sourceCompatibility` and `targetCompatibility` of the Java plugin to `9`. You should also upgrade Maven or Gradle plugins to the latest version for better support of Java 9.

After updating the configuration, try to run the build. It's possible you'll encounter issues with build tools, especially since Java 9 has just been released and it takes time for the build tools to provide the support for it. At the time of writing, Gradle 4.2 doesn't have first-class support for Java 9 modules yet. You may need to find workarounds for various issues. Or you can wait until the support of build tools is mature before you start the migration. As I mentioned before, Java 9 has no issues running applications written before it.

When building the project using Java 9, you may encounter some errors. Most of these errors are apt to be related to class access and reflections. This is because modules have stricter access control. The default value of the option `--illegal-access` is `permit`, so Java 9 already allows code on the class path to perform illegal access. With this default mode, most of reflection-related issues can be resolved. JVM outputs warning messages about illegal access, but the application can be started.

The Migration Path

After you can successfully build the project using JDK 9, you can start the migration process. The following are the basic steps for migrating an existing application to modules.

1. Add existing third-party libraries into the module path. If you are using build tools like Maven or Gradle, you may not need to do anything. These tools should already handle this for you. If you're using JDK tools, you can specify the module path using `--module-path` or `-p`, and then point to the directory of libraries. Putting libraries into the module path is the first step to moving from class path to module paths. These libraries become automatic modules and you don't need to migrate them.

2. List the dependency tree. You can do this using the task `dependency:tree` of the Maven dependency plugin or the Gradle command `gradle dependencies`. The dependency tree is used to understand dependencies between a project's modules or subprojects. This step is not necessary if you already know about the dependencies. As I mentioned before, you should do a bottom-up migration, so you need the project's module dependencies tree to figure out the migration order.

3. Start from the subprojects in the bottom of the dependency tree and add module declaration files for each subproject. Third-party libraries are also declared in the `module-info.java` files as automatic modules. As a start, simply export all packages for each module. If your code already uses conventions to distinguish between public and internal packages, you should leverage it in the module declaration. For example, you can export packages with names that end with `api` but not packages that end with `impl` or `internal`. If service providers are used in the application, add them in the module declaration.

4. Try to compile the code and fix compilation errors. The IDE can help you to auto-fix most of the issues.
5. Run all unit tests and integration tests to make sure all are passed.
6. Improve the module declarations to remove unnecessary exports and fix issues found during tests.

These steps can be simplified with the help of the `jdeps` tool. The `jdeps` tool can generate module declaration files.

BioJava

To demonstrate the migration of Maven projects, I'll use a real-world example. Let's take a look at the BioJava (`http://biojava.org/`) project. BioJava is an open-source project (`https://github.com/biojava/biojava`) dedicated to providing a Java framework for processing biological data. It's a Maven project with 13 modules, so it's a good candidate for demonstrating the migration process.

The first step is to build the project using Java 9. Before building the project, you need to make sure Maven is using Java 9. You can use `mvn -v` to verify the JVM that Maven is using. If Java 9 is the default JVM on your local machine, then you are good to go. If you have multiple JDKs installed, you need to make sure the system property `JAVA_HOME` points to JDK 9. Now you can use `mvn` to build the project. For Intellij IDEA users, you can configure the JRE to run Maven in **Preferences ➤ Build, Execution, Deployment ➤ Build Tools ➤ Maven ➤ Runner**.

Now you should configure the source and target level of the Maven compiler plugin. This can be done by updating the property `jdk.version` to `9`. You also need to update the plugins to the latest version. This is necessary to avoid issues when using these plugins in Java 9. Table 2-18 shows the latest versions of these plugins.

Table 2-18. *Versions of Maven Plugins*

Plugin	Version
maven-compiler-plugin	`3.7.0`
maven-jar-plugin	`3.0.2`
maven-assembly-plugin	`3.1.0`
maven-surefire-plugin	`2.20.1`
maven-enforcer-plugin	`3.0.0-M1`

When you first start the Maven build, you encounter the first issue with Maven enforcer plugin; see Listing 2-51. This is because the Maven enforcer plugin version 1.2 uses an older version (2.3) of Apache Commons Lang, which doesn't support the new version scheme of Java 9. Upgrading the enforcer plugin to version `3.0.0-M1` solves this issue.

Listing 2-51. Issue with Maven Enforcer Plugin

```
Caused by: java.lang.ExceptionInInitializerError
  at org.apache.maven.plugins.enforcer.RequireJavaVersion.execute(RequireJavaVersion.java:52)
  at org.apache.maven.plugins.enforcer.EnforceMojo.execute(EnforceMojo.java:178)
  at org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(DefaultBuildPlugin
  Manager.java:134)
  ... 22 more
```

```
Caused by: java.lang.StringIndexOutOfBoundsException: begin 0, end 3, length 1
  at java.base/java.lang.String.checkBoundsBeginEnd(String.java:3116)
  at java.base/java.lang.String.substring(String.java:1885)
  at org.apache.commons.lang.SystemUtils.getJavaVersionAsFloat(SystemUtils.java:1122)
  at org.apache.commons.lang.SystemUtils.<clinit>(SystemUtils.java:818)
  ... 25 more
```

When compiling the source code, you'll encounter the second issue `java.lang.ClassNotFoundException: javax.xml.bind.JAXBException`. This is because BioJava uses JAXB. The module `java.xml.bind` is not in the default set of root modules for the unnamed module, so its classes cannot be found. To fix this issue, you need to update the Java compiler option to add `--add-modules java.se.ee`. You can do this by configuring the Maven compiler plugin; see Listing 2-52.

Listing 2-52. Configuring the Maven Compiler Plugin to Add Extra Modules

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.7.0</version>
  <configuration>
    <source>${jdk.version}</source>
    <target>${jdk.version}</target>
    <compilerArgs>
      <arg>--add-modules</arg>
      <arg>java.se.ee</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

The same option should also be added to the Maven surefire plugin using the configuration `argLine`. After you add this option, you can compile the project successfully.

Now you should run the unit tests. During the unit test, you'll encounter the third issue that causes some tests to fail; see Listing 2-53.

Listing 2-53. Unit Test Error

```
org.biojava.nbio.structure.align.ce.CeCPMainTest
testCECP1(org.biojava.nbio.structure.align.ce.CeCPMainTest)  Time elapsed: 1.394 sec  <<< ERROR!
java.lang.ExceptionInInitializerError
  at org.biojava.nbio.structure.align.ce.CeCPMainTest.testCECP1(CeCPMainTest.java:415)
Caused by: java.lang.RuntimeException: Could not initialize JAXB context
  at org.biojava.nbio.structure.align.ce.CeCPMainTest.testCECP1(CeCPMainTest.java:415)
Caused by: javax.xml.bind.JAXBException: Package java.util with JAXB class java.util.TreeSet
defined in a module java.base must be open to at least java.xml.bind module.
  at org.biojava.nbio.structure.align.ce.CeCPMainTest.testCECP1(CeCPMainTest.java:415)
```

After enabling error stacktrace, it turns out that it's caused by the code in Listing 2-54 of class `org.biojava.nbio.structure.scop.server.XMLUtil`. The error message is very clear—the package `java.util` in the module `java.base` needs to be open to the module `java.xml.bind`.

Listing 2-54. Code with Issues in `XMLUtil`

```
static JAXBContext jaxbContextDomains;
static {
  try {
    jaxbContextDomains= JAXBContext.newInstance(TreeSet.class);
  } catch (JAXBException e){
    throw new RuntimeException("Could not initialize JAXB context", e);
  }
}
```

You can use the option `--add-opens java.base/java.util=java.xml.bind` to fix the issue. This option needs to be added to configuration `argLine` of the Maven surefire plugin.

Now all the unit tests can pass successfully and you can move on to the migration. Let's use `jdeps` to generate module declaration files for all modules. To do this, you need to copy all the project's artifacts and third-party libraries to a directory. You can use the task `dependency:copy-dependencies` to copy dependencies for all modules and the task `dependency:copy` to copy artifacts of modules. Listing 2-55 shows the Maven configuration to copy the artifacts and dependencies during the `install` phase.

Listing 2-55. Copying the Artifacts and Dependencies

```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <phase>install</phase>
      <id>copy-jar</id>
      <goals>
        <goal>copy</goal>
      </goals>
      <configuration>
        <artifactItems>
          <artifactItem>
            <groupId>${project.groupId}</groupId>
            <artifactId>${project.artifactId}</artifactId>
            <version>${project.version}</version>
            <type>${project.packaging}</type>
          </artifactItem>
        </artifactItems>
        <outputDirectory>${targetDirectory}</outputDirectory>
      </configuration>
    </execution>
    <execution>
      <phase>install</phase>
      <id>copy-dependencies</id>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${targetDirectory}</outputDirectory>
      </configuration>
```

```
    </execution>
  </executions>
</plugin>
```

After running these two tasks, the target directory contains all the JAR files you need to work with. There are two versions of `commons-lang` in the directory; you need to remove the `commons-lang-2.4.jar`. Now let's generate module declaration files. In the following command, the `<output_dir>` is the output directory for the generated module declaration files; `<biojava_assembly_dir>` is the directory contains all the artifacts.

```
$ jdeps --generate-module-info <output_dir> <biojava_assembly_dir>
```

Errors occur when generating module declaration files, because `jdeps` also tries to generate module declaration files for those third-party libraries. Since you only care about project's modules, you can simply ignore these errors. In the output directory, you can find the `module-info.java` files for each module. Listing 2-56 shows the `module-info.java` for the module `biojava.core`.

Listing 2-56. Generated `module-info.java` for `biojava.core`

```
module biojava.core {
  requires java.logging;
  requires java.rmi;
  requires slf4j.api;

  requires transitive java.xml;

  exports org.biojava.nbio.core.alignment;
  exports org.biojava.nbio.core.alignment.matrices;
  exports org.biojava.nbio.core.alignment.template;
  exports org.biojava.nbio.core.exceptions;
  exports org.biojava.nbio.core.search.io;
  exports org.biojava.nbio.core.search.io.blast;
  exports org.biojava.nbio.core.sequence;
  exports org.biojava.nbio.core.sequence.compound;
  exports org.biojava.nbio.core.sequence.edits;
  exports org.biojava.nbio.core.sequence.features;
  exports org.biojava.nbio.core.sequence.io;
  exports org.biojava.nbio.core.sequence.io.template;
  exports org.biojava.nbio.core.sequence.io.util;
  exports org.biojava.nbio.core.sequence.loader;
  exports org.biojava.nbio.core.sequence.location;
  exports org.biojava.nbio.core.sequence.location.template;
  exports org.biojava.nbio.core.sequence.reference;
  exports org.biojava.nbio.core.sequence.storage;
  exports org.biojava.nbio.core.sequence.template;
  exports org.biojava.nbio.core.sequence.transcription;
  exports org.biojava.nbio.core.sequence.views;
  exports org.biojava.nbio.core.util;

  provides org.biojava.nbio.core.search.io.ResultFactory with
      org.biojava.nbio.core.search.io.blast.BlastXMLParser,
      org.biojava.nbio.core.search.io.blast.BlastTabularParser;
}
```

You copy these `module-info.java` files into the `src/main/java` directory of the corresponding module. Now you can successfully migrate them to JPMS modules. You can try to build the project and when you do, you'll encounter the errors in Listing 2-57.

Listing 2-57. Compilation Errors After Adding `module-info.java` Files

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.7.0:compile
(default-compile) on project biojava-structure-gui: Compilation failure: Compilation
failure:
[ERROR] /Users/fucheng/git/biojava/biojava-structure-gui/src/main/java/demo/
ShowStructureInJmol.java:[21,1] package exists in another module: jcolorbrewer
[ERROR] /Users/fucheng/git/biojava/biojava-structure-gui/src/main/java/demo/DemoMultipleMC.
java:[21,1] package exists in another module: jcolorbrewer
[ERROR] /Users/fucheng/git/biojava/biojava-structure-gui/src/main/java/demo/DemoCeSymm.
java:[21,1] package exists in another module: jcolorbrewer
```

This is because the package `demo` in module `biojava-structure-gui` conflicts with the same package in the third-party library `jcolorbrewer`. Since the package `demo` is not important, you can exclude it in Maven compiler plugin; see Listing 2-58.

Listing 2-58. Excluding `demo` Packages

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.7.0</version>
  <configuration>
    <excludes>
      <exclude>demo/**</exclude>
    </excludes>
  </configuration>
</plugin>
```

Now you can build the project successfully. Once you migrate to JPMS modules, you can remove the `--add-modules java.se.ee` in the configuration of the Maven compiler and the surefire plugins.

Summary

As the most important new feature in Java 9, JPMS has an impact on different aspects of the Java platform. This chapter started with the module declarations, and then went on to discuss unnamed modules and automatic modules. I also covered JDK tools and related concepts. In addition, you learned that with the module-related API, you can interact with the module system in Java programs. Finally, this chapter provided information about the migration path and a concrete example of how to migrate to Java 9 modules. In next chapter, we'll discuss the REPL shell in Java 9—jshell.

CHAPTER 3

jshell

jshell is a useful utility tool that adds the Read-Eval-Print Loop (REPL) to Java. It allows developers to try Java language features interactively or to evaluate some expressions quickly. For example, it's common for Java programs to write files to the system's temporary directory. Sometimes you may want to check the temporary directory to verify the contents of created files. To get the path of the temporary directory, you need to get the value of the Java system property, `java.io.tmpdir`. You can do this by creating a simple Java program that has a main method to output the value, or you can use the Apache Groovy Console (http://groovy-lang.org/groovyconsole.html) to quickly run the code. But writing another Java program seems like overkill for simple tasks like this, and using Groovy Console requires that you downloaded and install Groovy. Neither approach seems ideal. Now you have a better choice in Java 9—jshell.

jshell is a built-in tool in Java 9. You can start it by running the command `jshell`.

```
$ jshell
```

Then you can just type `System.getProperty("java.io.tmpdir")` to get the value; see Listing 3-1.

Listing 3-1. Typing an Expression in jshell

```
jshell> System.getProperty("java.io.tmpdir")
$1 ==> "/var/folders/27/9mrlr7kd3pdgv8cw9fq9_1z40000gn/T/"
```

You may notice the `$1` in the output. This is the reference to the result of the expression you just typed. You can use this reference later in the shell. In Listing 3-2, type `$1.length()` to get the length of `$1` and another reference `$2`. `$2` now references the length of the string referenced by `$1`.

Listing 3-2. Using Result References

```
jshell> $1.length()
$2 ==> 49
```

If you prefer to use a properly named reference, you can create the variable and assign the value to it; see Listing 3-3.

Listing 3-3. Using Variables

```
jshell> String tmpdir = System.getProperty("java.io.tmpdir")
tmpdir ==> "/var/folders/27/9mrlr7kd3pdgv8cw9fq9_1z40000gn/T/"
```


F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_3

Code Completion

jshell provides the basic support for code completion. When you press the <Tab> key, you can see a list of possible choices. For example, if you type `System.get` and press the <Tab> key, jshell lists all possible methods in `System` that start with `get`.

```
jshell> System.get
getLogger(             getProperties()        getProperty(
  getSecurityManager()   getenv(
```

The code is completed automatically to the longest common prefix of all choices. For example, if you type `System.getP` and press the <Tab> key, the code is completed to `System.getPropert` with two choices: `getProperties()` and `getProperty(`.

Classes

Apart from simple expressions, you can also create classes in the shell. In Listing 3-4, let's create an abstract class `Shape` with the abstract method `getArea()`.

Listing 3-4. Creating the Class `Shape`

```
jshell> abstract class Shape {
   ...>   protected abstract double getArea();
   ...> }
|  created class Shape
```

Then create a new class `Circle` that inherits from `Shape`; see Listing 3-5.

Listing 3-5. Creating the Class `Circle`

```
jshell> class Circle extends Shape {
   ...>   private final double radius;
   ...>   public Circle(final double radius) {
   ...>     this.radius = radius;
   ...>   }
   ...>   public double getArea() {
   ...>     return Math.PI * this.radius * this.radius;
   ...>   };
   ...> }
|  created class Circle
```

Now you can create new instances of the class `Circle` and invoke its method; see Listing 3-6.

Listing 3-6. Creating Instances of the Class `Circle`

```
jshell> Circle circle1 = new Circle(10);
circle1 ==> Circle@2038ae61

jshell> circle1.getArea()
$7 ==> 314.1592653589793
```

Methods

You can also add methods in jshell. In Listing 3-7, let's create a method `add()`.

Listing 3-7. Adding Methods

```
jshell> int add(int x, int y) {
   ...>   return x + y;
   ...> }
|  created method add(int,int)
```

You can invoke this method as shown in Listing 3-8.

Listing 3-8. Invoking Methods

```
jshell> add(1, 2)
$19 ==> 3
```

Commands

By typing `/help` in jshell, you can see a list of available commands. For a command, for example, `/list`, you can use `/help /list` to view a detailed help message for the command.

I'll discuss these commands in following sections.

/list

`/list` lists all the snippets. Each snippet has an ID. There are three types of snippets:

- *Start snippets* are evaluated automatically during start-up. `/list -start` shows only start snippets.
- *Active snippets* are snippets you have typed. `/list` shows active snippets.
- *Error snippets* are snippets you have typed but have failed to compile.

You can use `/list -all` to list all snippets; see Listing 3-9. Each snippet is prefixed with an ID. The ID of a start snippet is prefixed with `s`, whereas the ID of an error snippet is prefixed with `e`. You can see IDs like `s1`, `s2` or `e3`. The ID of an active snippet matches the reference name of its evaluated result. For example, you can use `$7` to reference the evaluated result of the snippet with ID `7`. If you want to reference a previous snippet, you can use `/list` to find the ID first, then use `$<id>` to reference its result.

You can also use `/list <snippet id>` to list the source of a snippet by ID, for example, `/list s1` or `/list 4`.

Listing 3-9. Output of `/list -all`

```
s1 : import java.io.*;
s2 : import java.math.*;
s3 : import java.net.*;
s4 : import java.nio.file.*;
s5 : import java.util.*;
s6 : import java.util.concurrent.*;
s7 : import java.util.function.*;
```

```
s8 : import java.util.prefs.*;
s9 : import java.util.regex.*;
s10 : import java.util.stream.*;
 1 : System.getProperty("java.io.tmpdir")
 2 : $1.length()
e1 : tmpdir = System.getProperty("java.io.tmpdir")
 3 : String tmpdir = System.getProperty("java.io.tmpdir");
 4 : abstract class Shape {
       protected abstract double getArea();
     }
```

/edit

`/edit` edits a snippet by ID—for example, you would use `/edit 1` to edit the snippet with and ID of `1`. A new window opens with the current source of the snippet; see Figure 3-1.

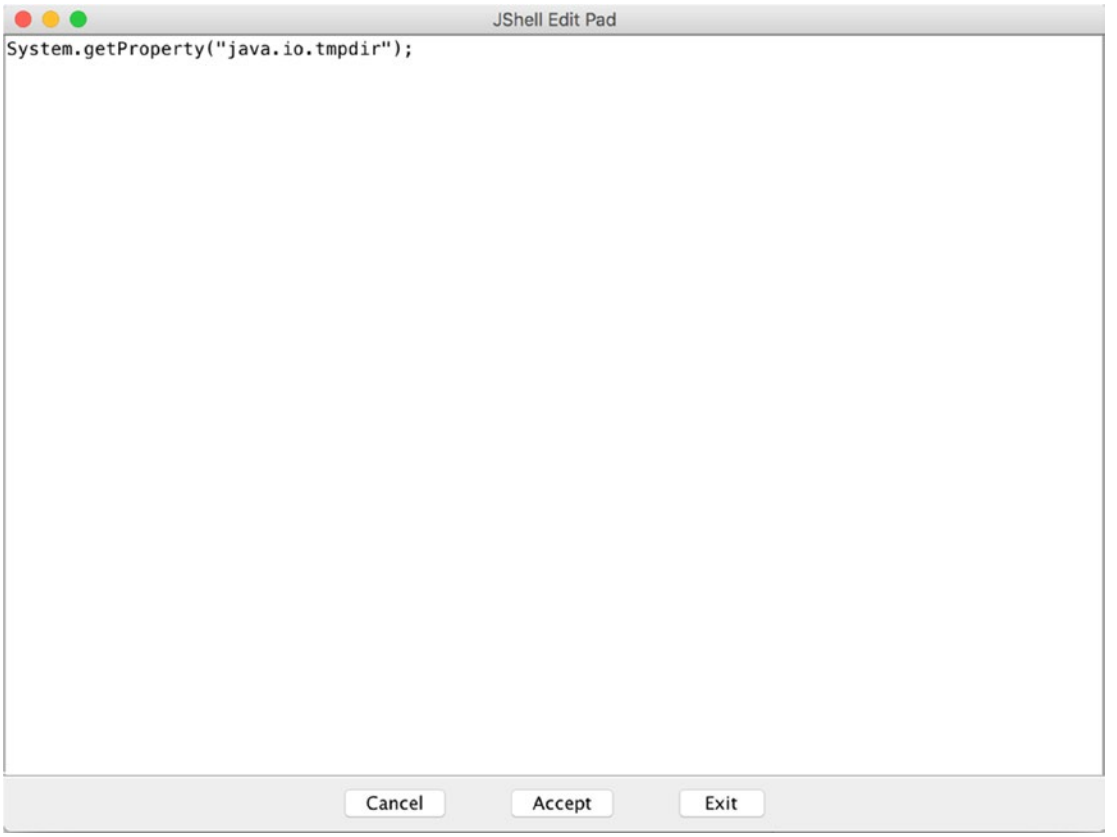

Figure 3-1. *The JShell edit pad*

If you update the source to `System.getProperty("os.name");` in the new window and click **Accept** to save it, a new snippet is created and run.

/drop

`/drop` deletes a snippet by ID; for example, use `/drop 1` to delete the snippet with an ID of `1`, as shown in Listing 3-10.

Listing 3-10. Deleting a Snippet by ID

```
jshell> /drop 23
|  dropped variable $23
```

/save

`/save` saves snippets and commands to a file. You can choose to save all snippets, active snippets, or start snippets using `/save -all`, `/save` or `/save -start`, respectively. You can also use `/save -history` to save commands; see Listing 3-11.

Listing 3-11. Saving Snippets

```
jshell> /save -history ~/Downloads/snippets.txt
```

/open

`/open` opens a file and uses its content as the input source. For example, you can create a file `rectangle.txt` with the content in Listing 3-12.

Listing 3-12. Content of the File `rectangle.txt`

```
class Rectangle extends Shape {
    private final double width;
    private final double height;
    public Rectangle(final double width, final double height) {
        this.width = width;
        this.height = height;
    }

    public double getArea() {
        return this.width * this.height;
    }
}
```

Then you can use `/open` to open it as shown in Listing 3-13.

Listing 3-13. Using `/open` to Open a File

```
/open ~/Downloads/rectangle.txt
```

When using `/list` to check snippets, you can find out that the file has been added as a new snippet with ID `27`. Now you can use the class `Rectangle`; see Listing 3-14.

Listing 3-14. Using the Class `Rectangle`

```
jshell> Rectangle rectangle = new Rectangle(10, 5)
rectangle ==> Rectangle@11438d26

jshell> rectangle.getArea()
$29 ==> 50.0
```

/imports

`/imports` lists all imported items; see Listing 3-15.

Listing 3-15. Listing All Imported Items

```
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.*
```

/vars

`/vars` lists all declared variables and their values; see Listing 3-16. It also supports the flags `-all` and `-start`.

Listing 3-16. Listing All Declared Variables

```
jshell> /vars
|    String $1 = "/var/folders/27/9mrlr7kd3pdgv8cw9fq9_1z40000gn/T/"
|    int $2 = 49
|    String tmpdir = "/var/folders/27/9mrlr7kd3pdgv8cw9fq9_1z40000gn/T/"
|    Circle circle1 = Circle@2038ae61
|    double $7 = 314.1592653589793
```

/types

`/types` lists all declared types; see Listing 3-17. It also supports the flags `-all` and `-start`.

Listing 3-17. Listing All Declared Types

```
jshell> /types
|    class Shape
|    class Circle
|    class Rectangle
```

/methods

/methods lists all declared methods; see Listing 3-18. It also supports the flags -all and -start.

Listing 3-18. Listing All Declared Methods

```
jshell> /methods
|    int add(int,int)
```

/history

/history lists all of you typed; see Listing 3-19.

Listing 3-19. List All of You Typed

```
jshell> /history

System.getProperty("java.io.tmpdir")
$1.length()
tmpdir = System.getProperty("java.io.tmpdir")
String tmpdir = System.getProperty("java.io.tmpdir")
```

/env

/env shows or modifies jshell's evaluation context. You can use /env to show the configuration of the evaluation context. To modify the context, you need to pass at least one value of -class-path, -module-path, -add-modules, and -add-exports.

The option -class-path sets the class path of the context. For example, if you want to test Guava in jshell, you add its JAR file to the class path; see Listing 3-20.

Listing 3-20. Adding a JAR File to the Class Path

```
jshell> /env -class-path ~/Downloads/libs/guava-19.0.jar
```

Now you can use classes from Guava; see Listing 3-21.

Listing 3-21. Using Classes from the JAR File

```
jshell> import com.google.common.collect.Lists

jshell> Lists.newArrayList("hello", "world")
$27 ==> [hello, world]
```

The options -module-path, -add-modules, and -add-exports have the same meanings as the ones used in javac or java in Chapter 2.

/set

`/set` configures the jshell. Typing `/set` shows the current configuration. `/set` has several subcommands:

- `/set editor`: Specifies the editor to use for the `/edit` command; for example, type `/set editor -wait atom` to use the Atom editor (https://atom.io/). It's important to use the `-wait` flag to make jshell wait for the editor to close, or the changes won't be captured by jshell.
- `/set start`: Sets the file to be the default start snippets and commands. If you have some setup code to run before using jshell, you can put that code into a file and use `/set start` to configure it to run first.
- `/set feedback`: Sets the feedback mode. Possible values are `normal`, `concise`, `silent`, and `verbose`. This configuration controls how much information jshell provides. For example, if you set the mode to `verbose`, jshell provides more information when running a snippet; see Listing 3-22.

Listing 3-22. `verbose` Feedback Mode

```
jshell> System.getProperty("java.io.tmpdir")
$4 ==> "/var/folders/27/9mrlr7kd3pdgv8cw9fq9_1z40000gn/T/"
|  created scratch variable $4 : String
```

jshell allows fine-grained control of the feedback mode, including the displayed prompt, maximum length of a displayed value, and the format of a field. These can be configured using following commands:

- `/set prompt <mode>`: Sets the displayed prompt
- `/set truncation <mode>`: Sets the maximum length of a displayed value
- `/set format <mode> <field>`: Sets the format of a field

Typing these commands directly shows the current configuration.

/reset

`/reset` resets jshell's internal state. After the reset, all normal and error snippets and variables are removed. Start snippets are re-executed. The configuration changes made by `/set` are kept. `/reset` supports the same options as `/env`: `-class-path`, `-module-path`, `-add-modules`, and `-add-exports`.

/reload

`/reload` resets jshell and replays each valid snippet and `/drop` command in the order they were entered. `/reload` supports two modes, normal mode and restore mode. In the normal mode with `/reload`, it only replays valid history since the last time jshell was entered, or since a `/reset` or `/reload` command was executed. In the restore mode with `/reload -restore`, it replays the valid history between the previous and most recent time that jshell was entered, or since a `/reset` or `/reload` command was executed. The restore mode can replay history from last jshell session. When the option `-quiet` is passed, `/reload` won't show output of the replay, but errors will still be displayed. `/reload` also supports the same options as`/env`: `-class-path`, `-module-path`, `-add-modules`, and `-add-exports`.

After the evaluation context is modified using `/env`, the history will be replayed silently, just like when the command `/reload -quiet` is invoked.

/!

This reruns the last snippet.

/<id>

This reruns the snippet with the given ID.

/-<n>

This reruns the *n*th snippet.

/exit

This exits the shell.

Summary

jshell is a very useful tool for developers to use to quickly test code and verify results. This chapter covered all jshell's features in detail, including all the available commands. In next chapter, I'll discuss the changes to the collections `Stream` and `Optional` in Java 9.

CHAPTER 4

Collections, Stream, and Optional

This chapter covers topics related to Java 9 changes in collections, `Stream`, and `Optional`.

Factory Methods for Collections

Java 9 adds some convenient methods you can use to create immutable collections.

The `List.of()` Method

The new static method `List.of()` has a number of overloads that you can use to create immutable lists from zero or many elements; see Listing 4-1.

Listing 4-1. Example of `List.of()`

```
List.of();
List.of("Hello", "World");
List.of(1, 2, 3);
```

`List.of()` doesn't allow `null` values. Passing `null` to it results in `NullPointerException`.

The `Set.of()` Method

The new static method `Set.of()` also has a number of overloads for creating immutable sets from zero or many elements; see Listing 4-2.

Listing 4-2. Example of `Set.of()`

```
Set.of();
Set.of("Hello", "World");
Set.of(1, 2, 3);
```

Because sets don't allow duplicate elements, passing duplicate elements to `Set.of()` causes it to throw `IllegalArgumentException`. `Set.of()` doesn't allow `null` values. Passing `null` to it results in `NullPointerException`.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_4

The `Map.of()` and `Map.ofEntries()` Methods

Two new static methods, `Map.of()` and `Map.ofEntries()`, are used to create immutable maps with zero or many entries.

`Map.of()` has 11 different overloads for creating maps with zero to ten entries; see Listing 4-3. Keys and values are specified in pairs.

Listing 4-3. Example of `Map.of()`

```
Map.of(); // empty map
Map.of("Hello", 1, "World", 2); // -> Map<String, Integer> with two entries
```

`Map.ofEntries()` can create maps from any number of `Map.Entry` objects; see Listing 4-4.

Listing 4-4. Example of `Map.ofEntries()`

```
Map.ofEntries(
    new AbstractMap.SimpleEntry<>("Hello", 1),
    new AbstractMap.SimpleEntry<>("World", 2)
);
```

`Map.of()` and `Map.ofEntries()` don't allow `null` keys or values. Passing duplicate keys causes them to throw `IllegalArgumentExceptions`. Passing `null` to them results in `NullPointerExceptions`.

Arrays

This section discusses the methods that have been added to `java.util.Arrays` in Java 9.

`Mismatch()` Methods

The group of `mismatch()` methods finds and returns the index of the first mismatch between two arrays. `mismatch()` returns `-1` if there is no mismatch. There are different overloads for `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `Object` arrays. You can also specify the start and end index in both arrays for checking the mismatch. For object arrays, you can also specify a `java.util.Comparator` object for the comparison. In Listing 4-5, you can see different use cases of `mismatch()` being tested.

Listing 4-5. Example of `mismatch()`

```
@Test
public void testMismatch() throws Exception {
  assertEquals(0, Arrays.mismatch(new int[]{1}, new int[]{2}));
  assertEquals(1, Arrays.mismatch(new int[]{1}, new int[]{1, 2}));
  assertEquals(1, Arrays.mismatch(new int[]{1, 3}, new int[]{1, 2}));
  assertEquals(-1, Arrays.mismatch(
      new int[]{1, 3}, 0, 1,
      new int[]{1, 2}, 0, 1));
}
```

Compare() Methods

The group of compare() methods compares two arrays lexicographically. There are different overloads for boolean, byte, char, double, float, int, long, short, and Object arrays. You can also specify the start and end index in both arrays for comparison. For object arrays, you can also specify a java.util.Comparator object for the comparison. Listing 4-6 tests different use cases of compare().

Listing 4-6. Example of compare()

```
@Test
public void testCompare() throws Exception {
  assertEquals(0, Arrays.compare(new int[]{1}, new int[]{1}));
  assertTrue(Arrays.compare(new int[]{0}, new int[]{1}) < 0);
  assertTrue(Arrays.compare(new int[]{1}, new int[]{0}) > 0);
  assertEquals(0, Arrays.compare(
      new int[]{1, 3}, 0, 1,
      new int[]{1, 2}, 0, 1));
}
```

Equals() Methods

The group of equals() methods returns true if two arrays are equal to each other. There are different overloads for boolean, byte, char, double, float, int, long, short, and Object arrays. You can also specify the start and end index in both arrays for an equality check. For object arrays, you can also specify a java.util.Comparator object for the comparison. Listing 4-7 tests different use cases of equals().

Listing 4-7. Example of equals()

```
@Test
public void testEquals() throws Exception {
  assertTrue(Arrays.equals(new int[]{1}, new int[]{1}));
  assertTrue(Arrays.equals(
      new int[]{1, 2}, 0, 1,
      new int[]{1, 3}, 0, 1));
}
```

Stream

This section covers the new methods that have been added to java.util.stream.Stream in Java 9.

The ofNullable() Method

The static method Stream<T> ofNullable(T t) returns a Stream of zero or one element, depending on whether the input value is null. Listing 4-8 shows examples of this method.

Listing 4-8. Example of `Stream.ofNullable()`

```
@Test
public void testOfNullable() throws Exception {
  assertEquals(1, Stream.ofNullable("").count());
  assertEquals(0, Stream.ofNullable(null).count());
}
```

The `dropWhile()` Method

The method `Stream<T> dropWhile(Predicate<? super T> predicate)` drops elements in the stream until it encounters the first element that doesn't match the predicate. If the stream is ordered, then the longest prefix of elements that matches the given predicate is dropped. If the stream is unordered, and some (but not all) elements of this stream match the given predicate, then it's undetermined to tell which elements are dropped. However, if all elements of the stream match the given predicate, or no elements of the stream match the given predicate, then it doesn't matter whether the stream is ordered or unordered; the result is determined.

In Listing 4-9, the stream contains elements from `1` to `5`. The predicate checks if the element is odd. The first element `1` is dropped. The result stream contains four elements.

Listing 4-9. Example of `Stream.dropWhile()`

```
@Test
public void testDropWhile() throws Exception {
  final long count = Stream.of(1, 2, 3, 4, 5)
      .dropWhile(i -> i % 2 != 0)
      .count();
  assertEquals(4, count);
}
```

The `takeWhile()` Method

The method `Stream<T> takeWhile(Predicate<? super T> predicate)` performs the opposite action on a stream that `dropWhile()` does. `takeWhile()` takes elements in the stream until it encounters the first element that doesn't match the predicate. `takeWhile()` has the same behavior as `dropWhile()` when it processes ordered and unordered streams.

In Listing 4-10, when the stream is processed as in Listing 4-6 using `takeWhile()`, the resulting stream only has one element of value `1`.

Listing 4-10. Example of `Stream.takeWhile()`

```
@Test
public void testTakeWhile() throws Exception {
  final long count = Stream.of(1, 2, 3, 4, 5)
      .takeWhile(i -> i % 2 != 0)
      .count();
  assertEquals(1, count);
}
```

The iterate() Method

The static method `Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)` is a new way to generate streams using the iterator pattern. `seed` is the initial element for iteration. `hasNext` is the predicate that applies to the current element to check if the stream should be terminated. `next` is the function that applies to the current element to produce the next element in the stream. If the initial element `seed` doesn't match the predicate `hasNext`, the result stream is empty.

In Listing 4-11, the result stream uses `2` as the initial element and increases the current value by `2` to produce the next value. The stream terminates when the value is greater than `10`. There will be `5` elements in the stream.

Listing 4-11. Example of `Stream.iterate()`

```
@Test
public void testIterate() throws Exception {
  final Stream<Integer> stream =
      Stream.iterate(2, i -> i <= 10, i -> i + 2);
  assertEquals(5, stream.count());
}
```

IntStream, LongStream, and DoubleStream

Methods `dropWhile()`, `takeWhile()`, and `iterate()` are also added to `IntStream`, `LongStream`, and `DoubleStream`.

Collectors

Several new methods have also been added to `java.util.stream.Collectors` in Java 9.

The filtering() Method

The method `<T, A, R> Collector<T, ?, R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)` filters input elements by applying the predicate function, and it only accumulates elements to the downstream `Collector` when the predicate returns `true`.

Listing 4-12 shows how to use `filtering()` to filter a stream of `Strings` by its length and collect them into a `Set`. This actually can be simplified to use `Stream.filter()` first and then collect the `Strings` into a `Set`.

Listing 4-12. Simple Example of `Collectors.filtering()`

```
@Test
public void testSimpleFiltering() throws Exception {
  final Set<String> result = Stream.of("a", "bc", "def")
      .collect(Collectors.filtering(
          v -> v.length() > 1,
          Collectors.toSet()));
  assertEquals(2, result.size());
}
```

Listing 4-13 shows a more complicated example of `filtering()`. The variable `users` is a map of users' names to their ages. This example uses `groupingBy()` to group the map entries by the first character of the name. Then it used `filtering()` to only keep entries with ages greater than `18`. Finally, it uses `mapping()` to map the entries into the names. The result is a map of each name's first character to the `Set` of names. Because the age of `Bob` is only `16`, the value of key `B` in the result map is an empty `Set`.

Listing 4-13. Complicated Example of `Collectors.filtering()`

```
@Test
public void testComplicatedFiltering() throws Exception {
  final Map<String, Integer> users = Map.of(
      "Alex", 30,
      "Bob", 16,
      "David", 50
  );
  final Map<String, Set<String>> result = users
      .entrySet()
      .stream()
      .collect(Collectors.groupingBy(entry -> entry.getKey().substring(0, 1),
          Collectors.filtering(entry -> entry.getValue() > 18,
              Collectors.mapping(
                  Map.Entry::getKey,
                  Collectors.toSet()))));
  assertEquals(1, result.get("A").size());
  assertEquals(0, result.get("B").size());
  assertEquals(1, result.get("D").size());
}
```

The `flatMapping()` Method

The method `<T, U, A, R> Collector<T, ?, R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)` applies a flat mapping function to the input elements and accumulates elements in the mapped streams to the downstream `Collector`.

In Listing 4-14, given a stream of `Strings`, you can see `flatMapping()` used to map a `String` to a stream of `Integers` and then collect all `Integers` into a `Set`. The result `Set` only contains three elements.

Listing 4-14. Example of `Collectors.flatMapping()`

```
@Test
public void testFlatMapping() throws Exception {
  final Set<Integer> result = Stream.of("a", "ab", "abc")
      .collect(Collectors.flatMapping(v -> v.chars().boxed(),
          Collectors.toSet()));
  assertEquals(3, result.size());
}
```

Optional

Three new methods have been added to `java.util.Optional`.

The `ifPresentOrElse()` Method

The method `void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)` performs the given `action` if the value is present, otherwise it performs the given action `emptyAction`.

In Listing 4-15, the method `checkValue()` increases or decreases the `count` based on whether the value is present. The first invocation of `checkValue()` decreases the `count`, whereas the second invocation increases the `count`.

Listing 4-15. Example of `Optional.ifPresentOrElse()`

```
public class OptionalTest {

  private AtomicInteger count;

  @Before
  public void setUp() throws Exception {
    this.count = new AtomicInteger();
  }

  @Test
  public void testIfPresentOrElse() throws Exception {
    checkValue(Optional.empty());
    assertEquals(-1, this.count.get());
    checkValue(Optional.of(1));
    assertEquals(0, this.count.get());
  }

  private void checkValue(final Optional<Integer> value) {
    value.ifPresentOrElse(
        v -> this.count.incrementAndGet(),
        () -> this.count.decrementAndGet()
    );
  }
}
```

The `Optional.or()` Method

The method `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)` returns an `Optional` with the value if a value is present, otherwise it returns an `Optional` produced by the supplier function. Listing 4-16 shows the example of this method.

Listing 4-16. Example of `Optional.or()`

```
@Test
public void testOr() throws Exception {
  assertTrue(Optional.empty().or(() -> Optional.of(1)).isPresent());
}
```

The stream() Method

The method Stream<T> stream() converts the Optional into a Stream. If a value is present, then the result stream contains only the value, otherwise the result stream is empty. This method is useful when you're working with flatMap() to convert a Stream of Optionals into a Stream of present value elements.

In Listing 4-17, the stream of Optionals contains three elements, but only two of them have values. After using flatMap(), the result stream contains the two present values.

Listing 4-17. Example of Optional.stream()

```
@Test
public void testStream() throws Exception {
  final long count = Stream.of(
      Optional.of(1),
      Optional.empty(),
      Optional.of(2)
  ).flatMap(Optional::stream)
      .count();
  assertEquals(2, count);
}
```

Methods ifPresentOrElse() and stream() are also added to classes java.util.OptionalInt, java.util.OptionalDouble, and java.util.OptionalLong.

Summary

This chapter covered changes related to collections, Stream, and Optional in Java 9. The new factory methods for creating immutable collections can save a lot of code in Java programs. The new methods added to Arrays, Stream, and Optional are also very useful. In the next chapter, we'll discuss the new Process API.

CHAPTER 5

The Process API

The Java Process API allows developers to create and manage native processes. You can now use the `java.lang.ProcessBuilder` from JDK 5 to create processes and redirect the output and error streams. The new interface `java.lang.ProcessHandle` in Java 9 allows you to control the native processes created by `ProcessBuilder.start()`.

The `ProcessHandle` Interface

The fine-grained control provided by `ProcessHandle` is very useful for long-running processes. Table 5-1 shows the methods of `ProcessHandle`. With the `ProcessHandle` interface, you can query the information about the native process and control its life cycle.

Table 5-1. *Methods of `ProcessHandle`*

Method	Description
`long pid()`	Returns the native process ID.
`ProcessHandle.Info info()`	Returns `ProcessHandle.Info`, which contains an information snapshot about the process.
`boolean isAlive()`	Checks if the process is alive.
`Optional<ProcessHandle> parent()`	Returns the parent process.
`Stream<ProcessHandle> children()`	Returns all the children processes.
`Stream<ProcessHandle> descendants()`	Returns all the descendant processes.
`boolean destroy()`	Kills the process. It returns `true` when the termination was requested successfully.
`boolean destroyForcibly()`	Kills the process forcibly. It returns `true` when the termination was requested successfully.
`boolean supportsNormalTermination()`	Checks if the process can be terminated normally using `destroy()`.
`CompletableFuture<ProcessHandle> onExit()`	Returns a `CompletableFuture<ProcessHandle>` that can be used to run custom actions when the process is terminated.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_5

Most of the methods in Table 5-1 are straightforward. The return value of `info()` is an object of class `ProcessHandle.Info`. Table 5-2 shows the methods of `ProcessHandle.Info`.

Table 5-2. *Methods of `ProcessHandle.Info`*

Method	Description
`Optional<String[]> arguments()`	Returns the arguments passed to the process
`Optional<String> command()`	Returns the path of the executable of the process
`Optional<String> commandLine()`	Returns the command line to start the process
`Optional<Instant> startInstant()`	Returns the start time of the process as `java.time.Instant`
`Optional<Duration> totalCpuDuration()`	Returns the total CPU time of the process as `java.time.Duration`
`Optional<String> user()`	Returns the user of this process

`ProcessHandle` also has some static methods to get `ProcessHandles` related to the current process; see Table 5-3.

Table 5-3. *Static Methods of `ProcessHandle`*

Method	Description
`ProcessHandle current()`	Returns the `ProcessHandle` of the current process
`Optional<ProcessHandle> of(long pid)`	Returns the `ProcessHandle` of an existing process by its PID
`Stream<ProcessHandle> allProcesses()`	Returns a snapshot of all processes that are visible to the current process

In Listing 5-1, the method `printProcessInfo()` prints out the information of the current native `java` process.

Listing 5-1. Printing the Native Process Info

```
import java.util.Arrays;

public class CurrentProcess {
  public static void main(String[] args) {
    new CurrentProcess().printProcessInfo(ProcessHandle.current());
  }

  private void printProcessInfo(final ProcessHandle processHandle) {
    final ProcessHandle.Info info = processHandle.info();
    System.out.println("Process info =>");
    System.out.format("PID: %s%n", processHandle.pid());
    info.arguments().ifPresent(args ->
      System.out.format("Arguments: %s%n", Arrays.toString(args)));
    info.command().ifPresent(command ->
      System.out.format("Command: %s%n", command));
    info.commandLine().ifPresent(commandLine ->
```

```
      System.out.format("Command line: %s%n", commandLine));
    info.startInstant().ifPresent(startInstant ->
      System.out.format("Start time: %s%n", startInstant));
    info.totalCpuDuration().ifPresent(cpuDuration ->
      System.out.format("CPU time: %s%n", cpuDuration));
    info.user().ifPresent(user ->
      System.out.format("User: %s%n", user));
  }
}
```

The output looks like Listing 5-2 when running use `java`.

Listing 5-2. Output of Info About the Current `java` Process

```
Process info =>
PID: 14946
Arguments: [-cp, feature9, io.vividcode.feature9.process.CurrentProcess]
Command: /Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home/bin/java
Command line: /Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home/bin/java
  -cp feature9 io.vividcode.feature9.process.CurrentProcess
Start time: 2017-05-05T04:01:22.686Z
CPU time: PT0.217612S
User: alexcheng
```

Process

`Process` also has several new methods; see Table 5-4. Some of these methods have the same name and functionality as they do in `ProcessHandle`.

Table 5-4. *Methods of* `Process`

Method	Description
`long pid()`	Returns the native process ID
`ProcessHandle.Info info()`	Returns `ProcessHandle.Info`, which contains an information snapshot about the process
`boolean isAlive()`	Checks if the process is alive
`ProcessHandle toHandle()`	Returns a `ProcessHandle` for this process
`Stream<ProcessHandle> children()`	Returns all the children processes
`Stream<ProcessHandle> descendants()`	Returns all the descendant processes
`boolean supportsNormalTermination()`	Checks if the process can be terminated normally using `destroy()`
`CompletableFuture<ProcessHandle> onExit()`	Returns a `CompletableFuture<ProcessHandle>` that can be used to run custom actions when the process is terminated

Managing Long-Running Processes

With the new method `onExit()` of `ProcessHandle`, it's now much easier to manage long-running processes. Listing 5-3 shows a simple example. In the method `start()`, I create a new process with the command `top` (https://linux.die.net/man/1/top) and it returns the `ProcessHandle` of the process. In the method `waitFor()`, I use the method `whenCompleteAsync()` of the `CompletableFuture` object returned by `onExit()` to add a handler that invokes when the process exits. In the handler, I can get the `ProcessHandle` object of the terminated process. Here I just output the PID of this process. The variable `shutdownThread` represents a thread that destroys the process after a one-second delay. I can check if the `ProcessHandle` supports graceful termination using `supportsNormalTermination()` and try to terminate it gracefully. I use the `CountDownLatch` to wait for the asynchronous completion handler to finish.

Listing 5-3. Managing Long-Running Processes

```
public class LongRunningProcess {

  public ProcessHandle start() throws IOException {
    final ProcessBuilder processBuilder = new ProcessBuilder("top")
        .inheritIO();
    return processBuilder.start().toHandle();
  }

  public void waitFor(final ProcessHandle processHandle) {
    final CountDownLatch latch = new CountDownLatch(1);
    processHandle.onExit().whenCompleteAsync((handle, throwable) -> {
      if (throwable == null) {
        System.out.println(handle.pid());
      } else {
        throwable.printStackTrace();
      }
      latch.countDown();
    });
    final Thread shutdownThread = new Thread(() -> {
      try {
        Thread.sleep(1000);
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
      if (processHandle.supportsNormalTermination()) {
        processHandle.destroy();
      } else {
        processHandle.destroyForcibly();
      }
    });
    shutdownThread.start();
    try {
      shutdownThread.join();
      latch.await();
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }
  }
```

```
  public static void main(final String[] args) {
    final LongRunningProcess longRunningProcess = new LongRunningProcess();
    try {
      longRunningProcess.waitFor(longRunningProcess.start());
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

When you run the code in Listing 5-3, you should see the output of `top` command for about one second, then the PID of the process displays and the program terminates.

Summary

In this chapter, we discussed the new `ProcessHandle` API that can provide control of native processes. I also showed you how to manage long-running processes using this new API. In next chapter, we'll discuss the platform logging API and service.

CHAPTER 6

The Platform Logging API and Service

Logging is an important aspect of application development. Many logging frameworks are available for use in the Java platform, including popular choices like Apache Log4j 2 (https://logging.apache.org/log4j/2.x/) and Logback (https://logback.qos.ch/). SLF4J is commonly used as the facade of different logging implementations. Java also has its own logging implementation, the `java.util.logging` API, which was added in Java 1.4. Even though `java.util.logging` is the built-in logging solution used by the Java standard library, it's not very popular. Most Java applications still use external logging frameworks. The JDK itself and the applications running on top of it may use different logging frameworks. If an application error is actually related to the Java standard library, even though such an error is unlikely, then two separate logging frameworks will make the error diagnostics much harder. In situations in which you are using two separate logging frameworks, you will also need to configure both logging frameworks, and their configurations are likely to have duplications.

In Java 9, it's now possible to use the same logging framework for both the JDK and the application by using the class `java.lang.System.LoggerFinder`. `LoggerFinder` is responsible for managing loggers used by the JDK. The JVM has a single system-wide `LoggerFinder` instance that is loaded using the service provider mechanism with `ServiceLoader`. If no `LoggerFinder` provider is found, the default `LoggerFinder` implementation will use `java.util.logging` as the logging framework when the module `java.logging` is present. If the module `java.logging` is not present, then the default `LoggerFinder` implementation outputs log messages to the console using `System.err`.

The static method `getLoggerFinder()` of `LoggerFinder` returns the single `LoggerFinder` instance in the JVM. Subclasses of `LoggerFinder` need to implement the abstract method `System.Logger getLogger(String name, Module module)`, which returns the logger instance of interface `java.lang.System.Logger` for a given module. `System.Logger` has different methods for logging messages of different levels. `LoggerFinder` also has the method `System.Logger getLocalizedLogger(String name, ResourceBundle bundle, Module module)` to return a localizable logger instance for a given module.

`System.Logger` is the primary class used by applications to log messages. Table 6-1 shows methods of `System.Logger`.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_6

Table 6-1. *Methods of* `System.Logger`

Method	Description
`String getName()`	Returns the name of this logger.
`boolean isLoggable(System.Logger.Level level)`	Checks if a log message of the given level would be logged by this logger.
`void log(System.Logger.Level level, String msg)`	Logs a message of the given level.
`void log(System.Logger.Level level, Supplier<String> msgSupplier)`	Logs a message of the given level that is produced by the given supplier function. The supplier function is only called when the message will be logged.
`void log(System.Logger.Level level, Object obj)`	Logs a message of the given level by calling the object's `toString()` method.
`void log(System.Logger.Level level, String msg, Throwable thrown)`	Logs a message of the given level with a given throwable.
`void log(System.Logger.Level level, Supplier<String> msgSupplier, Throwable thrown)`	Logs a message of the given level that is produced by the given supplier function with a given throwable.
`void log(System.Logger.Level level, String format, Object… params)`	Logs a message generated from the format and an optional list of parameters. The message uses the format specified in `MessageFormat`.
`void log(System.Logger.Level level, ResourceBundle bundle, String msg, Throwable thrown)`	Logs a localized message of the given level with a given throwable.
`void log(System.Logger.Level level, ResourceBundle bundle, String format, Object… params)`	Logs a localized message generated from the format and an optional list of parameters.

`System.Logger` defines its own enum for logging levels in `System.Logger.Level`. These levels can be easily mapped to logging levels in other logging frameworks. These logging levels are `ALL`, `TRACE`, `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `OFF`.

Default `LoggerFinder` Implementation

Let's first take a closer look at the default `LoggerFinder` implementation.

The class `jdk.internal.logger.LoggerFinderLoader` in the module `java.base` is responsible for loading the `LoggerFinder` implementation. It first tries to use `ServiceLoader` to load the implementation of `LoggerFinder`. If no provider exists, it uses the `ServiceLoader` again to load an implementation of the class `jdk.internal.logger.DefaultLoggerFinder`. If it still cannot find any providers, it uses a new instance of `DefaultLoggerFinder`. `DefaultLoggerFinder` itself is a logging implementation that uses the class `jdk.internal.logger.SimpleConsoleLogger` to write log messages to the console using `System.err`.

The class `sun.util.logging.internal.LoggingProviderImpl` in the module `java.logging` is an implementation of `DefaultLoggerFinder` that delegates the logging requests to the underlying loggers created from `java.util.logging`; see the module declaration of `java.logging` in Listing 6-1.

Listing 6-1. Module Declaration of `java.logging`

```
module java.logging {
  exports java.util.logging;

  provides jdk.internal.logger.DefaultLoggerFinder with
      sun.util.logging.internal.LoggingProviderImpl;
}
```

Creating Custom `LoggerFinder` Implementations

Now you can create your own `LoggerFinder` implementations. Since `LoggerFinder` can change the system's default logging implementation, when a `SecurityManager` is installed, the implementation of `LoggerFinder` should check the permission `loggerFinder` first.

Here I am going to use SLF4J with Logback as the logging framework to demonstrate how to create a custom `LoggerFinder`. In Listing 6-2, the class `SLF4JLoggerFinder` is the `LoggerFinder` implementation. The method `checkPermission()` is used to check the required permission. In the method `getLogger()`, I use the SLF4J method `LoggerFactory.getLogger()` to create a new `org.slf4j.Logger` instance. The logger name is created from the module name and the provided name. The return value of `getLogger()` is an instance of the class `SLF4JLoggerWrapper`, which is shown in Listing 6-3.

Listing 6-2. `SLF4JLoggerFinder`

```
public class SLF4JLoggerFinder extends System.LoggerFinder {

  private static final RuntimePermission LOGGERFINDER_PERMISSION =
      new RuntimePermission("loggerFinder");

  public SLF4JLoggerFinder() {
    this.checkPermission();
  }

  private void checkPermission() {
    final SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      sm.checkPermission(LOGGERFINDER_PERMISSION);
    }
  }

  @Override
  public System.Logger getLogger(final String name, final Module module) {
    checkPermission();
    final Logger logger = LoggerFactory
        .getLogger(String.format("%s/%s", module.getName(), name));
    return new SLF4JLoggerWrapper(logger);
  }
}
```

The class `SLF4JLoggerWrapper` implements the interface `System.Logger`. In the method `isLoggable()`, it delegates to different methods in the wrapped `org.slf4j.Logger` instance the task of checking whether the logging level should be logged. For example, if the level is `DEBUG`, it delegates to the method

isDebugEnabled(). The logging methods in org.slf4j.Logger—for example, trace() and debug()—accept different parameters than the method log() in System.Logger, so I need to create adapters for localization and message format. The method localizedMessage() returns the localized message retrieved from the ResourceBundle. The method doLog() only takes plain messages and delegates to different logging methods in org.slf4j.Logger based on the level. In the method log() of SLF4JLoggerWrapper, the plain message is prepared first and passed to doLog() for logging.

Listing 6-3. SLF4JLoggerWrapper

```
public class SLF4JLoggerWrapper implements System.Logger {

  private final Logger logger;

  public SLF4JLoggerWrapper(final Logger logger) {
    this.logger = logger;
  }

  @Override
  public String getName() {
    return this.logger.getName();
  }

  @Override
  public boolean isLoggable(final Level level) {
    switch (level) {
      case ALL:
      case TRACE:
        return this.logger.isTraceEnabled();
      case DEBUG:
        return this.logger.isDebugEnabled();
      case INFO:
        return this.logger.isInfoEnabled();
      case WARNING:
        return this.logger.isWarnEnabled();
      case ERROR:
        return this.logger.isErrorEnabled();
      default:
        return false;
    }
  }

  @Override
  public void log(final Level level, final ResourceBundle bundle,
      final String msg, final Throwable thrown) {
    this.doLog(level, localizedMessage(bundle, msg), thrown);
  }

  @Override
  public void log(final Level level, final ResourceBundle bundle,
      final String format, final Object... params) {
```

```
    this.doLog(level,
        String.format(localizedMessage(bundle, format), params),
        null);
  }

  private void doLog(final Level level,
      final String msg,
      final Throwable thrown) {
    switch (level) {
      case ALL:
      case TRACE:
        this.logger.trace(msg, thrown);
        break;
      case DEBUG:
        this.logger.debug(msg, thrown);
        break;
      case INFO:
        this.logger.info(msg, thrown);
        break;
      case WARNING:
        this.logger.warn(msg, thrown);
        break;
      case ERROR:
        this.logger.error(msg, thrown);
        break;
    }
  }

  private String localizedMessage(final ResourceBundle resourceBundle,
      final String msg) {
    if (resourceBundle != null) {
      try {
        return resourceBundle.getString(msg);
      } catch (final MissingResourceException e) {
        return msg;
      }
    }
    return msg;
  }
}
```

The file `module-info.java` of this module in Listing 6-4 uses `provides` to declare the provider of `System.LoggerFinder`.

Listing 6-4. Module Declaration

```
module feature.logging {
    requires transitive slf4j.api;
    provides java.lang.System.LoggerFinder
        with io.vividcode.feature9.logging.SLF4JLoggerFinder;
}
```

Now I can use the logging service to log messages. I create a logger using the method `System.getLogger()`, then use the method `log()` to log messages. Here I use some Java standard classes to see how both application and platform logs are combined.

Listing 6-5. Using the Logging Service

```
public class Main {

  private static final System.Logger LOGGER = System.getLogger("Main");

  public static void main(final String[] args) {
    LOGGER.log(Level.INFO, "Run!");
    try {
      new URL("https://google.com").openConnection().connect();
    } catch (final IOException e) {
      LOGGER.log(Level.WARNING, "Failed to connect", e);
    }
  }
}
```

Listing 6-6 shows the logs. In the output, you can see both application logs and JDK logs. Here I configured Logback to log out all messages.

Listing 6-6. Log Messages

```
2017-09-20 21:23:56,202 |  INFO [      main] feature.runtime/Main : Run!
2017-09-20 21:23:56,448 | TRACE [      main] .h.HttpURLConnection : ProxySelector Request
for https://google.com/
2017-09-20 21:23:56,450 | TRACE [      main] .h.HttpURLConnection : Looking for HttpClient
for URL https://google.com and proxy value of DIRECT
2017-09-20 21:23:56,451 | TRACE [      main] .h.HttpURLConnection : Creating new HttpsClient
with url:https://google.com and proxy:DIRECT with connect timeout:-1
2017-09-20 21:23:56,516 | TRACE [      main] .h.HttpURLConnection : Proxy used: DIRECT
```

Summary

With the new platform logging API and service, you can now use a single logging implementation for both the JDK itself and applications. This makes the error diagnostics much easier. In next chapter, we'll discuss reactive streams.

CHAPTER 7

Reactive Streams

Reactive programming (https://en.wikipedia.org/wiki/Reactive_programming) has gained popularity in the Java community recently. In the Java world, we have popular reactive libraries like RxJava (https://github.com/ReactiveX/RxJava) and Reactor (http://projectreactor.io/). Reactive Streams (http://www.reactive-streams.org/) is an initiative to provide a standard for asynchronous stream processing with nonblocking back pressure. Core interfaces from the Reactive Streams specification have been added to Java 9 in the class `java.util.concurrent.Flow`.

Core Interfaces

This section covers the four core interfaces in `Flow`.

Flow.Publisher<T>

The interface `Flow.Publisher<T>` represents a producer of items to be received by subscribers. It only has one method, `void subscribe(Flow.Subscriber<? super T> subscriber)`, which adds subscribers of type `Flow.Subscriber`. There are three types of notifications that a publisher can publish; see Table 7-1.

Table 7-1. *Notifications of* `Publisher`

Notification	Description
`onNext`	A new item has been published.
`onComplete`	No further notifications will be published.
`onError`	An error has been encountered when publishing items.

Both `onComplete` and `onError` are *terminal notifications* meaning that no more notifications will be published after an `onComplete` or `onError` notification. It's valid for a publisher to not publish any `onNext` notifications. It's also possible for a publisher to publish an infinite number of `onNext` notifications.

Flow.Subscriber<T>

The interface `Flow.Subscriber<T>` represents a receiver of notifications published by a `Flow.Publisher`. It has four methods; see Table 7-2.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_7

Table 7-2. *Methods of* `Flow.Subscriber`

Method	Description
`void onNext(T item)`	Invoked when an item is received
`void onComplete()`	Invoked when an `onComplete` notification is received
`void onError(Throwable throwable)`	Invoked when an `onError` notification is received
`void onSubscribe(Flow.Subscription subscription)`	Invoked when the subscriber successfully subscribes to the publisher

If a subscriber cannot subscribe to a publisher, the method `onError()` is invoked with the error.

Flow.Subscription

The interface `Flow.Subscription` represents the subscription when a `Flow.Subscriber` successfully subscribes to a `Flow.Publisher`. It's passed as an argument to the subscriber's method `onSubscribe()`. Table 7-3 shows the methods of `Flow.Subscription`.

Table 7-3. *Methods of* `Flow.Subscription`

Method	Description
`void request(long n)`	Request `n` items to be published.
`void cancel()`	Cancel the subscription.

A subscriber only receives items after requesting them. The method `request(long n)` makes a demand of the publisher to indicate that the subscriber is ready to handle `n` items. This is important because it makes sure the subscriber is not overwhelmed by the publisher. The subscriber may receive fewer items than requested if the publisher terminates earlier. After the `cancel()` method is invoked, the subscriber may still receive additional notifications, but the subscription will be eventually cancelled.

Flow.Processor<T,R>

The interface `Flow.Processor` represents a component that acts as both a subscriber and a publisher. Listing 7-1 shows its definition. It simply extends from both `Subscriber<T>` and `Publisher<R>`.

Listing 7-1. Definition of `Flow.Processor`

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

SubmissionPublisher

Although the four core interfaces are easy to understand, it's not easy to correctly implement them. However, you don't actually need to care about the implementations; you can simply rely on third-party libraries.

The class `java.util.concurrent.SubmissionPublisher<T>` is an implementation of `Flow.Publisher` that asynchronously publishes submitted items to subscribers until it's closed. The items of `SubmissionPublisher` are not generated by the publisher but are submitted by the client code. `SubmissionPublisher` can function as a bridge client code can use to communicate with subscribers.

`SubmissionPublisher` uses an `Executor` to send items to subscribers. The `Executor` can be provided in the constructor, or `ForkJoinPool.commonPool()` is used by default. When any subscriber method throws an error, you can provide a handler to handle the exception before the subscription is cancelled. `SubmissionPublisher` has three constructors.

- The no-arg constructor uses the default `Executor` and no exception handler.
- `SubmissionPublisher(Executor executor, int maxBufferCapacity)` configures the `Executor` and the maximum size of the buffer. The buffer contains submitted but not consumed items for a subscriber. Each subscriber has its own independent buffer.
- `SubmissionPublisher(Executor executor, int maxBufferCapacity, BiConsumer<? super Flow.Subscriber<? super T>,? super Throwable> handler)` configures the `Executor`, the maximum size of the buffer, and the exception handler.

■ **Note** The actual value of the maximum buffer size may be rounded up to the nearest power of two or bounded by the largest value supported by the implementation. For example, if `10` is passed to the constructor as the value of `maxBufferCapacity`, the actual used value is `16`. We can use the method `getMaxBufferCapacity()` to get the actual value.

Since subscribers may have different speeds at which they consume published items, it's possible that some items have not yet been consumed by all subscribers. The method `estimateMaximumLag()` returns an estimate of the maximum number of items that have been published but have not yet been consumed by all subscribers. On the other hand, it's also possible that the speed at which items are published cannot keep up with the consumption speed of the subscribers. In this case, the method `estimateMinimumDemand()` returns an estimate of the minimum number of items that have been requested but not yet published.

To submit items to `SubmissionPublisher`, you can use the `submit()` and `offer()` methods. The method `submit(T item)` publishes the item to all subscribers asynchronously by invoking their `onNext()` methods. `submit()` blocks uninterruptibly while there is no resource available for any subscriber to handle. You should use `submit()` when you want to make sure the items are published to all subscribers successfully. However, its blocking behavior may have a performance penalty. An overwhelmed subscriber may slow down the processing of other subscribers. The return value of `submit()` is the estimate of the maximum lag and is the same as the return value of the method `estimateMaximumLag()`.

The methods `int offer(T item, BiPredicate<Flow.Subscriber<? super T>,? super T> onDrop)` and `int offer(T item, long timeout, TimeUnit unit, BiPredicate<Flow.Subscriber<? super T>,? super T> onDrop)` also publish the items to all subscribers, but they allow items to be dropped when resources are not available for one or more subscribers. When an item is dropped, the provided handler `onDrop` is invoked with the current subscriber and the dropped item. The type of the handler is `BiPredicate`, so it returns a Boolean value. If the return value of the handler is `true`, the publisher will try to publish the item again. The return value of `offer()` indicates the result of publishing. If the value is negative, its absolute value represents the number of drops; otherwise, it represents an estimate of the maximum lag with the same meaning as the value `submit()` returned. If a timeout is added when `offer()` is invoked, it blocks while no resource is available for any subscriber and continues to block up to the specified timeout or until the caller thread is interrupted. The `onDrop` handler is invoked in both cases.

Table 7-4 shows other important methods of *SubmissionPublisher*.

Table 7-4. *Other Methods of SubmissionPublisher*

Method	Description
`CompletableFuture<Void> consume(Consumer<? super T> consumer)`	Processes all published items using the given consumer. The return value is a `CompletableFuture<Void>` that is completed normally when the publisher signals `onComplete` or completed exceptionally upon any error.
`void close()`	Closes the publisher and signals `onComplete` to all subscribers.
`void closeExceptionally(Throwable error)`	Closes the publisher and signals `onError` to all subscribers with the provided error.
`boolean isClosed()`	Checks if the publisher is closed.
`List<Flow.Subscriber<? super T>> getSubscribers()`	Returns a list of all subscribers.
`int getNumberOfSubscribers()`	Returns the number of subscribers.
`boolean hasSubscribers()`	Checks if the publisher has any subscriber.
`boolean isSubscribed(Flow.Subscriber<? super T> subscriber)`	Checks if the given subscriber is subscribed.

Now you can use `SubmissionPublisher` to publish some data. In Listing 7-2, the class `PeriodicPublisher` periodically publishes the given number of items and closes when all items are published. The constructor parameter `Consumer<PeriodicPublisher<T>> action` is used to publish the items. The method `waitForCompletion()` can be used to wait for all items to be published.

Listing 7-2. `PeriodicPublisher`

```
public class PeriodicPublisher<T> extends SubmissionPublisher<T> {

  private final ScheduledExecutorService scheduler;
  private final ScheduledFuture<?> periodicTask;
  private final AtomicInteger count = new AtomicInteger(0);
  private final CountDownLatch closeLatch = new CountDownLatch(1);

  public PeriodicPublisher(final Consumer<PeriodicPublisher<T>> action,
      final int maxBufferCapacity,
      final int total,
      final long period,
      final TimeUnit timeUnit) {
    super(ForkJoinPool.commonPool(),
        maxBufferCapacity,
        ((subscriber, throwable) ->
            System.out.printf("Publish error for %s: %s",
                subscriber, throwable)));
    this.scheduler = new ScheduledThreadPoolExecutor(1);
    this.periodicTask = this.scheduler.scheduleAtFixedRate(
        () -> {
          action.accept(this);
          final int value = this.count.incrementAndGet();
```

```
          if (value >= total) {
            this.doClose();
          }
        }, 0, period, timeUnit);
  }

  @Override
  public void close() {
    this.periodicTask.cancel(false);
    this.scheduler.shutdown();
    super.close();
  }

  public void waitForCompletion() {
    try {
      this.closeLatch.await();
    } catch (final InterruptedException e) {
      this.close();
    }
  }

  private void doClose() {
    try {
      this.close();
    } finally {
      this.closeLatch.countDown();
    }
  }
}
```

You can use `PeriodicPublisher` to create a random number generator; see Listing 7-3. The action passed in the constructor is `publisher.submit(ThreadLocalRandom.current().nextLong()`, which uses the method `submit()` to publish a random number.

Listing 7-3. Random Number Generator

```
public class RandomNumberGenerator extends PeriodicPublisher<Long> {

  public RandomNumberGenerator() {
    super((publisher) ->
            publisher.submit(ThreadLocalRandom.current().nextLong()),
        Flow.defaultBufferSize(),
        10,
        1,
        TimeUnit.SECONDS);
  }

  public static void main(final String[] args) {
    final RandomNumberGenerator generator = new RandomNumberGenerator();
    generator.subscribe(new Flow.Subscriber<>() {
```

```
      @Override
      public void onSubscribe(final Flow.Subscription subscription) {
        subscription.request(Long.MAX_VALUE);
      }

      @Override
      public void onNext(final Long item) {
        System.out.printf("Received: %s%n", item);
      }

      @Override
      public void onError(final Throwable throwable) {
        throwable.printStackTrace();
      }

      @Override
      public void onComplete() {
        System.out.println("Completed!");
      }
    });
    generator.waitForCompletion();
  }
}
```

Now I am going to show you the difference between `submit()` and `offer()`. In Listing 7-4, I use three different methods to publish items. The `DelayedSubscriber` class delays `100ms` before it processes each item. The publisher uses a buffer with a maximum size of 16. A total number of 50 sequence numbers are published with an interval of `50ms`. I use `waitForCompletion()` to wait for publishing to complete, then wait another 5 seconds to allow buffered items to be processed. When I publish items using `submit()`, all items can be processed. When I publish items using `offer()` without a timeout, some of the items are dropped. This is because the buffer size is only 16 and it takes `100ms` to process a single item. During that time, two items are generated. The buffer fills up before all items can be processed, and some of these items are dropped. The drop handler of `offer()` returns `true`, so the publisher tries to publish again. When you run the code, due to its concurrent nature, the dropped items may be different. When I publish items using `offer()` with a timeout of one second, because the timeout is much longer than the processing time, all items can be processed.

Listing 7-4. Differences Between `submit()` and `offer()`

```
public class DelayedSubscribers {

  public static void main(final String[] args) {
    final DelayedSubscribers delayedSubscribers = new DelayedSubscribers();
    delayedSubscribers.publishWithSubmit();
    System.out.println("===========");
    delayedSubscribers.publishWithOffer();
    System.out.println("===========");
    delayedSubscribers.publishWithOfferTimeout();
  }
```

```
public void publishWithSubmit() {
  final SequenceGenerator sequenceGenerator = new SequenceGenerator();
  this.publish(publisher -> publisher.submit(sequenceGenerator.get()));
}

public void publishWithOffer() {
  final SequenceGenerator sequenceGenerator = new SequenceGenerator();
  this.publish(publisher -> publisher.offer(sequenceGenerator.get(),
      ((subscriber, value) -> {
        System.out.printf("%s dropped %s%n", subscriber, value);
        return true;
      })));
}

public void publishWithOfferTimeout() {
  final SequenceGenerator sequenceGenerator = new SequenceGenerator();
  this.publish(publisher ->
      publisher.offer(
          sequenceGenerator.get(),
          1000,
          TimeUnit.MILLISECONDS,
          ((subscriber, value) -> {
            System.out.printf("%s dropped %s%n", subscriber, value);
            return true;
          })
      ));
}

private void publish(final Consumer<PeriodicPublisher<Integer>> action) {
  final PeriodicPublisher<Integer> publisher =
      new PeriodicPublisher<>(
          action,
          16,
          50,
          50,
          TimeUnit.MILLISECONDS);
  publisher.subscribe(new DelayedSubscriber<>("1"));
  publisher.subscribe(new DelayedSubscriber<>("2"));
  publisher.subscribe(new DelayedSubscriber<>("3"));
  publisher.waitForCompletion();
  System.out.println("Publish completed");
  try {
    Thread.sleep(5000);
  } catch (final InterruptedException e) {
    e.printStackTrace();
  }
}
```

```
  public static class SequenceGenerator implements Supplier<Integer> {

    private int count = 1;

    @Override
    public Integer get() {
      return this.count++;
    }
  }

  public static class DelayedSubscriber<T> implements Flow.Subscriber<T> {

    private final String id;
    private Flow.Subscription subscription;

    public DelayedSubscriber(final String id) {
      this.id = id;
    }

    @Override
    public void onSubscribe(final Flow.Subscription subscription) {
      this.subscription = subscription;
      System.out.printf("%s subscribed!%n", this.id);
      subscription.request(1);
    }

    @Override
    public void onNext(final T item) {
      this.subscription.request(1);
      try {
        Thread.sleep(100);
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
      System.out.printf("%s processed: %s%n", this.id, item);
    }

    @Override
    public void onError(final Throwable throwable) {
      throwable.printStackTrace();
    }

    @Override
    public void onComplete() {
      System.out.printf("%s completed!%n", this.id);
    }
```

```
    @Override
    public String toString() {
      return String.format("Subscriber %s", this.id);
    }
  }
}
```

Third-Party Libraries

If you have used RxJava or Reactor before, you'll find out that by comparison, the interfaces provided by `Flow` are quite minimal. The Reactive Streams specification is designed to be minimal and it focuses on the interoperability between different libraries. You can integrate `Flow` with RxJava and Reactor.

RxJava 2

You can use the library RxJava2Jdk9Interop (https://github.com/akarnokd/RxJava2Jdk9Interop) for the conversion. Listing 7-5 shows you how to do the conversion between RxJava 2 `Flowable` and `Flow`.

Listing 7-5. RxJava 2 and `Flow`

```
import hu.akarnokd.rxjava2.interop.FlowInterop;
import io.reactivex.Flowable;
import java.util.concurrent.TimeUnit;

public class RxJava2 {

  public void toFlow() {
    Flowable.interval(0, 50, TimeUnit.MILLISECONDS)
        .take(50)
        .to(FlowInterop.toFlow())
        .subscribe(new DelayedSubscribers.DelayedSubscriber<>("1"));
  }

  public void fromFlow() {
    final DelayedSubscribers.SequenceGenerator sequenceGenerator =
        new DelayedSubscribers.SequenceGenerator();
    final PeriodicPublisher<Integer> publisher =
        new PeriodicPublisher<>(
            pub -> pub.submit(sequenceGenerator.get()),
            16,
            50,
            50,
            TimeUnit.MILLISECONDS);
    FlowInterop.fromFlowPublisher(publisher)
        .map(v -> v * 10)
        .forEach(System.out::println);
  }
```

```
  public static void main(final String[] args) {
    final RxJava2 rxJava2 = new RxJava2();
    rxJava2.toFlow();
    rxJava2.fromFlow();
    try {
      Thread.sleep(10000);
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Reactor

Reactor has a built-in adapter for `Flow`. Listing 7-6 shows how to do the conversion between Reactor `Flux` and `Flow`.

Listing 7-6. Reactor and `Flow`

```
import java.time.Duration;
import java.util.concurrent.TimeUnit;
import reactor.adapter.JdkFlowAdapter;
import reactor.core.publisher.Flux;

public class Reactor {

  public void toFlow() {
    JdkFlowAdapter.publisherToFlowPublisher(
        Flux.interval(Duration.ZERO, Duration.ofMillis(50))
            .take(50)
    ).subscribe(new DelayedSubscribers.DelayedSubscriber<>("1"));
  }

  public void fromFlow() {
    final DelayedSubscribers.SequenceGenerator sequenceGenerator =
        new DelayedSubscribers.SequenceGenerator();
    final PeriodicPublisher<Integer> publisher =
        new PeriodicPublisher<>(
            pub -> pub.submit(sequenceGenerator.get()),
            16,
            50,
            50,
            TimeUnit.MILLISECONDS);
    JdkFlowAdapter.flowPublisherToFlux(publisher)
        .map(v -> v * 10)
        .subscribe(System.out::println);
  }

  public static void main(final String[] args) {
    final Reactor reactor = new Reactor();
    reactor.toFlow();
```

```
    reactor.fromFlow();
    try {
      Thread.sleep(10000);
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Interoperability

Listing 7-7 shows you how to use `Flow` to convert an RxJava 2 `Flowable` to a Reactor `Flux`.

Listing 7-7. Interoperability between RxJava 2 and Reactor

```
import hu.akarnokd.rxjava2.interop.FlowInterop;
import io.reactivex.Flowable;
import java.util.concurrent.Flow;
import java.util.concurrent.TimeUnit;
import reactor.adapter.JdkFlowAdapter;

public class RxInterop {

  public static void main(final String[] args) {
    final Flow.Publisher<Long> publisher =
        Flowable.interval(0, 50, TimeUnit.MILLISECONDS)
            .take(50)
            .to(FlowInterop.toFlow());
    JdkFlowAdapter.flowPublisherToFlux(publisher)
        .map(v -> v * 10)
        .toStream()
        .forEach(System.out::println);
  }
}
```

Summary

The introduction of the Reactive Streams specification to Java 9 is a big step toward embracing reactive programming principles in the Java platform. In this chapter, we discussed core interfaces in `Flow` and the built-in `SubmissionPublisher`. I also demonstrated the interoperability with the popular reactive libraries RxJava 2 and Reactor. In the next chapter, we'll discuss variable handles.

CHAPTER 8

Variable Handles

A *variable handle* is a reference to a variable, or to a family of variables, including static fields, nonstatic fields, array elements, or components of an off-heap data structure. The concept of variable handles is similar to method handles. Variable handles are represented using the class `java.lang.invoke.VarHandle`. With variable handles, you can perform different operations on variables called *access modes*. All supported access modes are defined in the enum `VarHandle.AccessMode`.

Creating Variable Handles

`VarHandle` objects can be created using factory methods in the `java.lang.invoke.MethodHandles.Lookup` class. First you can get an instance of `MethodHandles.Lookup` using the method `MethodHandles.lookup()`, then you can use its methods to create `VarHandle` objects.

findStaticVarHandle

The method `VarHandle findStaticVarHandle(Class<?> decl, String name, Class<?> type)` returns a `VarHandle` that accesses a static field `name` of the given type declared in a type `decl`. In Listing 8-1, I create a `VarHandle` that accesses the static `int` field `staticVar` declared in the class `HandleTarget`.

Listing 8-1. Example of `findStaticVarHandle()`

```
MethodHandles.lookup()
  .findStaticVarHandle(HandleTarget.class, "staticVar", int.class);
```

findVarHandle

The method `VarHandle findVarHandle(Class<?> recv, String name, Class<?> type)` returns a `VarHandle` that accesses a nonstatic field `name` of the given type in a type `recv`. In Listing 8-2, I create a `VarHandle` that accesses the nonstatic `int` field `count` declared in the class `HandleTarget`.

Listing 8-2. Example of `findVarHandle()`

```
MethodHandles.lookup()
  .findVarHandle(HandleTarget.class, "count", int.class);
```


F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_8

unreflectVarHandle

The method `VarHandle unreflectVarHandle(Field f)` creates a `VarHandle` from a `java.lang.reflect.Field` object. In Listing 8-3, I create a `VarHandle` using the `Field` object created from the method `getDeclaredField()` of `HandleTarget.class`.

Listing 8-3. Example of `unreflectVarHandle()`

```
MethodHandles
  .lookup()
  .unreflectVarHandle(HandleTarget.class.getDeclaredField("count"));
```

Access Modes

To understand different variable access modes, you first need to understand memory ordering.

Memory Ordering

The access modes of variable handles are compatible with C/C++11 atomics (http://en.cppreference.com/w/cpp/atomic). Table 8-1 shows the different memory orderings supported by access modes.

Table 8-1. *Memory Orderings*

Memory Ordering	Access	Description
plain	read/write	The same memory semantics as nonvolatile, which has no special memory ordering effects with respect to other threads. Only atomic for references and for primitive values of, at most, 32 bits.
volatile	read/write	The same memory semantics as Java `volatile`.
opaque	read/write	No assurance of memory ordering effects with respect to other threads. Only atomic with respect to accesses to the same variable.
acquire	read	Ensures that subsequent loads and stores are not reordered before this access; compatible with C/C++11 `memory_order_acquire` ordering.
release	write	Ensures that prior loads and stores are not reordered after this access; compatible with C/C++11 `memory_order_release` ordering.

■ **Note** For more details about C/C++11 `acquire` and `release` memory ordering, see http://en.cppreference.com/w/cpp/atomic/memory_order#Release-Acquire_ordering.

These memory orderings are used as the suffix of access modes to specify the memory ordering effects of these modes. For example, `GET_AND_ADD_ACQUIRE` has the memory ordering `acquire` for the read access.

VarHandle Methods

Thirty-one access modes are defined in the enum `VarHandle.AccessMode`. For each access mode, there is a corresponding method in `VarHandle` that you need to use with this access mode. For example, for the access mode `GET_AND_ADD`, the method in `VarHandle` is `getAndAdd()`.

Signature Polymorphic

The methods for access modes in `VarHandle` are *signature polymorphic*. Even though all these methods have the same signature of `Object methodName(Object... args)`, a runtime check is done to make sure the runtime types of arguments match the requirements of the access modes. If the argument matching fails, JVM throws a `java.lang.invoke.WrongMethodTypeException`. Because the compiler doesn't check the arity and types of arguments statically, it's the developers' responsibility to make sure that the correct number of arguments with the correct types are passed when invoking those methods.

The list of arguments consists of between zero and many objects of coordinate types and optional arguments that are required by different access modes:

- These coordinate types are used to locate the variable you want to access. For example, when accessing an element in the array, the coordinate types are the type of the array and the type of the array index. You can use the method `List<Class<?>> coordinateTypes()` of `VarHandle` to get the list of coordinate types.
- Different access modes may require extra arguments. For example, the method `compareAndSet()` requires the expected value to compare and the new value to set.

These access modes can be grouped into five categories: read access, write access, atomic update, numeric atomic update, and bitwise atomic update. The class `HandleTarget` in Listing 8-4 contains different static variables for testing. I'll use these variables in following sections.

Listing 8-4. `HandleTarget` for Testing

```
public class HandleTarget {
    public int count = 1;

    public String[] names = new String[]{"Alex", "Bob", "David"};

    public byte[] data = new byte[]{1, 0, 0, 0, 1, 0, 0, 0};

    public ByteBuffer dataBuffer = ByteBuffer.wrap(this.data);
}
```

Now let's discuss all five categories of methods.

Read Access

This category includes the `get()`, `getVolatile()`, `getOpaque()`, and `getAcquire()` methods with memory ordering effects specified by the suffix. The method `get()` has the plain memory ordering effect (see Table 8-1). Listing 8-5 shows the usage of different read access methods. The variable handle `varHandle` references the variable `count` in the class `HandleTarget` of Listing 8-4. Because the test case is not multithreaded, all these read access modes return the same result.

Listing 8-5. Read Access

```
public class VarHandleTest {

  private HandleTarget handleTarget = new HandleTarget();
  private VarHandle varHandle;

  @Before
  public void setUp() throws Exception {
    this.handleTarget = new HandleTarget();
    this.varHandle = MethodHandles
        .lookup()
        .findVarHandle(HandleTarget.class, "count", int.class);
  }

  @Test
  public void testGet() throws Exception {
    assertEquals(1, this.varHandle.get(this.handleTarget));
    assertEquals(1, this.varHandle.getVolatile(this.handleTarget));
    assertEquals(1, this.varHandle.getOpaque(this.handleTarget));
    assertEquals(1, this.varHandle.getAcquire(this.handleTarget));
  }
}
```

All these read access methods only require one parameter, which is the target object that contains this variable. The class `HandleTarget` is the only coordinate type.

Write Access

This category includes the `set()`, `setVolatile()`, `setOpaque()`, and `setRelease()` methods with memory ordering effects specified by the suffix. The method `set()` has the plain memory ordering effect (see Table 8-1). Listing 8-6 shows the usage of different write access methods.

Listing 8-6. Write Access

```
@Test
public void testSet() throws Exception {
  final int newValue = 2;
  this.varHandle.set(this.handleTarget, newValue);
  assertEquals(newValue, this.varHandle.get(this.handleTarget));
  this.varHandle.setVolatile(this.handleTarget, newValue + 1);
  assertEquals(newValue + 1, this.varHandle.get(this.handleTarget));
  this.varHandle.setOpaque(this.handleTarget, newValue + 2);
  assertEquals(newValue + 2, this.varHandle.get(this.handleTarget));
  this.varHandle.setRelease(this.handleTarget, newValue + 3);
  assertEquals(newValue + 3, this.varHandle.get(this.handleTarget));
}
```

All these write access methods takes two parameters. The first parameter is the target object, the second is the new value to set.

Atomic Update

This category includes methods from four subcategories with memory ordering effects specified by the suffix. All the methods perform both read and write access to a variable, which may have different memory ordering effects for read and write access.

The method `compareAndSet()` atomically sets the value of a variable if the variable's current value equals the expected value with the volatile memory ordering for both read and write access. The returned `boolean` value indicates if the operation successfully updates the value.

The `weakCompareAndSet()`, `weakCompareAndSetPlain()`, `weakCompareAndSetAcquire()`, and `weakCompareAndSetRelease()` methods could atomically set the value of a variable if the variable's current value equals the expected value. The returned Boolean value indicates if the operation successfully updated the value. This operation may fail even if the variable's current value does match the expected value. This is why they are called *weak* operations.

The `compareAndExchange()`, `compareAndExchangeAcquire()`, and `compareAndExchangeRelease()` methods atomically set the value of a variable if the variable's current value is equal to the expected value. The return value is the current value, which will be the same as the expected value if the operation is successful.

The `getAndSet()`, `getAndSetAcquire()`, and `getAndSetRelease()` methods atomically set the value of a variable to the new value and return the variable's previous value.

Table 8-2 shows each method's memory ordering for read and write access.

Table 8-2. *Memory Ordering for Read and Write Access of Atomic Update Methods*

Method	Read	Write
`compareAndSet()`	volatile	volatile
`weakCompareAndSet()`	volatile	volatile
`weakCompareAndSetPlain()`	plain	plain
`weakCompareAndSetAcquire()`	acquire	plain
`weakCompareAndSetRelease()`	plain	release
`compareAndExchange()`	volatile	volatile
`compareAndExchangeAcquire()`	acquire	plain
`compareAndExchangeRelease()`	plain	release
`getAndSet()`	volatile	volatile
`getAndSetAcquire()`	acquire	plain
`getAndSetRelease()`	plain	release

Listing 8-7 shows how to use the different atomic update methods.

Listing 8-7. Atomic Update Methods

```
@Test
public void testAtomicUpdate() throws Exception {
  final int expectedValue = 1;
  final int newValue = 2;
  assertEquals(true,
```

```
        this.varHandle.compareAndSet(this.handleTarget, expectedValue, newValue));
    assertEquals(newValue,
        this.varHandle.compareAndExchange(this.handleTarget, newValue, newValue + 1));
    assertEquals(newValue + 1, this.varHandle.getAndSet(this.handleTarget, newValue + 2));
}
```

Numeric Atomic Update

The `getAndAdd()`, `getAndAddAcquire()`, and `getAndAddRelease()` methods atomically add the value to the current value of a variable and return the variable's previous value. Table 8-3 shows the memory ordering for read and write access of these methods.

Table 8-3. *Memory Ordering for Read and Write Access of Numeric Atomic Update Methods*

Method	Read	Write
`getAndAdd()`	volatile	volatile
`getAndAddAcquire()`	acquire	plain
`getAndAddRelease()`	plain	release

Listing 8-8 shows how to use different numeric atomic update methods.

Listing 8-8. Numeric Atomic Update Methods

```
@Test
public void testNumericAtomicUpdate() throws Exception {
    final int expectedValue = 1;
    assertEquals(expectedValue,
        this.varHandle.getAndAdd(this.handleTarget, 1));
    assertEquals(expectedValue + 1,
        this.varHandle.getAndAddAcquire(this.handleTarget, 1));
    assertEquals(expectedValue + 2,
        this.varHandle.getAndAddRelease(this.handleTarget, 1));
}
```

Bitwise Atomic Update

The `getAndBitwiseAnd()`, `getAndBitwiseAndAcquire()`, `getAndBitwiseAndRelease()`, `getAndBitwiseOr()`, `getAndBitwiseOrAcquire()`, `getAndBitwiseOrRelease()`, `getAndBitwiseXor()`, `getAndBitwiseXorAcquire()`, and `getAndBitwiseXorRelease()` methods atomically set the value of a variable to the result of a bitwise AND/OR/XOR between the variable's current value and the mask value and return the variable's previous value. Table 8-4 shows the memory ordering for read and write access of these methods.

Table 8-4. *Memory Ordering for Read and Write Access of Bitwise Atomic Update Methods*

Method	Read	Write
`getAndBitwiseAnd()`	volatile	volatile
`getAndBitwiseOr()`	volatile	volatile
`getAndBitwiseXor()`	volatile	volatile
`getAndBitwiseAndAcquire()`	acquire	plain
`getAndBitwiseOrAcquire()`	acquire	plain
`getAndBitwiseXorAcquire()`	acquire	plain
`getAndBitwiseAndRelease()`	plain	release
`getAndBitwiseOrRelease()`	plain	release
`getAndBitwiseXorRelease()`	plain	release

Listing 8-9 shows how to use different bitwise numeric atomic update methods.

Listing 8-9. Bitwise Atomic Update Methods

```
@Test
public void testBitwiseAtomicUpdate() throws Exception {
  final int mask = 1;
  assertEquals(1, this.varHandle.getAndBitwiseAnd(this.handleTarget, mask));
  assertEquals(1, this.varHandle.get(this.handleTarget));
  assertEquals(1, this.varHandle.getAndBitwiseOr(this.handleTarget, mask));
  assertEquals(1, this.varHandle.get(this.handleTarget));
  assertEquals(1, this.varHandle.getAndBitwiseXor(this.handleTarget, mask));
  assertEquals(0, this.varHandle.get(this.handleTarget));
}
```

Arrays

`VarHandles` can also be used to access individual elements in an array. You can create a `VarHandle` that accesses array elements using the method `MethodHandles.arrayElementVarHandle(Class<?> arrayClass)`.

In Listing 8-10, I create a `VarHandle` that accesses `String[]` arrays. The method `compareAndSet()` updates the first element in the array to the new value. You need to provide the array, index, expected value, and new value when invoking this method.

Listing 8-10. `VarHandle` for Accessing Array Elements

```
@Test
public void testArray() throws Exception {
  final VarHandle arrayElementHandle = MethodHandles
      .arrayElementVarHandle(String[].class);
  assertEquals(true,
      arrayElementHandle.compareAndSet(
          this.handleTarget.names, 0, "Alex", "Alex_new"));
  assertEquals("Alex_new", this.handleTarget.names[0]);
}
```

byte[] and ByteBuffer Views

`VarHandle` allows you to view a byte array or `ByteBuffer` as an array of different primitive types, for example, `int[]` or `long[]`. You can use the `MethodHandles.byteArrayViewVarHandle()` or `MethodHandles.byteBufferViewVarHandle()` methods to create views of `byte[]` or `ByteBuffer`, respectively.

In order for you to be able to use the `byteArrayViewVarHandle()` method to create a `VarHandle`, the first parameter must be the array class you use to view the byte array. The element type can be `short`, `char`, `int`, `long`, `float`, and `double`. The second parameter is the byte order with type `java.nio.ByteOrder`, and it can be `BIG_ENDIAN` or `LITTLE_ENDIAN`. In Listing 8-11, I create a `VarHandle` by viewing the byte array as `int[]` with the byte order set to `BIG_ENDIAN`. Then I use the method `get()` to get the `int` value starting at the specified index in the original byte array. The maximum value of the index is the size of the byte array minus the byte size of the element type. For the `VarHandle` in Listing 8-11, the byte size of `int` is `4`, so the maximum value of the index is `8 - 4 = 4`. Using an index value greater than `4` will result in an `IndexOutOfBoundsException`.

Listing 8-11. Byte Array View

```
@Test
public void testByteArrayView() throws Exception {
  final VarHandle varHandle = MethodHandles
      .byteArrayViewVarHandle(int[].class, ByteOrder.BIG_ENDIAN);
  final byte[] data = this.handleTarget.data;
  assertEquals(16777216, varHandle.get(data, 0));
  assertEquals(1, varHandle.get(data, 1));
  assertEquals(256, varHandle.get(data, 2));
  assertEquals(65536, varHandle.get(data, 3));
  assertEquals(16777216, varHandle.get(data, 4));
}
```

Listing 8-12 uses the method `byteBufferViewVarHandle()` to create a `VarHandle`. The code has the same result as Listing 8-11.

Listing 8-12. `ByteBuffer` View

```
@Test
public void testByteBufferView() throws Exception {
  final VarHandle varHandle = MethodHandles
      .byteBufferViewVarHandle(int[].class, ByteOrder.BIG_ENDIAN);
  final ByteBuffer dataBuffer = this.handleTarget.dataBuffer;
  assertEquals(16777216, varHandle.get(dataBuffer, 0));
  assertEquals(1, varHandle.get(dataBuffer, 1));
  assertEquals(256, varHandle.get(dataBuffer, 2));
  assertEquals(65536, varHandle.get(dataBuffer, 3));
  assertEquals(16777216, varHandle.get(dataBuffer, 4));
}
```

Memory Fence

`VarHandle` also adds support for memory fencing that controls memory ordering in align with the C/C++11 atomic_thread_fence (`http://en.cppreference.com/w/cpp/atomic/atomic_thread_fence`). A fence ensures that loads and/or stores before the fence will not be reordered with loads and/or stores after the fence. There are five different static methods in `VarHandle` to create different types of fences that control what operations are not reordered; see Table 8-5.

Table 8-5. *Different Memory Fence Methods*

Method	Operations Before Fence	Operations After Fence
`void fullFence()`	loads and stores	loads and stores
`void acquireFence()`	loads	loads and stores
`void releaseFence()`	loads and stores	stores
`void loadLoadFence()`	loads	loads
`void storeStoreFence()`	stores	stores

Summary

In this chapter, we discussed how to use variable handles to access and manipulate variables. The methods in `VarHandle` can perform reads, writes, and atomic updates to variables. In the next chapter, we'll discuss the enhancements to method handles in Java 9.

CHAPTER 9

Enhanced Method Handles

The class `java.lang.invoke.MethodHandles` is enhanced in Java 9 to include more static methods for creating different kinds of method handles.

arrayConstructor

The method `MethodHandle arrayConstructor(Class<?> arrayClass)` returns a method handle to construct arrays of the specified type `arrayClass`. The sole argument of the created method handle is an `int` value that specifies the size of the array, while the return value is the array. In Listing 9-1, I create a method handle to construct `int[]` arrays. Then I create an array of size 3 using the `MethodHandle`.

Listing 9-1. Example of `arrayConstructor()`

```
@Test
public void testArrayConstructor() throws Throwable {
  final MethodHandle handle = MethodHandles.arrayConstructor(int[].class);
  final int[] array = (int[]) handle.invoke(3);
  assertEquals(3, array.length);
}
```

arrayLength

The method `MethodHandle arrayLength(Class<?> arrayClass)` returns a method handle to get the length of an array of type `arrayClass`. The sole argument of this method handle is the array to check, while the return value is an `int` value. In Listing 9-2, I create a method handle to get the length of `int[]` arrays. Then I invoke it to get the length of the input array.

Listing 9-2. Example of `arrayLength()`

```
@Test
public void testArrayLength() throws Throwable {
  final MethodHandle handle = MethodHandles.arrayLength(int[].class);
  final int[] array = new int[]{1, 2, 3, 4, 5};
  final int length = (int) handle.invoke(array);
  assertEquals(5, length);
}
```


F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_9

varHandleInvoker and varHandleExactInvoker

The `MethodHandle varHandleInvoker(VarHandle.AccessMode accessMode, MethodType type)` and `MethodHandle varHandleExactInvoker(VarHandle.AccessMode accessMode, MethodType type)` methods return a method handle to invoke a signature polymorphic access mode method on any `VarHandle` that has an access mode type compatible with the given type. Both methods have two parameters of following types:

- `VarHandle.AccessMode accessMode`: The access mode of `VarHandle`
- `MethodType type`: The type of the access mode method

The difference between `varHandleInvoker()` and `varHandleExactInvoker()` is that `varHandleInvoker()` does necessary type conversation, but `varHandleExactInvoker()` doesn't.

In Listing 9-3, the variable `varHandle` references the variable `count` in the class `HandleTarget`. I create the `MethodHandle` with the access mode `VarHandle.AccessMode.GET` and invoke it to get the value of the variable. The method `accessModeType()` of `VarHandle` is handy when you're creating the `MethodType` for a given access mode.

Listing 9-3. Example of `varHandleInvoker()`

```
@Test
public void testVarHandleInvoker() throws Throwable {
  final VarHandle varHandle = MethodHandles
      .lookup()
      .findVarHandle(HandleTarget.class, "count", int.class);
  final VarHandle.AccessMode accessMode = VarHandle.AccessMode.GET;
  final MethodHandle methodHandle = MethodHandles.varHandleInvoker(
      accessMode,
      varHandle.accessModeType(accessMode)
  );
  final HandleTarget handleTarget = new HandleTarget();
  final int result = (int) methodHandle.invoke(varHandle, handleTarget);
  assertEquals(result, 1);
}
```

zero

The method `MethodHandle zero(Class<?> type)` returns a method handle that always returns the default value for the given type. Listing 9-4 shows how to use `zero()` for different types.

Listing 9-4. Example of `zero()`

```
@Test
public void testZero() throws Throwable {
  assertEquals(0, MethodHandles.zero(int.class).invoke());
  assertEquals(0L, MethodHandles.zero(long.class).invoke());
  assertEquals(0F, MethodHandles.zero(float.class).invoke());
  assertEquals(0D, MethodHandles.zero(double.class).invoke());
  assertEquals(null, MethodHandles.zero(String.class).invoke());
}
```

empty

The method `MethodHandle empty(MethodType type)` returns a method handle that always return the default value based on the return type of the given `MethodType`. The returned method handle of `zero(type)` is actually equivalent to `empty(MethodType.methodType(type))`. Listing 9-5 shows how to use `empty()` for `int` and `String` types.

Listing 9-5. Example of `empty()`

```
@Test
public void testEmpty() throws Throwable {
  assertEquals(0, MethodHandles.
      empty(MethodType.methodType(int.class)).invoke());
  assertEquals(null, MethodHandles.
      empty(MethodType.methodType(String.class)).invoke());
}
```

Loops

`MethodHandles` has new static methods to create method handles representing different kinds of loops, including `for` loops, `while` loops, and `do-while` loops.

loop

The method `MethodHandle loop(MethodHandle[]... clauses)` represents `for` loops. It's quite complicated to create `for` loops using this method. A loop is defined by one or many clauses. A clause can specify up to four actions using method handles.

- `init`: Initializes an iteration variable before the loop executes.
- `step`: Updates the iteration variable when a clause executes. The return value of this action is used as the updated value of the iteration variable.
- `pred`: Executes a predicate to test for loop exit when a clause executes.
- `fini`: Computes the loop's return value if the loop exits.

Each clause can define its own iteration variable. Each iteration of the loop executes each clause in order. The actions `step`, `pred`, and `fini` receive all iteration variables as leading parameters. All these actions can also receive extra loop parameters. These loop parameters are passed as arguments when invoking the created `MethodHandle` and are available for all actions in all iterations. For the actions `step`, `pred`, and `fini`, loop parameters come after the iteration variables. The `init` action can only accept loop parameters.

In Listing 9-6, `init()`, `step()`, `pred()`, and `fini()` are methods used as actions in a clause. I only have one clause with `MethodHandles` for these methods when I'm using the method `loop()`. In the `step` action, the variable `sum` is updated to add the value of iteration variable `i`. The return value of `step` is used as the updated value of the iteration variable `i`. In the `pred` action, the variable `k` is the loop parameter with a value of `11`, which is provided when the created method handle `loop` is invoked. I check the values of `i` and `k` for loop exit. In the `fini` action, the value of `sum` is returned as the result of invoking the method handle `loop`. Because each clause is represented as an array of `MethodHandles`, I use the helper method `getMethodHandle()` to get the `MethodHandles` of those static methods in the current class.

Listing 9-6. Example of `loop()`

```
public class ForLoopTest {

  static int sum = 0;

  static int init() {
    return 1;
  }

  static int step(final int i) {
    sum += i;
    return i + 1;
  }

  static boolean pred(final int i, final int k) {
    return i < k;
  }

  static int fini(final int i, final int k) {
    return sum;
  }

  @Test
  public void testLoop() throws Throwable {
    final MethodHandle init = getMethodHandle("init",
        MethodType.methodType(int.class));
    final MethodHandle step = getMethodHandle("step",
        MethodType.methodType(int.class, int.class));
    final MethodHandle pred = getMethodHandle("pred",
        MethodType.methodType(boolean.class, int.class, int.class));
    final MethodHandle fini = getMethodHandle("fini",
        MethodType.methodType(int.class, int.class, int.class));
    final MethodHandle[] sumClause =
        new MethodHandle[]{init, step, pred, fini};
    final MethodHandle loop = MethodHandles.loop(sumClause);
    assertEquals(55, loop.invoke(11));
  }

  private MethodHandle getMethodHandle(final String name,
      final MethodType methodType)
      throws NoSuchMethodException, IllegalAccessException {
    return MethodHandles
        .lookup()
        .findStatic(ForLoopTest.class, name, methodType);
  }
}
```

In Listing 9-6, I use a static variable `sum` to maintain the internal state. I cannot use loop parameters here because they are not supposed to be updated. What I can do is to use another iteration variable from a different clause. That way, I can use the iteration variable in one clause to control the loop exit and use the other iteration variable to store the internal state.

countedLoop

The method `MethodHandle countedLoop(MethodHandle iterations, MethodHandle init, MethodHandle body)` creates a method handle representing a loop that runs for a given number of iterations. It's much easier to use than `loop()` for simple loops like `for (int i = 0; i < end; ++i)`. The parameters of `countedLoop()` are different `MethodHandles`.

- `iterations`: Determines the number of iterations. The result type of this method handle must be `int`.
- `init`: Initializes the optional loop variable. This variable can be updated in each iteration.
- `body`: Executes the method body.

The method handle created by `countedLoop()` has an implicit loop variable that starts at `0` and increases by `1` in each iteration. The method handle `init` can create a second loop variable that's passed to `body` as the first parameter and uses the return value of `body` as the updated value. The implicit loop variable is passed as the second parameter to `body`. Method handles `iterations`, `init`, and `body` all accept extra loop parameters that come after the second loop variable in the parameters list. The return value of invoking the method handle created by `countedLoop()` is the final value of the second loop variable.

The method `MethodHandle countedLoop(MethodHandle start, MethodHandle end, MethodHandle init, MethodHandle body)` is similar to `countedLoop(MethodHandle iterations, MethodHandle init, MethodHandle body)`, but it allows customization of the start and end values of the implicit loop variable. The method handles `start` and `end` must return values of type `int`, which determine the start (inclusive) and end (exclusive) values of the loop variable, respectively.

Listing 9-7 shows the usage of two `countedLoop()` methods to represent the same loop. In the method `body`, the first parameter `sum` is the loop variable I created to keep the summarized value. The second parameter `i` is the implicit loop variable.

Listing 9-7. Example of `countedLoop()`

```
public class CountedLoopTest {

  static int body(final int sum, final int i) {
    return sum + i + 1;
  }

  @Test
  public void testCountedLoop() throws Throwable {
    final MethodHandle iterations = MethodHandles.constant(int.class, 10);
    final MethodHandle init = MethodHandles.zero(int.class);
    final MethodHandle body = MethodHandles
        .lookup()
        .findStatic(CountedLoopTest.class, "body",
            MethodType.methodType(int.class, int.class, int.class));
    final MethodHandle countedLoop = MethodHandles
        .countedLoop(iterations, init, body);
    assertEquals(55, countedLoop.invoke());
  }

  @Test
  public void testCountedLoopStartEnd() throws Throwable {
    final MethodHandle start = MethodHandles.zero(int.class);
    final MethodHandle end = MethodHandles.constant(int.class, 10);
```

```
    final MethodHandle init = MethodHandles.zero(int.class);
    final MethodHandle body = MethodHandles
        .lookup()
        .findStatic(CountedLoopTest.class, "body",
            MethodType.methodType(int.class, int.class, int.class));
    final MethodHandle countedLoop = MethodHandles
        .countedLoop(start, end, init, body);
    assertEquals(55, countedLoop.invoke());
  }
}
```

iteratedLoop

The method `MethodHandle iteratedLoop(MethodHandle iterator, MethodHandle init, MethodHandle body)` creates a method handle representing a loop that iterates over the values produced by an `Iterator<T>`. The method handle `iterator` must return a value of type `Iterator<T>`. The usage of `iteratedLoop()` is similar with `countedLoop()`, except that the implicit loop variable is a value of type `T` instead of `int`. You can also use a second loop variable in `iteratedLoop()`. In Listing 9-8, I use `iteratedLoop()` to iterate a `Iterator<String>` from the list of `Strings` to calculate the sum of the length of these `Strings`.

Listing 9-8. Example of `iteratedLoop()`

```
public class IteratedLoopTest {

  static int body(final int sum, final String value) {
    return sum + value.length();
  }

  @Test
  public void testIteratedLoop() throws Throwable {
    final MethodHandle iterator = MethodHandles.constant(
        Iterator.class,
        List.of("a", "bc", "def").iterator());
    final MethodHandle init = MethodHandles.zero(int.class);
    final MethodHandle body = MethodHandles
        .lookup()
        .findStatic(
            IteratedLoopTest.class,
            "body",
            MethodType.methodType(
                int.class,
                int.class,
                String.class));
    final MethodHandle iteratedLoop = MethodHandles
        .iteratedLoop(iterator, init, body);
    assertEquals(6, iteratedLoop.invoke());
  }
}
```

whileLoop and doWhileLoop

The method `MethodHandle whileLoop(MethodHandle init, MethodHandle pred, MethodHandle body)` creates a method handle representing a `while` loop. A `while` loop is defined by three method handles.

- `init`: Initializes an optional loop variable. In each iteration, the loop variable is passed to the `body` and updated with the return value of the `body`. The result of the loop execution is the final value of the loop variable.
- `pred`: Executes a predicate to test for loop exit.
- `body`: Executes the method body.

The method `MethodHandle doWhileLoop(MethodHandle init, MethodHandle body, MethodHandle pred)` creates a method handle representing a `do-while` loop. It has the same parameters as `whileLoop`, but with a different order. In a `doWhileLoop`, the parameter `body` comes before `pred`. This is because the `do-while` loop executes the `body` before `pred` in each iteration. All three of these parameters have the same meaning as in `whileLoop()`.

In Listing 9-9, the loop variable is an `int[]` array with two elements; the first element is the iteration variable that controls the loop exit, while the second elements contains the sum. `k` is the extra parameter that determines the number of iterations. The return value of method `init()` is the initial value of the loop variable. The return value of method `body()` is passed as the input of the next iteration. The return value of the created `MethodHandle` is the final value of the loop variable.

Listing 9-9. Example of `whileLoop()` and `doWhileLoop()`

```
public class WhileLoopTest {
    static int[] init(final int k) {
        return new int[]{1, 0};
    }

    static boolean pred(final int[] local, final int k) {
        return local[0] < k;
    }

    static int[] body(final int[] local, final int k) {
        return new int[]{local[0] + 1, local[1] + local[0]};
    }

    @Test
    public void testWhileLoop() throws Throwable {
        final MethodHandle init = getMethodHandle("init",
                MethodType.methodType(int[].class, int.class));
        final MethodHandle pred = getMethodHandle("pred",
                MethodType.methodType(boolean.class, int[].class, int.class));
        final MethodHandle body = getMethodHandle("body",
                MethodType.methodType(int[].class, int[].class, int.class));
        final MethodHandle whileLoop = MethodHandles.whileLoop(init, pred, body);
        assertEquals(55, ((int[]) whileLoop.invoke(11))[1]);
    }

    @Test
    public void testDoWhileLoop() throws Throwable {
        final MethodHandle init = getMethodHandle("init",
```

```
                MethodType.methodType(int[].class, int.class));
        final MethodHandle pred = getMethodHandle("pred",
                MethodType.methodType(boolean.class, int[].class, int.class));
        final MethodHandle body = getMethodHandle("body",
                MethodType.methodType(int[].class, int[].class, int.class));
        final MethodHandle doWhileLoop = MethodHandles
                .doWhileLoop(init, body, pred);
        assertEquals(55, ((int[]) doWhileLoop.invoke(11))[1]);
    }

    private MethodHandle getMethodHandle(final String name,
                                         final MethodType methodType)
            throws NoSuchMethodException, IllegalAccessException {
        return MethodHandles
                .lookup()
                .findStatic(WhileLoopTest.class, name, methodType);
    }
}
```

■ **Note** The method handle `body` in `countedLoop()`, `iteratedLoop()`, `whileLoop()`, and `doWhileLoop()` can have an optional loop variable. If the return type of `body` is `void`, then there will be no loop variable as the leading parameter of `body`. In this case, the return type of `init` is also `void`. When there is no loop variable, the first parameter of `body` will be the implicit loop variable.

Try-finally

The method `MethodHandle tryFinally(MethodHandle target, MethodHandle cleanup)` creates a method handle that wraps the target method handle in a `try-finally` block. The parameter `target` is the `MethodHandle` to wrap, while `cleanup` is the `MethodHandle` to invoke in the `finally` block. Any exception thrown in the execution of `target` is passed to `cleanup`. The exception will be rethrown unless `cleanup` throws an exception first. The return value of the method handle created by `tryFinally()` is the return value of `cleanup`. `cleanup` accepts two optional leading parameters:

- A `Throwable` represents the exception thrown by invoking the method handle `target`.
- Return value of the method handle `target`.

Except those two leading parameters, `target` and `cleanup` should have the same corresponding arguments and return types. In Listing 9-10, I have two different target method handles, `success` and `failure`. The method handle `tryFinallySuccess` that wraps the method `success()` returns the value `1`, but the method handle `tryFinallyFailure` that wraps the method `failure()` throws the `IllegalArgumentException`. The method `cleanup` is invoked in both invocations and outputs different information to the console.

Listing 9-10. Example of `tryFinally`

```
public class TryFinallyTest {

  static int success() {
    return 1;
  }

  static int failure() {
    throw new IllegalArgumentException("");
  }

  static int cleanup(final Throwable throwable, final int result) {
    if (throwable != null) {
      throwable.printStackTrace();
    } else {
      System.out.println("Success: " + result);
    }
    return result;
  }

  @Test(expected = IllegalArgumentException.class)
  public void testTryFinally() throws Throwable {
    final MethodHandle targetSuccess = getMethodHandle("success",
        MethodType.methodType(int.class));
    final MethodHandle targetFailure = getMethodHandle("failure",
        MethodType.methodType(int.class));
    final MethodHandle cleanup = getMethodHandle("cleanup",
        MethodType.methodType(int.class, Throwable.class, int.class));
    final MethodHandle tryFinallySuccess = MethodHandles
        .tryFinally(targetSuccess, cleanup);
    assertEquals(1, tryFinallySuccess.invoke());
    final MethodHandle tryFinallyFailure = MethodHandles
        .tryFinally(targetFailure, cleanup);
    tryFinallyFailure.invoke();
  }

  private MethodHandle getMethodHandle(final String name,
      final MethodType methodType)
      throws NoSuchMethodException, IllegalAccessException {
    return MethodHandles
        .lookup()
        .findStatic(TryFinallyTest.class, name, methodType);
  }
}
```

Summary

In this chapter, we discussed the enhancements to method handles in Java 9. You can now create method handles for array constructors, variable handles, loops, and try-finally blocks. In the next chapter, we'll discuss changes in concurrency.

CHAPTER 10

Concurrency

This chapter summarizes changes related to concurrency in Java 9.

CompletableFuture

Several new methods have been added to `java.util.concurrent.CompletableFuture` in Java 9.

Async

The `CompletableFuture<T> completeAsync(Supplier<? extends T> supplier, Executor executor)` and `CompletableFuture<T> completeAsync(Supplier<? extends T> supplier)` methods complete the `CompletableFuture` by using an asynchronous task to invoke the `Supplier` to get the result. The task is executed using the provided executor or the default executor. Listing 10-1 shows examples of how to use `completeAsync()`.

Listing 10-1. Examples of `completeAsync()`

```
final Long v1 = new CompletableFuture<Long>().completeAsync(() -> 1L).join();
final Long v2 = new CompletableFuture<Long>().completeAsync(() -> 1L,
    Executors.newSingleThreadExecutor()).join();
System.out.printf("%s %s%n", v1, v2);
```

Timeout

The method `CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)` completes the `CompletableFuture` exceptionally with a `TimeoutException` if it's not completed before the given timeout. In Listing 10-2, because the `CompletableFuture` is never completed, the code throws an `ExecutionException` with a `TimeoutException` as the cause.

Listing 10-2. Example of `orTimeout()`

```
try {
  new CompletableFuture<Long>().orTimeout(3, TimeUnit.SECONDS).get();
} catch (final InterruptedException | ExecutionException e) {
  e.printStackTrace();
}
```


F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_10

The method `CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)` completes the `CompletableFuture` with the given value if it's not completed before the given timeout. In Listing 10-3, the `CompletableFuture` is completed with value `1` after a three-second timeout.

Listing 10-3. Example of `completeOnTimeout()`

```
try {
  final Long value = new CompletableFuture<Long>()
      .completeOnTimeout(1L, 3, TimeUnit.SECONDS).get();
  System.out.println(value);
} catch (final InterruptedException | ExecutionException e) {
  e.printStackTrace();
}
```

Utilities

Table 10-1 shows the new utilities methods of `CompletableFuture` that have been added in Java 9.

Table 10-1. *Utilities Methods of `CompletableFuture`*

Method	Description
`<U> CompletableFuture<U> newIncompleteFuture()`	Returns a new incomplete `CompletableFuture`
`Executor defaultExecutor()`	Returns the default `Executor` used for async methods
`CompletableFuture<T> copy()`	Returns a new `CompletableFuture` that completes normally with the same value as this `CompletableFuture` or completes exceptionally with an exception thrown in this `CompletableFuture` as the cause
`CompletionStage<T> minimalCompletionStage()`	Returns a new `CompletionStage` that can only be completed by this `CompletableFuture`
`Executor delayedExecutor(long delay, TimeUnit unit, Executor executor)`	Returns a new `Executor` that submits tasks to the given executor after the given delay
`Executor delayedExecutor(long delay, TimeUnit unit)`	Similar to the preceding method, but uses the default executor
`<U> CompletionStage<U> completedStage(U value)`	Returns a new `CompletionStage` that is completed with the given value
`<U> CompletionStage<U> failedStage(Throwable ex)`	Returns a new `CompletionStage` that is completed exceptionally with the given exception
`<U> CompletableFuture<U> failedFuture(Throwable ex)`	Returns a new `CompletableFuture` that is completed exceptionally with the given exception

TimeUnit and ChronoUnit

Two static methods `ChronoUnit toChronoUnit()` and `TimeUnit of(ChronoUnit chronoUnit)` are added to the class `java.util.concurrent.TimeUnit` to convert between `TimeUnit` and `java.time.temporal.ChronoUnit`; see Listing 10-4.

Listing 10-4. Example of `TimeUnit` and `ChronoUnit`

```
public class TimeUnitTest {
  @Test
  public void testChronoUnit() throws Exception {
    assertEquals(TimeUnit.MINUTES, TimeUnit.of(ChronoUnit.MINUTES));
    assertEquals(ChronoUnit.SECONDS, TimeUnit.SECONDS.toChronoUnit());
  }
}
```

Queues

The new methods `forEach(Consumer<? super E> action)`, `removeAll(Collection<?> c)`, `removeIf(Predicate<? super E> filter)`, and `retainAll(Collection<?> c)` have been added to the following classes:

- `java.util.concurrent.ArrayBlockingQueue`
- `java.util.concurrent.ConcurrentLinkedDeque`
- `java.util.concurrent.ConcurrentLinkedQueue`
- `java.util.concurrent.LinkedBlockingDeque`
- `java.util.concurrent.LinkedBlockingQueue`
- `java.util.concurrent.LinkedTransferQueue`

Listing 10-5 shows some examples of how to use these new methods in different queue implementations.

Listing 10-5. Examples of New Methods in Queues

```
public class QueueTest {

  @Test
  public void testForEach() {
    final ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(3);
    queue.offer(1);
    queue.offer(2);
    queue.offer(3);
    queue.forEach(System.out::println);
  }

  @Test
  public void testRemoveAll() {
    final LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue<>(3);
    queue.offer(1);
    queue.offer(2);
    queue.removeAll(List.of(1));
    assertEquals(1, queue.size());
  }
```

```
  @Test
  public void testRemoveIf() {
    final ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();
    queue.offer(1);
    queue.offer(2);
    queue.offer(3);
    queue.removeIf(i -> i % 2 == 0);
    assertEquals(2, queue.size());
  }

  @Test
  public void testRetainAll() {
    final LinkedBlockingDeque<Integer> deque = new LinkedBlockingDeque(3);
    deque.offer(1);
    deque.offer(2);
    deque.offer(3);
    deque.retainAll(List.of(1));
    assertEquals(1, deque.size());
  }
}
```

Atomic Classes

Those new methods in the class `VarHandle` related to the memory access modes are also added to existing atomic classes in the package `java.util.concurrent.atomic`, including `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray`, `AtomicReference`, and `AtomicReferenceArray`. These methods have the same memory semantics as the corresponding methods in `VarHandle`.

- `compareAndExchange()`
- `compareAndExchangeAcquire()`
- `compareAndExchangeRelease()`
- `weakCompareAndSetVolatile()`
- `weakCompareAndSetAcquire()`
- `weakCompareAndSetRelease()`
- `weakCompareAndSetPlain()`
- `getAcquire()`
- `getOpaque()`
- `getPlain()`
- `setOpaque()`
- `setPlain()`
- `setRelease()`

Listing 10-6 shows some examples of how to use these new methods.

Listing 10-6. Examples of New Methods in Atomic Classes

```
public class AtomicTest {

  @Test
  public void testGetAcquire() {
    final AtomicBoolean value = new AtomicBoolean();
    value.setRelease(false);
    assertEquals(false, value.getAcquire());
  }

  @Test
  public void testCompareAndExchange() {
    final AtomicInteger value = new AtomicInteger(10);
    final int returned = value.compareAndExchange(10, 5);
    assertEquals(10, returned);
    assertEquals(5, value.getPlain());
  }
}
```

Thread.onSpinWait

When you're using a thread to run a task, it's common to have the task wait for certain condition before it can continue. You usually do this by using a `volatile` variable as the flag. This flag is set by another thread to stop the waiting. Before Java 9, you used an empty loop to wait for the condition. The new static method `onSpinWait()` of class `Thread` in Java 9 can make the waiting more efficient.

Listing 10-7 shows an example of `Thread.onSpinWait()`. Classes `NormalTask` and `SpinWaitTask` both use the `volatile` Boolean variable `canStart` as the flag. The flag is set to `true` using the method `start()`. The different between these two classes is that `SpinWaitTask` uses `Thread.onSpinWait()` in the loop. Here I use another thread to call the method `start()` after three seconds to stop the waiting.

Listing 10-7. Example of `Thread.onSpinWait()`

```
public class ThreadOnSpinWait {

  public static void main(final String[] args) throws InterruptedException {
    final NormalTask task1 = new NormalTask();
    final SpinWaitTask task2 = new SpinWaitTask();
    final Thread thread1 = new Thread(task1);
    thread1.start();
    final Thread thread2 = new Thread(task2);
    thread2.start();
    new Thread(() -> {
      try {
        Thread.sleep(3000);
      } catch (final InterruptedException e) {
        e.printStackTrace();
      } finally {
        task1.start();
        task2.start();
      }
```

```
    }).start();
    thread1.join();
    thread2.join();
  }

  private abstract static class Task implements Runnable {

    volatile boolean canStart;

    void start() {
      this.canStart = true;
    }
  }

  private static class NormalTask extends Task {

    @Override
    public void run() {
      while (!this.canStart) {

      }
      System.out.println("Done!");
    }
  }

  private static class SpinWaitTask extends Task {

    @Override
    public void run() {
      while (!this.canStart) {
        Thread.onSpinWait();
      }
      System.out.println("Done!");
    }
  }
}
```

Summary

In this chapter, we discussed changes related to concurrency in Java 9, including asynchronous task execution and timeout support in `CompletableFuture`, new methods added to queues and atomic classes, and the new method `Thread.onSpinWait()`. In the next chapter, we'll discuss changes in the JavaScript engine Nashorn.

CHAPTER 11

Nashorn

Nashorn (http://openjdk.java.net/projects/nashorn/) is the JavaScript engine that was introduced in Java 8 to replace the old engine based on Mozilla Rhino. In Java 8 Nashorn was based on ECMAScript 5. Some new features of ECMAScript 6 have already been implemented in the Nashorn engine in Java 9. The Nashorn-related API is in the module `jdk.scripting.nashorn`.

Getting the Nashorn Engine

You can use the JSR 223 (https://www.jcp.org/en/jsr/detail?id=223) `ScriptEngine` to get the Nashorn engine; see Listing 11-1. The expression `new ScriptEngineManager().getEngineByName("Nashorn")` is the standard way to get an instance of the Nashorn `ScriptEngine`. However, the created `ScriptEngine` is only for ECMAScript 5. You need to pass the extra argument `--language=es6` to enable ECMAScript 6 features. Listing 11-1 shows you how to use the class `NashornScriptEngineFactory` directly to create the `ScriptEngine` for ECMAScript 6.

Listing 11-1. Creating the Nashorn `ScriptEngine`

```
public class NashornTest {
  private final ScriptEngine es5Engine =
      new ScriptEngineManager().getEngineByName("Nashorn");
  private final ScriptEngine es6Engine =
      new NashornScriptEngineFactory().getScriptEngine("--language=es6");

  @Test
  public void testSimpleEval() throws Exception {
    assertEquals(2, es5Engine.eval("1 + 1"));
  }
}
```

You can also use the `jjs` tool to try out ECMAScript 6's new features.

```
$ jjs --language=es6
```


F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_11

ECMAScript 6 Features

The following sections show ECMAScript 6's features that are supported by Nashorn in Java 9.

Template Strings

Nashorn supports the ECMAScript 6 template strings (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/template_strings), untagged and tagged; see Listing 11-2.

Listing 11-2. Template Strings

```
@Test
public void testTemplateString() throws Exception {
  final Bindings bindings = new SimpleBindings();
  bindings.put("name", "Alex");
  final Object result = this.es6Engine.eval("Hello ${name}", bindings);
  assertEquals("Hello Alex", result);
}
```

Binary and Octal Literals

Nashorn also supports binary (`b`) and octal (`o`) literals; see Listing 11-3.

Listing 11-3. Binary and Octal Literals

```
@Test
public void testBinaryAndOctalLiterals() throws Exception {
  assertEquals(503, this.es6Engine.eval("0b111110111"));
  assertEquals(503, this.es6Engine.eval("0o767"));
}
```

Iterators and for..of Loops

Nashorn supports iterators and `for..of` loops. The JavaScript code in Listing 11-4 creates a `random` iterator to generate random numbers, then it uses a `for..of` loop to get the first `10` elements in the iterator.

Listing 11-4. JavaScript Iterator

```
let random = {
  [Symbol.iterator]() {
  return {
      next() {
        return { done: false, value: Math.random() }
      }
    }
  }
}
```

```
let result = [];
for (var n of random) {
  if (result.length >= 10)
    break;
  result.push(n);
}

result
```

The method `eval()` in Listing 11-5 takes a JavaScript file name and evaluates it with a `Bindings` object. Since the JavaScript code in Listing 11-4 returns an array, the return value of evaluation is a `jdk.nashorn.api.scripting.ScriptObjectMirror` object. In Listing 11-5, I use its method `isArray()` to verify that the value is an array and check that the array size is 10.

Listing 11-5. Evaluating JavaScript in Files

```
@Test
public void testIterator() throws Exception {
  final ScriptObjectMirror result =
      (ScriptObjectMirror) eval("iterator", null);
  assertTrue(result.isArray());
  assertEquals(10, result.size());
}

private Object eval(final String fileName, final Bindings inputBindings)
    throws ScriptException {
  final Bindings bindings = Optional.ofNullable(inputBindings)
      .orElse(new SimpleBindings());
  return this.es6Engine.eval(
      new InputStreamReader(
          NashornTest.class.getResourceAsStream(
              String.format("/%s.js", fileName)
          )),
      bindings
  );
}
```

Functions

In Nashorn, you can use the arrow functions (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions). Listing 11-6 is the content of the JavaScript file `function.js`. It uses the arrow to create the function `add`.

Listing 11-6. JavaScript Arrow Functions

```
let add = (a, b) => a + b;

add(1, 2)
```

In Listing 11-7, I verify the result of the function invocation in Listing 11-6 using the method `eval()`.

Listing 11-7. Verifying the Result of the Function Invocation

```
@Test
public void testFunction() throws Exception {
  final Object result = eval("function", null);
  assertEquals(3, result);
}
```

Parser API

Nashorn has a standard parser API you can use to parse ECMAScript source code into abstract syntax trees (ASTs). This API is useful when you want to perform analysis of ECMAScript source code. With this API, you can get information about the source code, but you cannot modify existing source code. The parser API is in the package `jdk.nashorn.api.tree`.

Basic Parsing

In the method `testSimpleParser()` of Listing 11-8, I create a new instance of `Parser` using the method `Parser.create(String… options)`. The parameter `options` is a list of options to configure the parser's behavior. Here I use `--language=es6` to enable the ECMAScript 6 parsing mode. `Parser` has different `parse()` methods for parsing source code. All these `parse()` methods take arguments to specify the source code using `File`, `Reader`, `String`, `URL`, `Path`, or `jdk.nashorn.api.scripting.ScriptObjectMirror` objects and use another argument of type `DiagnosticListener` to specify a listener to receive parsing errors. Here I parse the JavaScript code `function a() {return 'hello';}`, which only contains one function declaration. The interface `DiagnosticListener` only contains one method `report(Diagnostic diagnostic)`. The interface `Diagnostic` contains methods for retrieving information about diagnostic data, including kind, code, message, line number, column number, and file name; see Table 11-1. Here I simply print out the data to console.

Table 11-1. *Methods of `Diagnostic`*

Method	Description
`Diagnostic.Kind getKind()`	Returns what kind of diagnostic this is
`long getPosition()`	Returns the character offset that indicates the location of the problem
`String getFileName()`	Returns the source file name
`long getLineNumber()`	Returns the line number of the character offset returned by `getPosition()`
`long getColumnNumber()`	Returns the column number of the character offset returned by `getPosition()`
`String getCode()`	Returns a code indicating the type of diagnostic
`String getMessage()`	Returns a message for this diagnostic

The enum `Diagnostic.Kind` represents different kinds of diagnostics. It has the following values: `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE`, and `OTHER`.

The return value of `parse()` is an instance of the interface `CompilationUnitTree`, which represents the parsed abstract syntax tree. The method `accept(TreeVisitor<R,D> visitor, D data)` accepts a visitor to visit the tree using the visitor pattern. Nashorn provides classes, `SimpleTreeVisitorES5_1` and `SimpleTreeVisitorES6`, as simple implementations for ECMAScript 5.1 and 6, respectively. Listing 11-8 extends the class `SimpleTreeVisitorES6` and overrides the method `visitFunctionDeclaration()` to verify that there is one function declaration in the source code.

Listing 11-8. Simple JavaScript Code Parsing

```
public class ParserAPITest {
  @Test
  public void testSimpleParser() throws Exception {
    final Parser parser = Parser.create("--language=es6");
    final CompilationUnitTree tree =
        parser.parse(
            "test.js",
            "function a() {return 'hello';}",
            System.out::println
        );
    final List<String> functions = new ArrayList<>();
    tree.accept(new SimpleTreeVisitorES6<Void, Void>() {
      @Override
      public Void visitFunctionDeclaration(
          final FunctionDeclarationTree node,
          final Void r) {
        final String name = node.getName().getName();
        assertEquals("a", name);
        functions.add(name);
        return null;
      }
    }, null);
    assertEquals(1, functions.size());
  }
}
```

Parsing Error

In Listing 11-9, I parse the JavaScript code that has a syntax error. I provide an implementation of `DiagnosticListener` that verifies the line number of the error.

Listing 11-9. JavaScript Code Parse Error

```
@Test
public void testParseError() throws Exception {
  final Parser parser = Parser.create("--language=es6");
  final List<Diagnostic> errors = new ArrayList<>();
  parser.parse(
    "error.js",
    "function error() { var a = 1;",
    diagnostic -> {
      assertEquals(1, diagnostic.getLineNumber());
```

```
      assertEquals(Diagnostic.Kind.ERROR, diagnostic.getKind());
      errors.add(diagnostic);
    }
  );
  assertEquals(1, errors.size());
}
```

Analyzing Function Complexity

Now I'll show you a complicated example using the parser API. Listing 11-10 shows the class `FunctionLengthAnalyzer` that analyzes the length of functions in the source code. I add visitors for `FunctionDeclarationTree` and `FunctionExpressionTree`. To measure the length of a function, I first get the start and end character offset of the function body in the source code using the methods `getStartPosition()` and `getEndPosition()`, respectively. Then I calculate the number of characters in the method body by subtracting these two offsets. For `FunctionExpressionTree` nodes, the function can be anonymous, so I create a name for these anonymous functions. Here I analyze the source code of jQuery 3.

Listing 11-10. Analyzing the Length of Functions

```
import com.google.common.collect.Lists;
import io.vavr.Tuple2;
import jdk.nashorn.api.tree.*;

import java.io.IOException;
import java.net.URL;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

public class FunctionLengthAnalyzer {
  public List<Tuple2<String, Long>> analyze(final URL url) throws IOException {
    final Parser parser = Parser.create();
    final CompilationUnitTree tree = parser.parse(url, System.out::println);
    final List<Tuple2<String, Long>> result = Lists.newArrayList();
    tree.accept(new SimpleTreeVisitorES5_1<Void, Void>() {
      @Override
      public Void visitFunctionDeclaration(
          final FunctionDeclarationTree node,
          final Void r) {
        result.add(new Tuple2<>(
            node.getName().getName(),
            node.getBody().getEndPosition() - node.getBody().getStartPosition()));
        return super.visitFunctionDeclaration(node, r);
      }

      @Override
      public Void visitFunctionExpression(
          final FunctionExpressionTree node,
          final Void r) {
        final String name = Optional.ofNullable(node.getName())
```

```
            .map(IdentifierTree::getName)
            .orElse("anonymous_" + node.getBody().getStartPosition());
        result.add(new Tuple2<>(
            name,
            node.getBody().getEndPosition() - node.getBody().getStartPosition()));
        return super.visitFunctionExpression(node, r);
      }
    }, null);
    Collections.sort(result,
        Collections.reverseOrder(Comparator.comparingLong(Tuple2::_2)));
    return result;
  }

  public static void main(final String[] args) throws IOException {
    final List<Tuple2<String, Long>> result =
        new FunctionLengthAnalyzer().analyze(
            new URL("https://code.jquery.com/jquery-3.2.1.js"));
    result.forEach(tuple2 ->
        System.out.printf("%30s -> %s%n", tuple2._1, tuple2._2));
  }
}
```

The function length calculated in Listing 11-10 is not very intuitive. It would be better if I could get the start and end line number of a function body and then calculate the lines of code for a function. Unfortunately, the line number is not exposed in the parser API. I can only use the character offset in the source code. For `FunctionDeclarationTree` nodes, the method `getBody()` returns a `BlockTree` object that has the method `getStatements()` to return a list of statements in the method body. I can also use the number of statements to measure the complexity of a function.

Summary

In this chapter, we discussed the new ECMAScript 6 features supported by Nashorn in Java 9 and the new parser API you can use to parse JavaScript code. In the next chapter, we'll discuss changes related to I/O in Java 9.

CHAPTER 12

I/O

This chapter covers the changes to the `java.io` package in Java 9.

InputStream

The class `java.io.InputStream` adds three methods for reading and copying data from the input stream.

- `readAllBytes()`: Reads all remaining bytes from the input stream.
- `readNBytes(byte[] b, int off, int len)`: Reads the given number of bytes from the input stream into the given byte array `b`. `offset` is the reading position, while `len` is the number of bytes to read.
- `transferTo(OutputStream out)`: Reads all bytes from the input stream and writes to the given output stream.

In Listing 12-1, the file `input.txt` contains the content "Hello World". I use the methods `readAllBytes()` and `readNBytes()` to read data from the input stream and create strings from the data. I also use the method `transferTo()` to copy the data to a `ByteArrayOutputStream`.

Listing 12-1. Methods in `InputStream` for Reading and Copying Data

```
public class TestInputStream {

  private InputStream inputStream;
  private static final String CONTENT = "Hello World";

  @Before
  public void setUp() throws Exception {
    this.inputStream =
        TestInputStream.class.getResourceAsStream("/input.txt");
  }

  @Test
  public void testReadAllBytes() throws Exception {
    final String content = new String(this.inputStream.readAllBytes());
    assertEquals(CONTENT, content);
  }
```


F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_12

```
  @Test
  public void testReadNBytes() throws Exception {
    final byte[] data = new byte[5];
    this.inputStream.readNBytes(data, 0, 5);
    assertEquals("Hello", new String(data));
  }

  @Test
  public void testTransferTo() throws Exception {
    final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    this.inputStream.transferTo(outputStream);
    assertEquals(CONTENT, outputStream.toString());
  }
}
```

Now that these three new methods have been added, you no longer need to use other third-party libraries like Apache Commons IO for some common usage scenarios.

The ObjectInputStream Filter

You can use Java object serialization to transform object graphs for persistence or network transfer. When deserializing the data to re-create the objects, make sure you check the input to make sure the data is valid; otherwise malicious content may compromise the system. You can set an implementation of the new interface `java.io.ObjectInputFilter` on an `ObjectInputStream` to filter the input.

`ObjectInputFilter` is a functional interface with only one method: `ObjectInputFilter.Status checkInput(ObjectInputFilter.FilterInfo filterInfo)`. The interface `ObjectInputFilter.FilterInfo` contains information about the object being deserialized and the statistics of the `ObjectInputStream` object; see Table 12-1.

Table 12-1. *Methods of `ObjectInputFilter.FilterInfo`*

Method	Description
`Class<?> serialClass()`	Returns the class of the object being deserialized
`long arrayLength()`	If the object being deserialized is an array, returns the length of the array
`long depth()`	Returns the depth of the current object in the object graph
`long references()`	Returns the current number of object references
`long streamBytes()`	Returns the current number of bytes consumed

The return value of `checkInput()` is the enum `ObjectInputFilter.Status`; see Table 12-2.

Table 12-2. *Values of `ObjectInputFilter.Status`*

Status	Description
`ALLOWED`	The object is allowed to be deserialized.
`REJECTED`	The object is not allowed to be deserialized.
`UNDECIDED`	This filter doesn't decide whether the object is allowed to be deserialized.

An ObjectInputFilter should only check for the condition that it's designed for and return ALLOWED or REJECTED only when it knows the result; otherwise it should return UNDECIDED to let other filters decide the result. If REJECTED is returned as the status, then the ObjectInputStream throws the java.io.InvalidClassException.

ObjectInputFilters can be set on individual ObjectInputStream objects using the new method setObjectInputFilter(). You can also set a process-wide filter using the static method setSerialFilter(ObjectInputFilter filter) of the class ObjectInputFilter.Config. If the individual ObjectInputFilter is not set, then the process-wide filter will be applied.

Another important feature of ObjectInputFilter.Config is that it can create ObjectInputFilters from string patterns. String patterns are much easier to write than an ObjectInputFilter interface is to implement. A filter can have multiple patterns separated by semicolons (;). A pattern can set a limit or specify the class name to match. The following limits are supported:

- maxdepth=value: The maximum depth of a graph
- maxrefs=value: The maximum number of object references
- maxbytes=value: The maximum number of bytes in the input stream
- maxarray=value: The maximum length of arrays

For example, you can use maxdepth=5 to limit the maximum depth of a graph to 5.

Other patterns match the class names. Here are some scenarios:

- If the pattern contains "/", then it specifies both the module name and the class name to match.
- If the pattern ends with ".**", then it matches any class in the package and subpackages.
- If the pattern ends with ".*", then it matches any class in the package.
- If the pattern ends with "*", then it matches any class with the pattern as the prefix.
- If the pattern is equal to the class name, then it matches the class name.
- If the pattern starts with "!", then it negates the pattern after it.

Listing 12-2 shows some examples of using ObjectInputFilters to filter input streams. Nested classes A, B, C, and D are used to demonstrate an object graph. In the method setUp(), I create a new instance of class A and use ObjectOutputStream to get the serialized bytes. objectInput is the ByteArrayInputStream object for the serialized data. In the test case testFilterByClass, I check to make sure that objects of class B do not exist in the input stream. I expect the exception InvalidClassException to be thrown. In the test case testFilterByDepth, I check to make sure the maximum depth of object graphs is not greater than 6. In the test case testProcessWideFilter, I configure a process-wide filter to check for class C. In the test case testFilterPattern, I use the pattern "!io.vividcode.feature9.** " to specify that all classes in the package io.vividcode.feature9 and its subpackages are not allowed.

Listing 12-2. Using ObjectInputFilter to Filter Input Streams

```
public class TestObjectInputFilter {

  static class A implements Serializable {

    public int a = 1;
    public B b = new B();
  }
```

```
  static class B implements Serializable {

    public String b = "hello";
    public C c = new C();
  }

  static class C implements Serializable {

    public int c = 2;
    public D[] ds = new D[]{new D(), new D()};
  }

  static class D implements Serializable {

    public String d = "world";
  }

  private ByteArrayInputStream objectInput;

  @Before
  public void setUp() throws Exception {
    final A a = new A();
    final ByteArrayOutputStream baos = new ByteArrayOutputStream();
    final ObjectOutputStream outputStream = new ObjectOutputStream(baos);
    outputStream.writeObject(a);
    this.objectInput = new ByteArrayInputStream(baos.toByteArray());
  }

  @Test(expected = InvalidClassException.class)
  public void testFilterByClass() throws Exception {
    final ObjectInputStream inputStream = new ObjectInputStream(this.objectInput);
    inputStream.setObjectInputFilter(filterInfo -> {
      if (B.class.isAssignableFrom(filterInfo.serialClass())) {
        return ObjectInputFilter.Status.REJECTED;
      }
      return ObjectInputFilter.Status.UNDECIDED;
    });
    final A a = (A) inputStream.readObject();
    assertEquals(1, a.a);
  }

  @Test
  public void testFilterByDepth() throws Exception {
    final ObjectInputStream inputStream = new ObjectInputStream(this.objectInput);
    inputStream.setObjectInputFilter(filterInfo -> {
      if (filterInfo.depth() > 6) {
        return ObjectInputFilter.Status.REJECTED;
      }
      return ObjectInputFilter.Status.UNDECIDED;
    });
```

```
    final A a = (A) inputStream.readObject();
    assertEquals(1, a.a);
  }

  @Test(expected = InvalidClassException.class)
  public void testProcessWideFilter() throws Exception {
    ObjectInputFilter.Config.setSerialFilter(filterInfo -> {
      if (C.class.isAssignableFrom(filterInfo.serialClass())) {
        return ObjectInputFilter.Status.REJECTED;
      }
      return ObjectInputFilter.Status.UNDECIDED;
    });
    final ObjectInputStream inputStream = new ObjectInputStream(this.objectInput);
    final A a = (A) inputStream.readObject();
    assertEquals(1, a.a);
  }

  @Test(expected = InvalidClassException.class)
  @Ignore
  public void testFilterPattern() throws Exception {
    ObjectInputFilter.Config.setSerialFilter(
        ObjectInputFilter.Config.createFilter("!io.vividcode.feature9.**"));
    final ObjectInputStream inputStream = new ObjectInputStream(this.objectInput);
    final A a = (A) inputStream.readObject();
    assertEquals(1, a.a);
  }
}
```

The process-wide filter can be set exactly once. Trying to set it again causes the `IllegalStateException`. That's why the test case `testFilterPattern` is marked as ignored. If it weren't, the test case would fail with the `IllegalStateException` because the process-wide filter is already set in the test case `testProcessWideFilter`. When running the test cases using Maven, you can configure the Maven Surefire plugin to fork the JVM for each test method; then you can run both test cases successfully.

Summary

In this chapter, we discussed changes related to I/O in Java 9, including new methods in `InputStream` and `ObjectInputStream` filters. In the next chapter, we'll discuss changes related to security in Java 9.

CHAPTER 13

Security

This chapter covers changes related to security in Java 9.

SHA-3 Hash Algorithms

Java 9 adds four SHA-3 (https://en.wikipedia.org/wiki/SHA-3) hash algorithms to generate message digest: `SHA3-224`, `SHA3-256`, `SHA3-384`, and `SHA3-512`. Listing 13-1 shows an example of how to use the algorithm `SHA3-224` to generate a message digest.

Listing 13-1. Using SHA-3 to Generate Message Digest

```
import org.apache.commons.codec.binary.Hex;

public class SHA3 {

  public static void main(final String[] args) throws NoSuchAlgorithmException {
    final MessageDigest instance = MessageDigest.getInstance("SHA3-224");
    final byte[] digest = instance.digest("".getBytes());
    System.out.println(Hex.encodeHexString(digest));
  }
}
```

SecureRandom

The class `java.security.SecureRandom` is responsible for generating cryptographically strong random numbers. Java has several built-in algorithms to generate random numbers. Java 9 adds the new deterministic random bit generator (DRBG) algorithm. Because DRBG requires extra parameters to configure, the existing API is also updated to support it.

The new marker interface `SecureRandomParameters` represents parameters used in different `SecureRandom` methods. Currently only DRBG-related parameters implement this interface. `SecureRandomSpi` adds a new constructor that takes `SecureRandomParameters` as the parameter. The new method `void engineNextBytes(byte[] bytes, SecureRandomParameters params)` can use the `SecureRandomParameters` to generate random bytes. If the `SecureRandom` supports reseeding, the method `void engineReseed(SecureRandomParameters params)` should be overridden to reseed. The last new method `SecureRandomParameters engineGetParameters()` returns the effective `SecureRandomParameters` used by the engine.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_13

To use the SecureRandomParameters interface, SecureRandom has three new overloads of the static method getInstance() to add to the extra SecureRandomParameters parameter. The method nextBytes() also has a new overload with the SecureRandomParameters parameter. Since DRBG supports reseeding, new methods void reseed() and void reseed(SecureRandomParameters params) can reseed this SecureRandom with input from its source. The last new method SecureRandomParameters getParameters() returns the effective SecureRandomParameters for this SecureRandom instance.

The DRBG algorithm has three implementations of SecureRandomParameters to be used in different stages; see Table 13-1.

Table 13-1. *SecureRandomParameters Implementations*

Implementation	Description	Method to Use
DrbgParameters.Instantiation	Instantiation	getInstance
DrbgParameters.NextBytes	Random bits generation	nextBytes
DrbgParameters.Reseed	Reseed	reseed

DrbgParameters has static methods for creating instances of these three types of parameters. When creating a new DrbgParameters.Instantiation, you can specify the minimal capability requirement for DRBG with the enum DrbgParameters.Capability; see Table 13-2. DRBG capabilities include whether it's reseedable and supports prediction resistance. Because a DRBG implementation that supports prediction resistance must also support reseeding, there are only three options.

Table 13-2. *DrbgParameters.Capability Values*

Value	Description
NONE	Neither reseedable nor prediction resistant
RESEED_ONLY	Reseedable only
PR_AND_RESEED	Both reseedable and prediction resistant

The created DRBG SecureRandom instances may have more capabilities than requested. Once a SecureRandom instance is created, you can get the effective SecureRandomParameters using the method getParameters() and then check the actual capabilities.

Listing 13-2 shows an example of using the DRBG algorithm to generate random numbers.

Listing 13-2. Example of Using the DRBG Algorithm

```
public class DRBGRandom {

  public static void main(final String[] args) throws NoSuchAlgorithmException {
    final DrbgParameters.Instantiation instantiation =
        DrbgParameters.instantiation(
            256,
            Capability.PR_AND_RESEED,
            "HelloWorld".getBytes()
        );
    final SecureRandom secureRandom = SecureRandom.getInstance("DRBG", instantiation);
    final DrbgParameters.NextBytes nextBytes =
```

```
        DrbgParameters.nextBytes(-1, true, "Hello".getBytes());
    final byte[] result = new byte[32];
    secureRandom.nextBytes(result, nextBytes);
    System.out.println(Hex.encodeHexString(result));
    final DrbgParameters.Reseed reseed = DrbgParameters.reseed(true, null);
    secureRandom.reseed(reseed);
    secureRandom.nextBytes(result, nextBytes);
    System.out.println(Hex.encodeHexString(result));
  }
}
```

As a result of implementing DRBG, hash algorithms `SHA-512/224` and `SHA-512/256` and related `HmacSHA512/224` and `HmacSHA512/256` have also been added.

Using PKCS12 as the Default Keystore

JDK has used its proprietary keystore format, JKS, since JDK 1.2. The JKS keystore can only store private keys and trusted public-key certificates. PKCS12 (https://en.wikipedia.org/wiki/PKCS_12) is a widely supported format for storing keys. In Java 9, the default keystore type changes from JKS to PKCS12. Existing keystores are not changed and applications can still run without changes.

Summary

In this chapter, we discussed the new SHA-3 hash algorithm, the DRBG `SecureRandom` algorithm, and using the PKCS12 keystore. In the next chapter, we'll discuss changes related to the user interface in Java 9.

CHAPTER 14

User Interface

A large number of desktop applications are still developed in Java. Java 9 continues to improve on support for these desktop applications.

Desktop

The class `java.awt.Desktop` allows Java applications to interact with various desktop capabilities. Java 9 extends this support by adding more capabilities. However, these new capabilities may not be supported by all native platforms. If the underlying platform doesn't support a capability, the corresponding method has no effect. All these capabilities have an enum value in `Desktop.Action`. You can use the method `boolean isSupported(Desktop.Action action)` in `Desktop` to check whether or not an action is supported.

Application Events

The method `void addAppEventListener(SystemEventListener listener)` adds listeners for different kinds of system events from the native system. `SystemEventListener` has the following subinterfaces for different system events:

- `AppForegroundListener`: Gets notified when the app becomes the foreground app and when it's no longer the foreground app.
- `AppHiddenListener`: Gets notified when the app is hidden or shown by the user.
- `AppReopenedListener`: Gets notified when the app has been asked to open again.
- `ScreenSleepListener`: Gets notified when the displays have entered power save sleep and after displays have awoken from power save sleep.
- `SystemSleepListener`: Gets notified when the system is entering sleep and after the system wakes.
- `UserSessionListener`: Gets notified when the user session has been switched to and when the user session has been switched away from.

If the underlying system doesn't support a particular type of system event, adding the corresponding listener has no effect. You can remove added listeners using the method `void removeAppEventListener (SystemEventListener listener)`.

You can also use the method `void requestForeground(boolean allWindows)` to request that an application move to the foreground. If the parameter `allWindows` is `true`, all windows of the application will be moved to the foreground, otherwise only the foremost one will be moved.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_14

About Window

The method `void setAboutHandler(AboutHandler aboutHandler)` sets a custom handler to handle the request to show the **About** window for your application. Setting the `AboutHandler` to `null` can revert it to the default behavior.

Preferences Window

The method `void setPreferencesHandler(PreferencesHandler preferencesHandler)` sets a custom handler to handle the request to show the **Preferences** window for your application. Setting the `PreferencesHandler` to `null` can revert it to the default behavior.

Listing 14-1 shows how to use application events in `Desktop`. I use a `JTextArea` to display received system events. I also add `AboutHandler` and `PreferencesHandler`.

Listing 14-1. Application Events

```
public class SystemEventsDemo {

  private static class MainFrame extends JFrame {

    final JTextArea textArea = new JTextArea();

    public MainFrame() throws HeadlessException {
      final Container container = getContentPane();
      final BorderLayout layout = new BorderLayout(5, 5);
      container.setLayout(layout);
      this.textArea.setEditable(false);
      container.add(this.textArea, BorderLayout.CENTER);

      setTitle("System events");
      setSize(400, 300);
      setVisible(true);

      final Desktop desktop = Desktop.getDesktop();
      desktop.addAppEventListener(new AppForegroundListener() {
        @Override
        public void appRaisedToForeground(final AppForegroundEvent e) {
          log("App to foreground");
        }

        @Override
        public void appMovedToBackground(final AppForegroundEvent e) {
          log("App to background");
        }
      });

      desktop.setAboutHandler(e -> log("About"));

      desktop.setPreferencesHandler(e -> log("Preferences"));
    }
```

```
    private void log(final String message) {
      this.textArea.append(String
          .format("%s: %s%n", LocalDateTime.now()
                  .format(DateTimeFormatter.ISO_LOCAL_DATE_TIME),
              message));
    }
  }

  public static void main(final String[] args) {
    final MainFrame frame = new MainFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

Figure 14-1 shows the screenshot of the application in Listing 14-1 running on macOS. You can see the logs of system events. Clicking the default **About** and **Preferences** menu items triggers the `AboutHandler` and `PreferencesHandler` to log out messages, respectively.

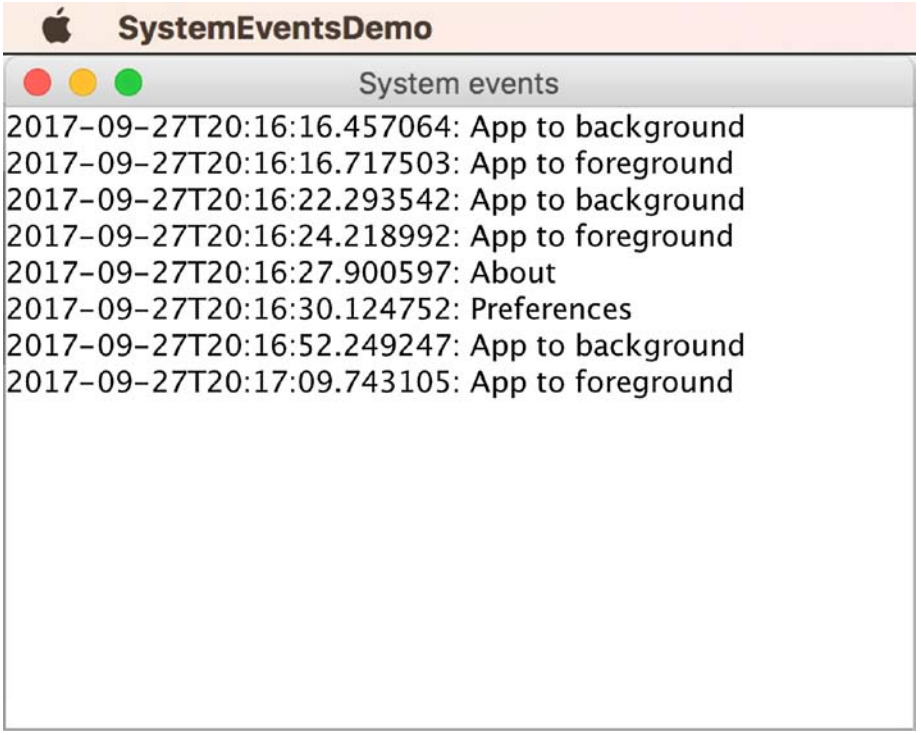

Figure 14-1. *Demo of system events*

Open Files

The method `void setOpenFileHandler(OpenFilesHandler openFileHandler)` sets a custom handler to get notified when the application is asked to open a list of files. The event object `OpenFilesEvent` received by `OpenFilesHandler` contains the list of files and the search term used to find the files.

Print Files

The method `void setPrintFileHandler(PrintFilesHandler printFileHandler)` sets a custom handler to get notified when the application is asked to print a list of files. The event object `PrintFilesEvent` received by `PrintFilesHandler` contains the list of files to print.

Open URI

The method `void setOpenURIHandler(OpenURIHandler openURIHandler)` sets a custom handler to get notified when the application is asked to open a URI. The event object `OpenURIEvent` received by `OpenURIHandler` contains the URI to open.

■ **Note** On macOS, notifications for `setOpenFileHandler()`, `setPrintFileHandler()`, and `setOpenURIHandler()` are only sent if the Java application is a bundled application with a `CFBundleDocumentTypes` array present in its `Info.plist`.

Application Exit

The method `void setQuitStrategy(QuitStrategy strategy)` sets the default strategy to quit the application. The enum `QuitStrategy` contains two values:

- `CLOSE_ALL_WINDOWS` means shutting down the application by closing each window from back-to-front.
- `NORMAL_EXIT` means shutting down the application by calling `System.exit(0)`.

The method `void setQuitHandler(QuitHandler quitHandler)` sets a custom handler to determine if the application should exit. The method `void handleQuitRequestWith(QuitEvent e, QuitResponse response)` receives two parameters: the first, `QuitEvent`, represents the event; the second, `QuitResponse`, represents the response to the quit request. In the implementation of method `handleQuitRequestWith()`, you should either call `QuitResponse.cancelQuit()` to cancel the quit request or `QuitResponse.performQuit()` to perform the default quit strategy.

The methods `void enableSuddenTermination()` and `void disableSuddenTermination()` enable and disable sudden termination of the application, respectively. If the application can be suddenly terminated, it is terminated without notifications. `QuitHandler` is not notified and shutdown hooks are not run. When your application's state is saved, you should call the method `enableSuddenTermination()` to indicate that. On the other hand, if your application's state is not saved, you should call the method `disableSuddenTermination()` to indicate that.

Listing 14-2 shows a demo of `QuitHandler`. In the implementation of `QuitHandler`, I show a confirm dialog and call `performQuit()` or `cancelQuit()` based on the user's response.

Listing 14-2. Quit Handler

```
public class QuitHandlerDemo {

  private static class MainFrame extends JFrame {

    public MainFrame() throws HeadlessException {
      setTitle("Quit Handler");
      setSize(400, 300);
```

```
      setVisible(true);
    }
  }

  public static void main(final String[] args) {
    final MainFrame frame = new MainFrame();
    frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    final Desktop desktop = Desktop.getDesktop();
    desktop.setQuitHandler((e, response) -> {
      final int result = JOptionPane
          .showConfirmDialog(frame, "Quit the app?", "Quit?", YES_NO_OPTION);
      if (result == YES_OPTION) {
        response.performQuit();
      } else {
        response.cancelQuit();
      }
    });
  }
}
```

Figure 14-2 shows the screenshot of the application in Listing 14-2 running on macOS. After clicking the **Quit** menu item, the confirmation dialog pops up. If you click **No**, the quit request is cancelled and the application keeps running.

Figure 14-2. *Demo of* `QuitHandler`

Other Functionalities

These other methods provide different functionalities.

- `void openHelpViewer()`: Opens the native help viewer application
- `void setDefaultMenuBar(JMenuBar menuBar)`: Sets the default menu bar when no frames are active
- `void browseFileDirectory(File file)`: Opens a folder that contains the file and selects it in a default system file manager
- `boolean moveToTrash(File file)`: Moves the file to the trash

Multiresolution Images

If you have developed iOS apps, you should be familiar with multiresolution images. An image asset in iOS apps can have multiple images with different resolutions. These images have the same base name, but they contain suffixes like `@1x`, `@2x` or `@3x`. The system will choose the most suitable image to show based on the current display DPI metric. Java 9 adds the same functionality to the AWT image API.

The new interface `java.awt.image.MultiResolutionImage` represents images that support multiple resolutions. `MultiResolutionImage` has two methods: the method `Image getResolutionVariant(double destImageWidth, double destImageHeight)` gets the specific image that should be rendered at the given size; the method `List<Image> getResolutionVariants()` gets a list of all resolution variants. The class `BaseMultiResolutionImage` is a simple implementation of `MultiResolutionImage`. `BaseMultiResolutionImage` is constructed from an array of images. Its implementation of the method `getResolutionVariant()` simply iterates through the array of images and returns the first image that is large enough to satisfy the rendering request.

Most of the time, you don't need to use `BaseMultiResolutionImage` directly. The images obtained from methods `Toolkit.getImage(String name)` and `Toolkit.getImage(URL url)` will implement the interface `MultiResolutionImage` if the platform supports naming conventions for resolution variants.

Listing 14-3 shows an example of using multiresolution images. I have two images `flower.png` and `flower@2x.png` in the resources directory `/images`. The method `Toolkit.getImage(URL url)` is used to load images using the resource URL.

Listing 14-3. Multiresolution Images

```
public class MultiResolutionImageDemo {

  private static class ImageCanvas extends JComponent {

    private final String name;

    public ImageCanvas(final String name) {
      this.name = name;
    }

    @Override
    public void paint(final Graphics g) {
      final Image image = Toolkit.getDefaultToolkit()
          .getImage(
              MultiResolutionImageDemo.class
```

```
                .getResource(String.format("/images/%s", this.name)));
      g.drawImage(image, 10, 10, this);
      g.dispose();
    }
  }

  public static void main(final String[] args) {
    final JFrame frame = new JFrame();
    frame.setTitle("Multi-resolution images");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLayout(new BorderLayout());
    final ImageCanvas canvas = new ImageCanvas("flower.png");
    frame.add("Center", canvas);
    frame.setSize(360, 300);
    frame.setVisible(true);
  }
}
```

When running on macOS, you can see that the actual loaded image is `flower@2x.png`. As shown in Figure 14-3, the "2x" in the image confirms that it's the `flower@2x.png`.

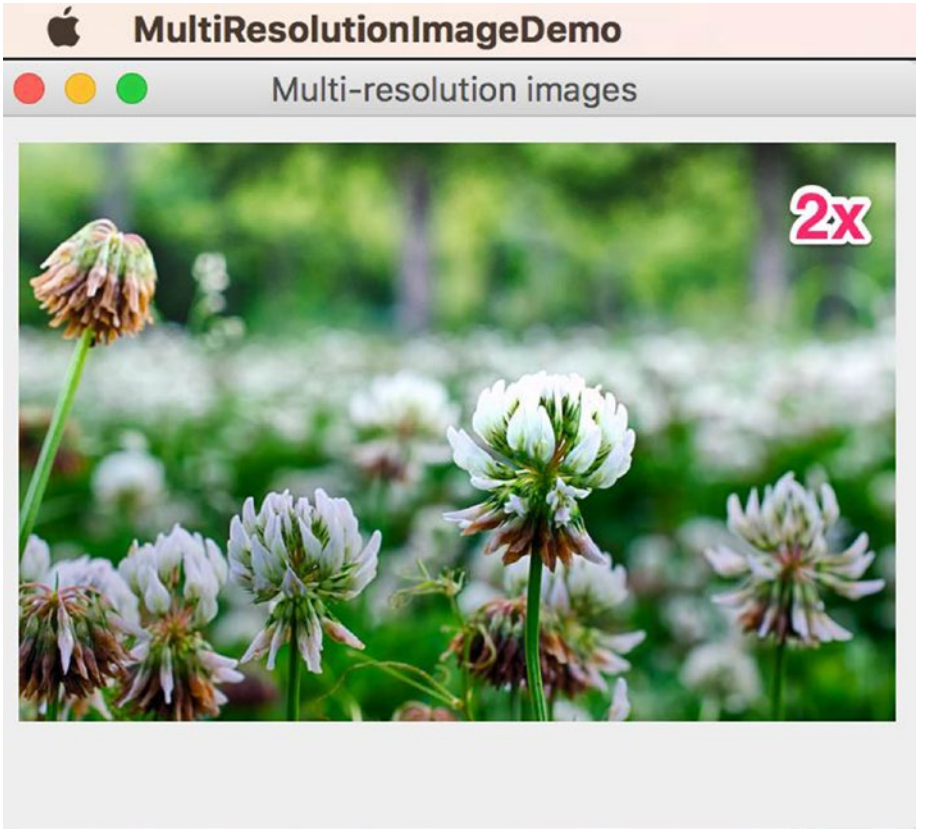

Figure 14-3. *Demo of multiresolution images*

TIFF Image Format

Java 9 has added support for the TIFF image format. The Image I/O framework now adds codecs for TIFF format.

Deprecating the Applet API

The Applet API has been deprecated in Java 9 as browser vendors remove support for Java browser plugins. The annotation `@Deprecated(since="9")` is added to following Applet-related classes:

- `java.applet.AppletStub`
- `java.applet.Applet`
- `java.applet.AudioClip`
- `java.applet.AppletContext`
- `javax.swing.JApplet`

The `appletviewer` tool is also deprecated. However, the Applet API will not be removed in the next major release. Since the usage of Applet is quite rare these days, this deprecation should only have minor impact.

Summary

In this chapter, we discussed the new features added to AWT `Desktop`, multiresolution images, the TIFF image format, and the depreciation of the Applet API. In the next chapter, we'll discuss changes related to the JVM.

CHAPTER 15

JVM

This chapter covers changes related to the JVM in Java 9.

Unified Logging

In Java 9, JVM has a common logging system for all components. The new command-line option `-Xlog` controls logging for all components of the JVM. The new unified logging system supports tags, levels, decorations, and output.

Tags, Levels, Decorations, and Output

Tags are used to classify log messages. A log message can be associated with a tag set that consists of one or more tags. JVM has a predefined set of tags, for example, `gc`, `system`, `thread`. Each log message has a logging level associated with it, just like the messages logged using Java's logging frameworks. The available levels are `error`, `warning`, `info`, `debug`, `trace`, and `develop`. `develop` is only available in nonproduct builds. Logging messages are decorated with information about the message. The following is a list of possible decorations:

- `time`: Current time and date in ISO-8601 format
- `utctime`: Current UTC time and date ISO-8601 format
- `uptime`: Time since the start of JVM in seconds and milliseconds
- `timemillis`: The same value as generated by `System.currentTimeMillis()`
- `uptimemillis`: Milliseconds since JVM started
- `timenanos`: The same value as generated by `System.nanoTime()`
- `uptimenanos`: Nanoseconds since JVM started
- `hostname`: The hostname
- `pid`: The process identifier
- `tid`: The thread identifier
- `level`: The logging level
- `tags`: The tag set

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_15

Output is the destination of the log messages. There are three types of supported output:

- `stdout`: Outputs to `stdout`.
- `stderr`: Outputs to `stderr`.
- *text file*: Outputs to text files.

Text file output supports file rotation based on the size and the maximum number of files.

For each output, you can configure a logging level to control the amount of log messages written to it. For example, if the output's logging level is set to `info`, then only messages with levels `error`, `warning`, and `info` will be written. This is the same as using other logging frameworks in Java. The logging level can also be set to `off` to disable logging. You can also configure the set of decorators used by the output. Decorations will be prepended to the log messages.

Logging Configuration

The configuration to the JVM logging system is done through the option `-Xlog`. The syntax of the option is a little bit complicated. The best way to understand the syntax is by running the command `java -Xlog:help`; then the help message will be printed out in the console. The option contains four parts separated by colons (`:`) in the format of `-Xlog[:[what][:[output][:[decorators][:output-options]]]]`.

The first part, `what`, selects the log messages to output. Its value is a comma-separated list of selectors. Each selector contains a tag set and an optional logging level. Tags in the tag set are separated by plus (`+`). The tag set and logging level are separated by an equal sign (`=`). Wildcard (`*`) can be used to match any number of tags. All available tags are listed in the output of `-Xlog:help`. The default logging level is `info`. The following are some examples of the first part:

- `gc`: Selects the tag `gc` with level `info`
- `gc+heap=debug`: Selects the tags `gc` and `heap` with level `debug`
- `gc=debug,heap=warning`: Selects the tag `gc` with level `debug` and the tag `heap` with level `warning`
- `*`: Selects all tags with level `info`
- `*=debug`: Selects all tags with level `debug`

If a tag set contains multiple tags, a log message can only be selected when it's associated exactly with all the tags in the tag set. When there are multiple selectors, a selected log message must match all selectors.

The second part, `output`, configures the output of log messages. Here are some possible values:

- `stdout`
- `stderr`
- `file=<filename>`: Supports `%p` and/or `%t` in the filename to use the JVM's PID and startup timestamp, respectively.

The third part, `decorators`, is a list of comma-separated decorators. You can use `none` to disable decorations. The default value of decorators is `"uptime,level,tags"`.

The last part, `output-options`, is the extra comma-separated options for output. Each option is a name-value pair separated by an equal sign (`=`). There are two options supported:

- `filecount`: The maximum number of rolling log files
- `filesize`: The maximum size of each log file

By combining these four parts, you can configure the behavior of the logging system. Here are some examples:

- `-Xlog`: Equivalent to `-Xlog:all=info:stdout:uptime,levels,tags`.
- `-Xlog:gc,class`: Log messages associated with the tag `gc` or `class`, using level `info`, to `stdout`, with default decorations. Messages associated with both gc and class will not be logged.
- `-Xlog:gc+class`: Log messages associated with both the tags `gc` and `class`, using level `info`, to `stdout`, with default decorations. Messages associated with only one of the two tags will not be logged.
- `-Xlog:gc=debug:file=gc_%p.txt`: Log messages associated with tag `gc` using level `debug` to the file. The file name has JVM's PID.
- `-Xlog:gc=debug:file=gc.txt:time:filecount=5,filesize=1m`: Log messages associated with tag `gc` using level `debug` to the file `gc.txt`. The maximum number of rolling files is `5` and each file has a maximum size of `1M`.

You can also use `-Xlog:disable` to disable all loggings, including errors and warnings.

With the new unified logging system, GC logging has been updated to leverage the same system. All GC-related log messages are associated with the tag `gc`. `-Xlog:gc` should be used to replace `-XX:+PrintGC` and `-XX:+PrintGCDetails`.

The Diagnostic Command VM.log

Logging configuration can be updated at runtime using the diagnostic command `VM.log` through `jcmd`. All the options supported in `-Xlog` can be specified dynamically with this command. Since diagnostic commands are automatically exposed as MBeans, you can also use the JMX to change logging configuration in runtime.

To configure the logging at runtime, you use `jcmd` first to list all Java processes and locate the PID of the application to configure. You can also use the main class name to select the process to check. The option `list` in the following command lists current logging configuration. `31683` is the PID.

```
$ jcmd 31683 VM.log list
```

You can use key-value pairs to specify the values for the four parts mentioned earlier: `what`, `output`, `decorators`, and `output-options`. Using the following command, you update the logging configuration to be like `-Xlog:all=log:file=all.txt:time,level,tags`.

```
$ jcmd 31683 VM.log what=all=info output=all.txt decorators=time,level,tags
```

To disable the logging, you can use the following command:

```
$ jcmd 31683 VM.log disable
```

You can also force the log files to rotate using this command:

```
$ jcmd 31683 VM.log rotate
```

Remove GC Combinations

GC tuning is an important task in tuning your JVM performance. JVM has a lot of GC combinations that use different GC algorithms. Some of these GC combinations were already deprecated in Java 8. These deprecated GC combinations have been removed in Java 9; see Table 15-1. The JVM will not start if these GC combinations are used. Since JVM already prints out warning messages for these GC combinations in Java 8, existing applications should already have migrated to use other GC combinations. If your application still uses these deprecated GC combinations, you should run thorough performance tests after the migration.

Table 15-1. *Removed GC Combinations*

GC Configuration	Flags	Recommended Configuration
DefNew + CMS	`-XX:-UseParNewGC` `-XX:+UseConcMarkSweepGC`	ParNew + CMS
ParNew + SerialOld	`-XX:+UseParNewGC`	ParallelScavenge + SerialOld
ParNew + iCMS	`-Xincgc`	CMS
ParNew + iCMS	`-XX:+CMSIncrementalMode` `-XX:+UseConcMarkSweepGC`	CMS

DefNew + iCMS	`-XX:+CMSIncrementalMode`
	`-XX:+UseConcMarkSweepGC`
	`-XX:-UseParNewGC`
		CMS

CMS foreground	`-XX:+UseCMSCompactAtFullCollection`	G1 or CMS
CMS foreground	`-XX:+CMSFullGCsBeforeCompaction`	G1 or CMS
CMS foreground	`-XX:+UseCMSCollectionPassing`	G1 or CMS

Making G1 the Default Garbage Collector

In Java 9, G1 becomes the default garbage collector on 32-bit and 64-bit server configurations. As a low-pause collector, G1 should provide a better overall performance than a throughput-oriented collector such as the Parallel GC, which was the default GC in Java 8.

Deprecating the Concurrent Mark Sweep (CMS) Garbage Collector

Now that G1 is the default garbage collector, the Concurrent Mark Sweep (CMS) garbage collector is deprecated and the support for it will be dropped in a future major release. JVM issues a warning message when the command-line option `-XX:+UseConcMarkSweepGC` is used.

Removing Launch-Time JRE Version Selection

JDK 5 introduced the Multiple JRE (mJRE) that allowed developers to specify the required JRE version, or a range of versions, to launch an application. The version requirement could be specified in the manifest entry `JRE-Version` of the application's JAR file, or as a command-line option `-version:` to the `java` command. If the version of the current JRE didn't meet the requirement, it tried to search for a version that met the requirement and launched that version.

This mJRE feature sounds useful, but it is rarely used in practice. This feature has been removed in Java 9. The command-line option `-version:` of the `java` command has also been removed. If the manifest entry `JRE-Version` is found in a JAR file, the launcher emits a warning message and continues.

More Diagnostic Commands

More diagnostic commands have been added in Java 9 to run with `jcmd`. You can list all the available diagnostic commands using the `help` option of jcmd; see the following code. `6239` is the PID.

```
$ jcmd 6239 help
```

Listing 15-1 shows the list of available diagnostic commands in Java 8.

Listing 15-1. Diagnostic Commands in Java 8

```
JFR.stop
JFR.start
JFR.dump
JFR.check
VM.native_memory
VM.check_commercial_features
VM.unlock_commercial_features
ManagementAgent.stop
ManagementAgent.start_local
ManagementAgent.start
GC.rotate_log
Thread.print
GC.class_stats
GC.class_histogram
GC.heap_dump
GC.run_finalization
GC.run
VM.uptime
VM.flags
VM.system_properties
VM.command_line
VM.version
```

Listing 15-2 shows the list of available diagnostic commands in Java 9.

Listing 15-2. Diagnostic Commands in Java 9

```
JFR.configure
JFR.stop
JFR.start
JFR.dump
JFR.check
VM.log
VM.native_memory
ManagementAgent.status
ManagementAgent.stop
```

```
ManagementAgent.start_local
ManagementAgent.start
Compiler.directives_clear
Compiler.directives_remove
Compiler.directives_add
Compiler.directives_print
VM.print_touched_methods
Compiler.codecache
Compiler.codelist
Compiler.queue
VM.classloader_stats
Thread.print
JVMTI.data_dump
JVMTI.agent_load
VM.stringtable
VM.symboltable
VM.class_hierarchy
GC.class_stats
GC.class_histogram
GC.heap_dump
GC.finalizer_info
GC.heap_info
GC.run_finalization
GC.run
VM.info
VM.uptime
VM.dynlibs
VM.set_flag
VM.flags
VM.system_properties
VM.command_line
VM.version
```

When you compare the lists in Listing 15-1 and Listing 15-2, you can see that many diagnostic commands have been added. For each command, you can use the following `help` command to view help information.

```
$ jcmd 6239 help VM.info
```

The following are brief descriptions of these new commands:

- `JFR.configure`: Configures the Java Flight Recorder
- `VM.log`: Configures the unified logging system
- `VM.classloader_stats`: Displays statistics of all the class loaders
- `VM.stringtable`: Displays statistics of JVM's StringTable
- `VM.symboltable`: Displays statistics of JVM's SymbolTable
- `VM.print_touched_methods`: Displays all methods that have ever been touched during the lifetime of this JVM; requires the option `-XX:+LogTouchedMethods`
- `VM.class_hierarchy`: Displays the hierarchy of all loaded classes

- `VM.info`: Displays information about the JVM environment and status
- `VM.dynlibs`: Displays loaded dynamic libraries
- `VM.set_flag`: Sets the VM flag
- `ManagementAgent.status`: Displays the status of the management agent
- `Compiler.directives-*`: Manages the compiler directives
- `Compiler.codecache`: Displays the compiler code cache layout and bounds
- `Compiler.codelist`: Displays all compiled methods in the code cache that are alive
- `Compiler.queue`: Displays the methods queued for compilation
- `JVMTI.data_dump`: Signals the JVM to do a data dump
- `JVMTI.agent_load`: Loads a JVMTI native agent
- `GC.heap_info`: Displays Java heap information
- `GC.finalizer_info`: Displays information about the Java finalization queue.

Removal of the JVM TI hprof Agent

The `hprof` agent was added as a sample project for the JVM Tool Interface and was not intended to be a production tool. The agent has been removed in Java 9. If you want to use features provided by this agent, you should switch to other Java built-in tools or third-party solutions; see Table 15-2.

Table 15-2. *Alternatives to the `hprof` Agent*

Feature	Alternative Tool
Heap dumps	Diagnostic command `GC.heap_dump` or `jmap -dump`
Allocation profiler	VisualVM
CPU profiler	VisualVM or Flight Recorder

Removal of the jhat Tool

Introduced in JDK 6, `jhat` was an experimental and unsupported tool for heap visualization and analysis. This tool has been removed in JDK 9.

Removal of Demos and Samples

Although you may not notice, the JDK actually comes with some demos and samples. However, these demos and samples are outdated and unmaintained, so they have been removed in JDK 9.

Javadoc

The `javadoc` tool supports the generation of HTML5 markup in JDK 9. When the option `-html5` is present, `javadoc` generates HTML5 markup. HTML4 is still the default output type, but HTML5 will become the default in JDK 10. The generated HTML5 markup uses semantic structural HTML5 elements, including `header`, `footer`, and `nav`. It also implements the WAI-ARIA standard for accessibility.

A search box has been added to the documentation generated by `javadoc`. Users can use it to search for program elements and indexed terms and phrases. Names of modules, packages, types, and members are indexed and searchable. You can use the new tag `@index` to mark a term or phrase as searchable.

The generate documentations have been upgraded to include modules. For each module, the documentation shows its module dependencies and exported packages.

The tool `javadoc` now supports generating documentation for multiple modules. You can provide the source code of modules using the option `--module-source-path` and specify the modules to process using the option `--module`. Third-party libraries are specified using the option `-p`. In the module source directory, each subdirectory contains the source code of a module. The subdirectory name must be the module name; see Figure 15-1 for the source code of the sample application. This directory structure for the module source path is required, otherwise `javadoc` cannot find the modules.

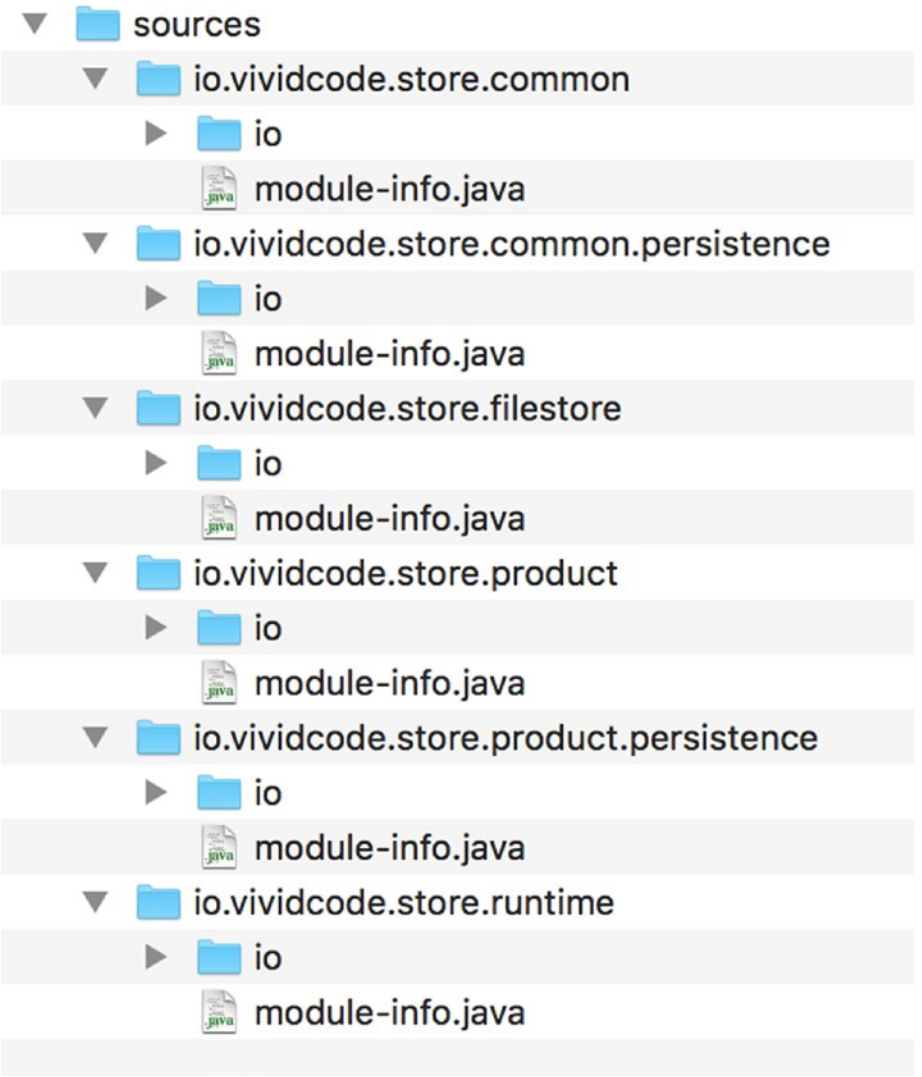

Figure 15-1. *Directory structure of Javadoc module source path*

For example, you can use the following command to generate documentation for the example project. The option `-d` specifies the output directory, while the option `-link` links documentations of the Java built-in API.

```
$ javadoc --module-source-path <sources_dir> \
    -p <lib_dir> \
    -d <output_dir>
    --module <modules> \
    -link https://docs.oracle.com/javase/9/docs/api/
```

Because the sample application is a Maven project, you need to use an extra task to copy the source code of all modules into one directory for Javadoc generation. You do this by using an Ant task. You can check the source code to see how it's implemented.

Summary

In this chapter, we discussed changes related to the JVM, including the unified logging system, removed GC combinations, diagnostic commands, javadoc, and other small changes. In the next chapter, we'll discuss several other small changes in Java 9.

CHAPTER 16

Miscellaneous

This chapter covers various changes in Java 9.

Small Language Changes

Java 9 has added some small language changes.

Private Interface Methods

It's now possible to add private methods to interfaces in Java 9. The interface `SayHi` in Listing 16-1 has the private method `buildMessage()` to generate the default message to use in the default method `sayHi()`.

Listing 16-1. Private Interface Methods

```
public interface SayHi {
  private String buildMessage() {
    return "Hello";
  }

  void sayHi(final String message);

  default void sayHi() {
    sayHi(buildMessage());
  }
}
```

Resource References in try-with-resources

The *try-with-resources* statement introduced in Java 7 is a small yet useful improvement in the Java language. In Java 9, it's been improved again to allow you to use *effectively-final* variables, so you no longer need to declare fresh variables for resources managed by this statement.

In Listing 16-2, the variable `inputStream` is *effectively-final*, so it can be used in the *try-with-resources* statement. The same code cannot compile in Java 8.

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6_16

Listing 16-2. try-with-resources Statement with effectively-final Variables

```
public void process() throws IOException {
  InputStream inputStream = Files.newInputStream(Paths.get("test.txt"));
  try (inputStream) {
    inputStream.readAllBytes();
  }
}
```

Other Changes

There are a few other changes in Java 9:

- The underscore (_) cannot be used as an identifier. Doing so now causes a compile error.
- `@SafeVarargs` can now be used on private instance methods.

The Stack-Walking API

`java.lang.StackWalker` is the new class in Java 9 to traverse stack traces that supports filtering and lazy access.

You start by creating new instances of `StackWalker` using the static method `getInstance()`. When using `getInstance()`, you can pass one or more options defined in the enum `StackWalker.Option`; see Table 16-1. You can also pass an estimate depth of the traverse.

Table 16-1. *Options of `getInstance()`*

Option	Description
`RETAIN_CLASS_REFERENCE`	Retains the `Class` object in the `StackFrame`
`SHOW_HIDDEN_FRAMES`	Shows all hidden frames
`SHOW_REFLECT_FRAMES`	Shows all reflection frames

After you get an instance of `StackWalker`, you can traverse the stack frames of the current thread using the method `<T> T walk(Function<? super Stream<StackWalker.StackFrame>,? extends T> function)`. The only parameter is a function that applies to a stream of `StackWalker.StackFrames` and returns a value. When the method `walk()` returns, the stream is closed and cannot be used again. `StackWalker.StackFrame` has different methods for retrieving information about the stack frame; see Table 16-2.

Table 16-2. *Methods of* `StackWalker.StackFrame`

Method	Description
`String getClassName()`	Returns the binary name of the declaring class of the method
`String getMethodName()`	Returns the name of the method
`Class<?> getDeclaringClass()`	Returns the declaring `Class` of the method
`int getByteCodeIndex()`	Returns the index to the code array of the `Code` attribute containing the execution point
`String getFileName()`	Returns the name of the source file
`int getLineNumber()`	Returns the line number of the source line
`boolean isNativeMethod()`	Returns `true` if the method is native
`StackTraceElement toStackTraceElement()`	Returns the `StackTraceElement`

If the option `RETAIN_CLASS_REFERENCE` is not passed when you create the `StackWalker`, the method `getDeclaringClass()` throws `UnsupportedOperationException`.

In Listing 16-3, I create a `StackWalker` with the option `RETAIN_CLASS_REFERENCE`, so I can use the method `getDeclaringClass()` to get the `Class` object. Here I use `getClassName()` to get the class name and collect them into a `Set`. I verify that the set should contain the current class `StackWalkingTest`.

Listing 16-3. Example of `walk()` in `StackWalker`

```
public class StackWalkingTest {

  @Test
  public void testWalkClass() throws Exception {
    final StackWalker stackWalker =
        StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE);
    final Set<String> classNames = stackWalker.walk(stream ->
        stream.map(StackWalker.StackFrame::getClassName)
            .collect(Collectors.toSet())
    );
    assertTrue(classNames.contains("StackWalkingTest"));
  }
}
```

If I just need to iterate all the `StackFrames` and perform actions on each of them, the method `void forEach(Consumer<? super StackWalker.StackFrame> action)` is easier to use than `walk()`; see Listing 16-4.

Listing 16-4. Example of `forEach()` in `StackWalker`

```
@Test
public void testForEach() throws Exception {
  final StackWalker stackWalker =
      StackWalker.getInstance(Set.of(Option.SHOW_HIDDEN_FRAMES,
          Option.SHOW_REFLECT_FRAMES));
```

```
  stackWalker.forEach(stackFrame -> System.out.printf("%6d| %s -> %s %n",
      stackFrame.getLineNumber(),
      stackFrame.getClassName(),
      stackFrame.getMethodName()));
}
```

`StackWalker` also has the method `getCallerClass()` that returns the `Class` object of the caller who invoked the method that invoked `getCallerClass()`. The description of `getCallerClass()` seems quite complicated and hard to understand. This method is actually very useful for utility methods that deal with classes, class loaders, and resources. Those utility methods should use the caller's class to perform actions.

`LoggerManager` in Listing 16-5 creates new `System.Logger` instances. The names of these loggers are based on the caller class of the method `getLogger()`. The option `RETAIN_CLASS_REFERENCE` must be configured for `getCallerClass()` to work. `getCallerClass()` always ignores reflection frames and hidden frames.

Listing 16-5. `LoggerManager`

```
public class LoggerManager {

  private final static StackWalker walker =
      StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE);

  public static System.Logger getLogger() {
    final Class<?> caller = walker.getCallerClass();
    return System.getLogger(
        String.format("%s/%s", caller.getModule().getName(), caller.getName()));
  }
}
```

In Listing 16-6, I create a new logger using `LoggerManager.getLogger()` and verify its name. Because the test class is in the unnamed module, the module name is `null`. `StackWalkingTest` is the class name. The logger name is `null/StackWalkingTest`.

Listing 16-6. Test `LoggerManager`

```
@Test
public void testGetCallerClass() throws Exception {
  final System.Logger logger = LoggerManager.getLogger();
  assertEquals("null/StackWalkingTest", logger.getName());
  logger.log(Level.INFO, "Hello World");
}
```

Objects

In Java 9, more methods have been added to `java.util.Objects`:

- `T requireNonNullElse(T obj, T defaultObj)`: Returns the first argument if it's non-null, otherwise returns the second non-null argument.
- `T requireNonNullElseGet(T obj, Supplier<? extends T> supplier)`: Returns the first argument if it's non-null, otherwise uses the `Supplier` to get the non-null value.

- `int checkIndex(int index, int length)`: Checks if `index` is within the bounds of range from `0` (inclusive) to `length` (exclusive). It can be used to check for array access. This method returns `index` if it's in the range, otherwise the method throws `IndexOutOfBoundsException`.
- `int checkFromIndexSize(int fromIndex, int size, int length)`: Checks if the subrange from `fromIndex` (inclusive) to `fromIndex + size` (exclusive) is within the bounds of the range from `0` (inclusive) to `length` (exclusive). It can be used to check for subarray access. This method returns `fromIndex` if the subrange is in the range, otherwise the method throws `IndexOutOfBoundsException`.
- `int checkFromToIndex(int fromIndex, int toIndex, int length)`: Checks if the subrange from `fromIndex` (inclusive) to `toIndex` (exclusive) is within the bounds of the range from `0` (inclusive) to `length` (exclusive). This method returns `fromIndex` if the subrange is in the range, otherwise the method throws `IndexOutOfBoundsException`.

Listing 16-7 shows some examples of these new methods.

Listing 16-7. Examples of Methods in `Objects`

```
public class TestObjects {

  @Test
  public void testRequireNonNullElse() throws Exception {
    assertEquals("hello", Objects.requireNonNullElse("hello", "world"));
    assertEquals("world", Objects.requireNonNullElse(null, "world"));
  }

  @Test(expected = IndexOutOfBoundsException.class)
  public void testCheckIndex() throws Exception {
    assertEquals(0, Objects.checkIndex(0, 1));
    Objects.checkIndex(3, 1);
    assertEquals(1, Objects.checkFromIndexSize(0, 2, 5));
    Objects.checkFromIndexSize(0, 3, 1);
    assertEquals(1, Objects.checkFromToIndex(1, 3, 5));
    Objects.checkFromToIndex(0, 3, 2);
  }
}
```

Unicode 8.0

Java 9 has upgraded its support of Unicode to 8.0 (www.unicode.org/versions/Unicode8.0.0/), which is two major versions upgraded from Unicode 6.2 in Java 8. In Unicode 8.0, you can use emojis (www.unicode.org/emoji/). This upgrade also brings better text display for right-to-left languages, such as Arabic and Hebrew.

UTF-8 Property Resource Bundles

The resource bundles in the Java platform have been using the ISO-8859-1 encoding for properties files for a long time. Properties files need to be converted using the tool `native2ascii` (https://docs.oracle.com/javase/8/docs/technotes/tools/unix/native2ascii.html) to escape Unicode characters. It's impossible to directly edit these properties files.

In Java 9, the default file encoding for `ResourceBundle` to load properties files has changed from ISO-8859-1 to UTF-8. Applications can use UTF-8 properties files directly without conversion. In Listing 16-8, I use `ResourceBundle` to load a properties file with Chinese characters.

Listing 16-8. UTF-8 Resource Bundles

```
public class UTF8ResourceBundleTest {

  @Test
  public void testGetProperty() {
    final ResourceBundle resourceBundle = ResourceBundle.getBundle("demo");
    assertEquals("你好", resourceBundle.getString("greeting"));
  }
}
```

You can override the default resource bundle encoding using the system property `java.util.PropertyResourceBundle.encoding`. The possible encodings are `ISO-8859-1` and `UTF-8`.

Enhanced Deprecation

The annotation `@Deprecated` marks a program element as deprecated and as one that should not be used. `@Deprecated` is enhanced in Java 9 with the addition of two new attributes. The attribute `since` indicates the version in which the annotated element became deprecated. The version string has the same format as the javadoc tag `@since`. The default value of `since` is an empty string. The attribute `forRemoval` indicates whether the annotated element is subject to removal in a future version. The default value is `false`. If the attribute `forRemoval` is `false`, the annotated element will still exist, at least in the next major release. If the attribute `forRemoval` is `true`, the future version in which the annotated element will be removed is unclear. It could be the next major release, or even after several other major releases. `@Deprecated` and the javadoc tag `@deprecated` should always be present at the same time on deprecated elements.

With these two new attributes, it's easier for developers to know which actions they should take when they're using deprecated elements. If the `forRemoval` is `true` for a deprecated element, then they need to update code that uses this element. Otherwise, the code will break when the element is removed in a future version.

In Java 9, `@Deprecated` annotations in many places have been updated to include these two attributes. For example, in the class `java.lang.Thread`, the methods `suspend()`, `resume()`, and `stop()` are marked as `@Deprecated(since="1.2")`, which means these methods are deprecated since Java 1.2, but that they will still exist, at least in next major release. The methods `countStackFrames()`, `destroy()`, and `stop(Throwable obj)` are marked as `@Deprecated(since="1.2", forRemoval=true)`, so these methods will be removed in a future version.

When deprecated elements are used, the compiler issues warnings. With the introduction of `forRemoval`, there are now two types of deprecation warning. The ordinary deprecation warnings are for `@Deprecated` with `forRemoval` set to `false`; the terminal deprecation warnings are for `@Deprecated` with `forRemoval` set to `true`. Before Java 9, you could use `@SuppressWarnings("deprecation")` to suppress the deprecation

warnings. In Java 9, `@SuppressWarnings("deprecation")` can only suppress ordinary deprecation warnings. To suppress terminal deprecation warnings, you need to use `@SuppressWarnings("removal")`.

You can use the tool `jdeprscan` to scan the usages of deprecated APIs. `jdeprscan` supports scanning a directory that is the root of a package hierarchy, a JAR file and a class file. By using the option `--for-removal`, you can limit the scanning results to be with APIs that are deprecated for removal. `jdeprscan -l` can print out the set of deprecated APIs.

The following command scans the JAR file of Guava:

```
$ jdeprscan <path>/guava-21.0.jar
```

Listing 16-9 shows the scanning result.

Listing 16-9. `jdeprscan` Scanning Result of Guava

```
class com/google/common/io/FileBackedOutputStream$1 overrides
  deprecated method java/lang/Object::finalize()V
class com/google/common/reflect/Element uses deprecated
  method java/lang/reflect/AccessibleObject::isAccessible()Z
class com/google/common/reflect/Element overrides deprecated
  method java/lang/reflect/AccessibleObject::isAccessible()Z
class com/google/common/util/concurrent/AtomicDoubleArray uses deprecated method
  java/util/concurrent/atomic/AtomicLongArray::weakCompareAndSet(IJJ)Z
```

NetworkInterface

The class `java.net.NetworkInterface` adds three new methods related to network interfaces:

- `Stream<NetworkInterface> networkInterfaces()`: Static; returns a stream of `NetworkInterfaces`
- `Stream<InetAddress> inetAddresses()`: Returns a stream of `InetAddresses` bound to this network interface
- `Stream<NetworkInterface> subInterfaces()`: Returns a stream of subinterfaces that attached to this network interface

Listing 16-10 shows how to use these three new methods. It displays information of all network interfaces.

Listing 16-10. Usage of New Methods in NetworkInterface

```
public class NetworkInterfaceDemo {

  public static void main(final String[] args) throws SocketException {
    final String allInterfaces = NetworkInterface.networkInterfaces().map(
        networkInterface ->
            String.format("Name:%s%nHost addresses:%s%nSub-interfaces:%s",
                networkInterface.getDisplayName(),
                networkInterface.inetAddresses().map(InetAddress::getHostAddress)
                    .collect(Collectors.joining(", ")),
                networkInterface.subInterfaces().map(NetworkInterface::getName)
                    .collect(Collectors.joining(", ")))
```

```
    ).collect(Collectors.joining("\n\n"));
    System.out.println(allInterfaces);
  }
}
```

Summary

As the last chapter of this book, this chapter covered changes in Java 9 that are too small to be in their own chapters, including small language changes, the stack-walking API, UTF-8 property resource bundles, enhanced deprecation, and other small changes.

Index

A

Abstract syntax trees (ASTs), 128
Migration in action
  biojava, 52
  building, project using Java 9, 51
  migration path, 51
add(), jshell, 59
Apache Maven, 4
Process API
  managing long running processes, 78
  printProcessInfo(), 76
  ProcessHandle.Info, 76
  ProcessHandle methods, 75
  ProcessHandle static method, 76
  process methods, 77
AppForegroundListener, 143
AppHiddenListener, 143
Applet API, 150
Appletviewer tool, 150
Application events, 143
AppReopenedListener, 143
arrayConstructor, enhanced method handles(), 109
arrayLength(), enhanced method handles(), 109
Array methods
  compare(), 69
  equals(), 69
  mismatch(), 68
Artifacts module
  JAR file, 46–47
  JDK modules, 49
Async(), concurrency, 119
Atomic classes, concurrency, 122
Atomic update methods, 103
Automatic module names, java API, 43
Automatic modules, 16

B

BinaryAndOctalLiterals(), ECMAScript 6, 126
Bitwise atomic update methods, 104
Break encapsulation, JDK tool, 19
ByteBuffer, access mode, 106

C

Chronounit, concurrency, 120
CircleCI, 6
Class, java API, 42
Class loaders, java API, 39
CLOSE_ALL_WINDOWS, 146
CMS, *See* Concurrent Mark Sweep (CMS)
Collection methods
  List.of(), 67
  Map.of() and Map.ofEntries(), 68
  Set.of(), 67
Collector methods
  filtering(), 71
  flatMapping(), 72
Compare(), array, 69
Concurrency
  Async(), 119
  atomic classes, 122
  chronounit, 120
  queues, 121
  Thread.onSpinWait(), 123
  Timeout(), 119
  TimeUnit, 120
  utilities method, 120
Concurrent Mark Sweep (CMS), 154
Configuration, java API, 31
Core interfaces
  Flow.Processor<T,R>, 88
  Flow.Publisher<T>, 87
  Flow.Subscriber<T>, 87
  Flow.Subscription, 88
counterLoop(), enhanced method handles, 113

F. Cheng, *Exploring Java 9*, https://doi.org/10.1007/978-1-4842-3330-6

D

Declaration module
  opening modules, packages, 14
  qualified exports, 14
  requires and exports, 9
  services, 12
  static dependencies, 12
  transitive dependencies, 9
DefaultLoggerFinder, 82
Deprecation, miscellaneous, 166
Desktop applications
  about window, 144
  application events, 143
  application exit, 146
  functionalities, 148
  open files, 145
  open URI, 146
  preferences window, 144
  print files, 146
Deterministic random bit generator (DRBG)
    algorithm, 139
Diagnostic contains method, parsing, 128
Diagnostic commands, JVM, 155
Docker, 5
doWhileLoop(), enhanced method handles, 115
DRBG algorithm, 140
DrbgParameters, 140
dropWhile(), stream, 70

E

Eclipse Java 9 support (BETA), 2
ECMAScript 6
  BinaryAndOctalLiterals(), 126
  function (), 127
  iterators and for..of loops, 126
  TemplateString(), 126
Empty(), enhanced method handles, 111
Enhanced method handles
  arrayConstructor(), 109
  arrayLength(), 109
  empty(), 111
  loops(), 111
  try finally(), 116
  varHandleExactInvoker, 110
  varHandleInvoker, 110
  zero(), 110
Equals(), array, 69

F

filtering(), collector, 71
Filter input streams, 135
findStaticVarHandle, 99
findVarHandle, 99
flatMapping(), collector, 72
Flow.Processor<T,R>, 88
Flow.Publisher<T>, 87
Flow.Subscriber<T>, 87
Flow.Subscription, 88
ForkJoinPool.commonPool(), 89
for..of loops, ECMAScript 6, 126
Function (), ECMAScript 6, 127–128
FunctionLengthAnalyzer(), Nashorn, 130

G

Garbage collector (GC)
  CMS, 154
  default G1, 154
  removed combinations, 154
getInstance(), Stack Walker API, 162
getMaxBufferCapacity(), 89
getPlatformClassLoader(), 40
Gradle, 4

H

handleQuitRequestWith(), 146
hprof agent, JVM, 157

I

ifPresentOrElse(), optional, 73
Input stream, 133
Installation
  Eclipse, 2
  JDK 9, 1
Integrated development
    environment (IDE), 2
Intellij IDE, 2
I/O
  input stream, 133
  ObjectInputstream Filter, 134
IteratedLoop(), enhanced method handles, 114
iterate(), stream, 71
Iterators, ECMAScript 6, 126

J

JAR file, artifacts module, 46–47
java, JDK tools, 24
Java 8, diagnostic commands, 155
Java 9
  CircleCI, 6
  Docker, 5
  IDE, 2
  installation, 1
  tools, 4

Java API
automatic module names, 43
class, 42
class loaders, 39
configuration, 31
ModuleDescriptor, 28
ModuleFinder, 27
module layers, 34
ModuleReader methods, 30
ModuleReference, 27
readability graph, 32
reflection, 43
ResolvedModule methods, 32
Javac
Docker, 5
JDK tools, 20
javadoc tool, JVM, 157
java.logging module, 82–83
Java virtual machine (JVM)
Concurrent Mark Sweep, 154
default GC, 154
diagnostic commands, 155
GC combinations, 154
hprof agent, 157
javadoc tool, 157
mJRE, 154
tool interface, 157
unified logging system
logging configuration, 152
tags, levels, decorations, output, 151
diagnostic command VM.log, 153
jdeps, JDK tools, 24
JDK modules, artifacts module, 49
JDK tools
break encapsulation, 19
java, 24
javac, 20
jdeps, 24
jlink, 21
observable module, 19
root module, 18
upgrading module path, 19
module paths, 17
module version, 18
jhat tool, JVM, 157
jlink, JDK tools, 21
jshell
/!, 65
add(), 59
classes, 58
/drop, 61
/edit, 60
/env, 63
/exit, 65
/help, 59
/history, 63
/imports, 62
$1.length(), 57
/list, 59
/<id>, 65
/-<n>, 65
/methods, 63
/open, 61
/reload, 64
/reset, 64
/save, 61
/set, 64
System.getP, 58
/types, 62
var, 57
/vars, 62

K

Keystore, 141

L

Language changes, miscellaneous
changes, 162
private interface method, 161
try-with-resources statement, 161
List.of(), collection, 67
LoggerFinder, 81
LoggerManager, 164
Logging API and service
DefaultLoggerFinder, 82
LoggerFinder, 81
SLF4J, 81
SLF4JLoggerFinder, 83
SLF4JLoggerWrapper, 83
System.getLogger(), 86
System.LoggerFinder, 85
System.Logger method, 81
Logging configuration, 152
Logging messages, 151
Log messages, 86
Loop(), enhanced method
handles, 111

M

Managing long running processes, 78
Map.of() and Map.ofEntries(), collection, 68
Memory Fence method, 107
Memory order, variable access mode, 100
Miscellaneous
deprecation, 166
language changes
other changes, 162
private interface method, 161
try-with-resources statement, 161

Miscellaneous (*cont.*)
network interface, 167
objects, 164
Stack Walker API, 162
Unicode 8.0, 165
UTF-8 resource bundles, 166
Mismatch(), array, 68
ModuleDescriptor, 28
ModuleFinder, java API, 27
Module layers, java API, 34
Module paths, 17
ModuleReader methods, java API, 30
ModuleReference, java API, 27
Module system
migration in action
biojava, 52
building the project, Java 9, 51
migration path, 51
application, 8
artifacts module
JAR file, 46–47
JDK modules, 49
common issues, 49
declaration
opening modules, packages, 14
qualified exports, 14
requires and exports, 9
services, 12
static dependencies, 12
transitive dependencies, 9
defined, 8
java API
automatic module names, 43
class, 42
class loaders, 39
configuration, 31
ModuleDescriptor, 28
ModuleFinder, 27
module layers, 34
ModuleReader methods, 30
ModuleReference, 27
readability graph, 32
reflection, 43
ResolvedModule methods, 32
JDK tools
break encapsulation, 19
common modules, 17
observable module, 19
root module, 18
upgrading module path, 19
module paths, 17
module version, 18
JPMS, 7
working with exsting code
automatic modules, 16
unnamed modules, 15
Module version, 18
Multiple JRE (mJRE), 154
Multiresolution images, user interface, 148

N

Nashorn
ECMAScript 6
BinaryAndOctalLiterals(), 126
function (), 127
iterators and for..of loops, 126
TemplateString(), 126
FunctionLengthAnalyzer(), 130
parser API
basic parsing, 128
parsing error, 129
ScriptEngine, 125
Network interface, miscellaneous, 167
NORMAL_EXIT, 146
Numeric atomic update methods, 104

O

ObjectInputFilter.Config, 135
ObjectInputFilter.FilterInfo method, 134
ObjectInputFilter.Status values, 134
ObjectInputstream Filter, 134
Objects, miscellaneous, 164
ofNullable(), stream, 69–70
Optional methods
ifPresentOrElse(), 73
Optional.or(), 73
stream(), 74
Optional.or(), optional, 73

P

Parser API, 128
basic parsing, 128
parsing error, 129
Parsing error, 129
PeriodicPublisher, 90
PKCS12, 141
Preferences window, 144
printProcessInfo(), 76
Private interface method, 161
ProcessHandle static method, 76
ProcessHandle.Info methods, 76
ProcessHandle method, 75
Process methods, 77

Q

Observable module, 19
Qeues, concurrency, 121
QuitHandler, 146

R

Reactive streams
    core interfaces, 87
    interoperability, 97
    Reactor, 96
    RxJava 2, 95
    SubmissionPublisher, 88
Reactor, 96
Readability graph, java API, 32
Read access mode, 101
ReadAllBytes(), input stream, 133
Read-Eval-Print Loop (REPL), 57
Reading and copying data, input stream, 133
ReadNBytes, input stream, 133
Reflection, java API, 43
ResolvedModule methods, java API, 32
Root module, 18
RxJava 2, 95

S

ScreenSleepListener, 143
ScriptEngine, Nashorn, 125
SecureRandom, 139
SecureRandomParameters, 140
Security
    PKCS12 keystore, 141
    SecureRandom, 139
    SHA-3 hash algorithms, 139
Set.of(), collection, 67
setOpenFileHandler(), 145
setOpenURIHandler(), 146
setPrintFileHandler(), 146
SHA-3 hash algorithms, 139
Signature polymorphic, access mode, 101
SLF4J, 81
SLF4JLoggerFinder, 83
SLF4JLoggerWrapper, 83
Stack Walker API, 162
StackWalker.StackFrame method, 163
Stream methods
    dropWhile(), 70
    iterate(), 71
    ofNullable(), 69
    takeWhile(), 70
    stream(), optional, 74
SubmissionPublisher method, 90
System.getLogger(), 86
System.getP, jshell, 58
System.LoggerFinder, 85
System.Logger.Level, 82
System.Logger method, 81
SystemSleepListener, 143

T

takeWhile(), stream, 70
TemplateString(), ECMAScript 6, 126
Thread.onSpinWait(), concurrency, 123
TIFF image format, 150
Timeout(), concurrency, 119
TimeUnit(), concurrency, 120
Tool interface, JVM, 157
TransferTo, input stream, 133
Try finally(), enhanced method handles, 116
Try-with-resources statement, 161

U

Unicode 8.0, miscellaneous, 165
Unified logging system
    decorations, 151
    levels, 151
    logging configuration, 152
    output, 151
    tags, 151
    diagnostic command VM.log, 153
Unnamed modules, 15
unreflectVarHandle, 100
Upgrading module path, 19
User interface
    Applet API, 150
    desktop applications
        about window, 144
        application events, 143
        application exit, 146
        functionalities, 148
        open files, 145
        open URI, 146
        preferences window, 144
        print files, 146
    multiresolution images, 148
    TIFF image format, 150
UserSessionListener, 143
UTF-8 resource bundles, miscellaneous, 166
Utilities method, concurrency, 120

V

VarHandle.AccessMode, 99, 101, 110
varHandleExactInvoker, 110
varHandleInvoker, 110
Variable handles
  VarHandle.AccessMode, 101
  arrays, 105
  atomic update method, 103
  bitwise atomic update, 104
  byte [], 106
  ByteBuffer, 106
  findStaticVarHandle, 99
  findVarHandle, 99
  Memory Fence, 107
  memory orderings, 100
  numeric atomic update, 104
  read access mode, 101
  signature polymorphic, 101
  unreflectVarHandle, 100
  write access mode, 102
Diagnostic command VM.log, 153

W, X, Y

whileLoop(), enhanced method handles, 115
Working with exsting code modules
  automatic modules, 16
  unnamed modules, 15
Write access mode, 102

Z

Zero(), enhanced method handles, 110