Java 9 Modularity Revealed

Project Jigsaw and Scalable Java Applications

Alexandru Jecan

Apress®

www.allitebooks.com

Java 9 Modularity Revealed

Project Jigsaw and Scalable Java Applications

Alexandru Jecan

Apress®

www.allitebooks.com

Java 9 Modularity Revealed

Alexandru Jecan
Munich, Germany

ISBN-13 (pbk): 978-1-4842-2712-1 ISBN-13 (electronic): 978-1-4842-2713-8
DOI 10.1007/978-1-4842-2713-8

Library of Congress Control Number: 2017954918

Copyright © 2017 by Alexandru Jecan

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Technical Reviewer: Josh Juneau
Coordinating Editor: Jill Balzano
Copy Editor: Corbin Collins

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/us/book/9781484227121. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

www.allitebooks.com

To my wife, Diana, who sustains me and encourages me every day in all my efforts.
To my parents, Alexandrina and Eugen, who provided me with a very good education since I was a young child. Thank you and I love you.

www.allitebooks.com

Contents at a Glance

www.allitebooks.com

Contents

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

About the Author

Alexandru Jecan is a senior software engineer, consultant, author, trainer, speaker and tech entrepreneur currently residing in Munich, Germany. He earned a degree in computer science from the Technical University of Cluj-Napoca, Romania.

Alexandru provides professional in-house trainings on various software technologies across Germany. His areas of specialization are: big data, data analysis, artificial intelligence, machine learning, back-end software development, front-end software development, database development, microservices and devops.

He speaks at tech conferences and user groups, both in Europe and the United States, on different topics related to software development and software technologies.

In his free time, Alexandru likes to read a lot and to spend time with his family. Alexandru is an avid reader, reading lots of books and magazines in the fields of information technology, economy, business and stock markets. He also reads Hacker News frequently and is delighted with how many extraordinary, outstanding and very smart people are on this planet. You can follow Alexandru on Twitter at @alexandrujecan, read his tech blog at `www.alexandrujecan.com`, or email him at `alexandrujecan@gmail.com`.

About the Technical Reviewer

Josh Juneau has been developing software and enterprise applications since the early days of Java EE. Application and database development have been his focus since the start of his career. He became an Oracle database administrator and adopted the PL/SQL language for performing administrative tasks and developing applications for the Oracle database. In an effort to build more complex solutions, he began to incorporate Java into his PL/SQL applications and later developed standalone and web applications with Java. Josh wrote his early Java web applications utilizing JDBC and servlets or JSP to work with backend databases. Later, he began to incorporate frameworks into his enterprise solutions, such as Java EE and JBoss Seam. Today, he primarily develops enterprise web solutions utilizing Java EE and other technologies. He also includes the use of alternative languages, such as Jython and Groovy, in some of his projects.

Over the years, Josh has dabbled in many different programming languages, including alternative languages for the JVM, in particular. In 2006, Josh began devoting time to the Jython Project as editor and publisher of the *Jython Monthly* newsletter. In late 2008, he began a podcast dedicated to the Jython programming language. Josh was the lead author for *The Definitive Guide to Jython*, *Oracle PL/SQL Recipes*, and *Java 7 Recipes*, and a solo author of *Java EE 7 Recipes* and *Introducing Java EE 7*—all published by Apress. He works as an application developer and system analyst at Fermi National Accelerator Laboratory and writes technical articles for Oracle and OTN. He was a member of the JSR 372 and JSR 378 expert groups and is an active member of the Java community, helping to lead the Chicago Java User Group's Adopt-a-JSR effort.

When not coding or writing, Josh enjoys spending time with his wonderful wife and five children, especially swimming, fishing, playing ball, and watching movies. To hear more from Josh, follow his blog at `http://jj-blogger.blogspot.com`. You can also follow him on Twitter at @javajuneau.

Acknowledgments

I would like to thank my family and my wife, Diana, for supporting me, encouraging me, and understanding me during the long nights and weekends spent on writing on this book.

I would like to thank my parents, Alexandrina and Eugen, for providing me with a very good education since I was a young child. Thank you for investing in my education and for providing me computer science and foreign languages courses since my first years . Thank you for raising me very well.

I would like to thank the entire team at Apress for their very good and professional work. Thanks to my coordinating editor, Jill Balzano, and my acquisitions editor, Jonathan Gennick, for trusting me and guiding me with precious advice and support through the difficult journey of writing this book. I also thank you for your patience and for encouraging me throughout the entire writing process. I thank my technical reviewer, Josh Juneau, for providing me with very helpful and useful reviews. Thank you, Apress team, for the excellent work!

—Alexandru Jecan

Introduction

The Java programming language, introduced in 1995, has had a very successful story. It's evolved constantly since its birth and became one of the most popular programming languages in the world. Every new release of Java has added new features—small, medium, and big.

Java 9 is finally here! It is scheduled for release in September 2017, more than three years after the release of Java 8.

Issues with the Monolithic JDK

The release of Java 8 in March 2014 brought very important features to the Java platform like Lambdas and the Stream API, which were definitely needed by developers. Nevertheless, some well-known weaknesses of the platform had still not been addressed in Java 8: the huge monolithic JDK and the class path. These problems are addressed in Java 9 by Project Jigsaw.

The most important feature of Java 9 is by far the new modular system it introduced. Other new features have been introduced in Java 9, but they're not the focus of this book. This book covers the new modular system introduced in Java 9. The big, monolithic, and indivisible JDK has been problematic for a long time. It's difficult to install it on small devices because many don't have enough memory to hold it. In many cases, a large number of classes that comprise the JDK aren't needed because the application may not require them. CORBA, for instance, is still part of the JDK, but it's rarely used in real projects today. It makes no sense to use the entire JDK when only a part or a small part of it is needed. The Compact Profiles introduced in Java 8 recognized the problems caused by the huge JDK and attempted to solve them, but only to a low degree. The three Compact Profiles still contain a lot of libraries that a developer may not actually need. There had to be a better way to split the entire JDK and to create a much smaller custom JDK as a runtime image that contains only the libraries needed and nothing more. Project Jigsaw is that way, as we'll see throughout this book.

Big, monolithic software applications present a series of disadvantages. Maintaining them is tough and expensive, and performing a small change can result into a great effort. In large projects, modularity is crucial because it allows easy maintenance of the source code by reducing its complexity due to the loose coupling mechanism it provides.

Issues with the Class Path

The problems related to class path have been in Java since the birth of the language. The Java Virtual Machine (JVM) doesn't know that a JAR on the class path depends on another JAR. It simply loads a group of JARs, but it doesn't check their dependencies. When a JAR is missing, the JVM breaks the execution at runtime. The JARs can't define concepts related to accessibility. They don't define accessibility constraints like public or private. The entire content of all JARs on the class path is completely visible to all the other JARs from the class path. There's no way to declare that some classes in a JAR are private. All classes and methods are public related to the class path, which leads to a problem sometimes called *JAR hell*. Conflicts between versions of JARs can arise, especially when two different versions of the same library are required.

Loading the classes from the class path is a slow process because the JVM doesn't know where exactly the class is located and therefore has to check all the existing files from the class path. Jigsaw addresses this pain point. By taking advantage of reliable configuration, module boundaries are enforced, and the JVM knows about the dependencies that are needed. This has a positive impact on performance. Java 9 defines the concept of *module path* and allows you to have a library as a JAR file on the class path or have the same library as a module on the module path. This means nobody is forced to turn all their libraries into modules when they switch to Java 9. The libraries can still be used on the class path, even in Java 9. This is a big advantage because Java 9 provides a smooth transition of libraries.

The module path introduced in Java 9 tends to solve the problems caused by the class path. It can replace the class path completely or can interact and work together with the class path.

Modularity is important because it provides a maintainable code base for the future. We should use modular programming when we want to separate the effort put into design, development, and testing. Modular programming speeds up development and makes debugging applications easier by reducing their complexity.

Overview of Chapters

The first chapter of this book describes the concepts that build the foundation of a modular application: high module cohesion, strong encapsulation, low module coupling and explicit interfaces. It also illustrates some of the most important principles of modular programming, like: continuity, understandability, reusability, combinability, and decomposability.

You may wonder why we shouldn't use OSGi instead of Jigsaw. The reason is that OSGi can't be used to modularize the JDK because it's built on top of the Java Development Kit Platform. Jigsaw isn't built *on top* of the platform but rather directly *into the core of it*. This allows Jigsaw to completely change the structure of the JDK. The main differences between Jigsaw and OSGi are described in Chapter 2.

Prior to JDK 9, there was no way to manage modules. That's where Project Jigsaw comes into action. It introduces a brand new modular system into the JDK and therefore allows applications to be built on the skeleton of a modular architecture. It brings flexibility, maintainability, security, and scalability to the Java platform. It introduces loosely coupled modules by clearly defining the dependencies between the modules.

Project Jigsaw groups the source code of the Java platform into modules and provides a new system to implement modules as part of the platform. It applies the modular system to the platform and to the JDK itself and offers programmers the possibility of writing programs using the modular system on top of the JDK.

The main goal of Project Jigsaw is to modularize the JDK and the runtime. A new component called *module* is introduced. Chapter 4 explains what a module is and how to define a module in Java 9. That chapter also examines how to declare a module's dependencies, how to encapsulate, and how to hide a module's internal implementation. You'll learn what an unnamed module, a named module, an automatic module, and an open module are. The chapter introduces the notion of module path and shows how to build a module graph with no cycles.

Regarding modules declarations, I show practical examples using each of the five clauses introduced in Jigsaw: the `requires` clause, the `exports` clause, the `uses` clause, the `provides` clause, and the `opens` clause.

The objective of the Java modular system is to provide *strong encapsulation* and *reliable configuration*. Chapter 2 explains what these notions mean and how they're achieved through the modular system in Java 9.

The class path is partially replaced by modules and the module path. The class path–specific well-known `ClassNotFound` exceptions are avoided on the module path because the compiler can now test, based on the modules' definitions, whether all the modules needed to run the application are available. If they're not found, it doesn't compile the application.

Accessibility is an important part of the entire ecosystem and is covered throughout this book. The integral concept regarding the manner in which accessibility is achieved fundamentally changes in Java 9. The public access identifier that we all know no longer means *accessible everywhere and to anyone*. Supplementary accessibility levels have been added in JDK 9 that extend the existing accessibility mechanism by defining accessibility at the module level. Concepts like *direct* and *implied readability*, prerequisites for defining accessibility, are also outlined in this book. You'll see how accessibility is imposed by the compiler, by the virtual machine, or by using core reflection.

We'll look at the new concept of *modular JAR files* and the great advantage it brings: the possibility of compiling a library with JDK 9+, using it on the module path for JDK 9+, and compiling it with JDK8 or earlier and using it on the class path. Because the class path can still be used, the migration of libraries to JDK 9 is smooth. Even if the libraries contain a module descriptor and are to be treated as "modules," they will still work on previous versions of JDK 9 because their module descriptor won't be taken into consideration on the class path. By using modular JARs, developers have the freedom to decide whether they want to switch to the module platform or not.

We'll highlight the distinctions between regular and modular JARs with some examples and describe the new format for files introduced in Java 9, called JMOD, which is very similar to the format of JAR files. We'll go over the new JMOD tool and describe its use in detail.

The compilation of multiple modules using the `--module-source-path` option is illustrated with some explanatory examples. We'll also describe the enhancements added to the jar tool and show how to use it to package everything as a modular JAR or as a JMOD file.

We'll see how to run the compiled classes and the `module-info.class` using the java launcher. New options like `--module-path` and `-m`, introduced in JDK9, are covered thoroughly. When attempting to run an application using the `-m` option, which tells the launcher where the main module is, a resolution is triggered. We'll describe in detail all the steps involved when running a Java modular application, including resolution triggering and generation of the module graph. We'll also look at special cases like when a module is missing and the startup fails, and we'll present some workarounds for this, such as using the newly introduced java launcher option `--show-module-resolution`.

We'll also put modular JARs on the class path and see how to successfully run them. This is very important: We'll explain how to mix the class path and the module path when running with the java launcher. For this, we'll take advantage of the newly introduced `--add-modules` command-line option.

Chapter 3 describes the JEP 200, called the Modular JDK. This is the JEP that splits the JDK into a set of modules. We'll consider the new structure of the JDK together with its modules. We'll talk about platform modules and show the module graph that represents the new modular structure of the JDK. We'll examine the graph, show how modules depend on each other, and learn how to list all the modules from the system using the `--list-modules` command-line option. We'll also explain the notions of standard module and non-standard module.

It's beyond the scope of this book to go through every module in detail. You can find a comprehensive list of all the existing modules on the Open JDK website at `http://cr.openjdk.java.net/~mr/jigsaw/ea/module-summary.html`.

The JEP 260, Encapsulate most internal APIs, is also covered in this book. In order to manage incompatibilities, all non-critical internal APIs were encapsulated by default. Besides that, all critical internal APIs for which supported replacements exist in JDK 8 were encapsulated. Other critical internal APIs weren't encapsulated. Since they were deprecated in JDK 9, a workaround via a command-line flag is provided.

A linker and a new phase called *link-time* were introduced in Java 9. This phase is optional and is executed after the compile-time but before the runtime. It basically assembles the modules in order to form a runtime image. Runtime images allow us to create custom JDKs that contain only the minimum number of modules necessary to run our application. The minimum possible runtime would contain a single module, the java.base module. Runtime images allow us to scale down the JDK or to scale it up based on our needs. They're a replacement of the runtime rt.jar.

The linker represents a new phase of the development life-cycle. It improves performance by selecting only the minimum modules that it needs to successfully compile the code and provides many optimization options for the future.

Jigsaw makes it possible to install separate modules as part of the JDK platform installation. It also allows us to dynamically include other additional modules in the JDK runtime image. We'll talk about the changes related to the binary structure of the JRE and the JDK and about the new structure of the legacy JDK image. You'll learn more about all these new concepts in Chapters 5 and 7.

In Chapter 6 you'll learn what *services* are, and we'll describe through examples the notions of service interface and service providers. We'll show how to define service providers in modules and how to make them available to other modules.

In Chapter 8 you'll see how to migrate applications and libraries to modules in a smooth way. We'll describe how to migrate an application to Java 9 using the top-down migration strategy. For this, we'll look at a concrete example of how to migrate an application that contains some third-party JARs to modules. We'll see which kind of applications present the risk of breaking when switching to JDK 9. We'll will give useful solutions to avoid this, such as searching the code for dependencies, avoiding splitting packages, and checking the usage of the internal APIs. If you've already switched to JDK 9, we recommend trying first to run your applications with the new JDK to see if it breaks your code. We'll cover the JDeps tool and how to use it to audit your code and search for JDK-internal APIs. We'll discuss the Maven JDeps plugin, which represents the integration of the JDeps tool with the build tool Maven. We'll also talk about the impact and consequences of the removal of rt.jar and tools.jar from the JRE.

Chapter 9 covers the new API introduced in JDK 9 for working with modules. We'll see how to perform basic operations on modules.

Chapter 10 gets into some advanced topics like layers, class loading mechanism in JDK 9, multi-release JAR files, the JMOD tool, and upgradeable modules. We'll describe the concept of layers, which are a group of class loaders used to load classes from a module graph. We'll look at the boot layer and the correlation to the so-called well-formed graphs.

Chapter 11 talks about how to handle different scenarios for testing modular applications. Three scenarios are covered: Junit test classes and objects under test residing in different modules, Junit test classes and objects under test residing in the same module, and only objects under test residing inside a module. We'll show how to patch a module and how to use Maven for easing testing.

In Chapter 12 you'll learn how Jigsaw interacts with build tools like Maven and what kind of support the most popular IDEs offer for Project Jigsaw.

As you can see, this book is structured into 12 chapters. Chapter 1 covers modular programming concepts. Chapters 2–9 provide you with a very strong foundation on Project Jigsaw. Chapter 10 describes some advanced features that will help you understand some complex topics on Jigsaw. Chapter 11 shows how to test modular applications using Junit. Chapter 12 teaches using Jigsaw together with build tools and integrated development environments (IDEs).

Each topic should be easy to find. We recommend reading the chapters sequentially in order to understand all the topics. Some examples build on concepts that were explained in previous chapters.

Who Should Read This Book

This book is intended for anyone who wants to get familiar with the new modularity system introduced in Java 9. It gives a strong foundation for anyone who wants to understand the core concepts as well as the advanced concepts of Java 9 modularity. The examples are designed to help you deeply understand all the notions introduced in Project Jigsaw. We've tried, as much as possible, to give a plenty of examples for most of the theoretical concepts discussed throughout the book.

Whether you already have experience with modular systems or not, this book is for you.

The book can't cover *everything* on Java 9 modularity. Java 9 modularity is a very large and complex subject, and covering every corner of it wouldn't be possible in a book of this size. However, the book goes through all the core parts of Java 9 modularity and touches some advanced topics. By reading this book, you'll not only understand the concepts behind Java 9 modularity, you'll also be able to apply them on your day-by-day projects.

We advise you to try the examples from this book by yourself in order to get familiar with Project Jigsaw.

What You Will Learn

This book aims to provide comprehensive information on the new modular system introduced in Java 9. A well-structured tutorial is conducted throughout the book by combining theoretical concepts with practical examples.

Learning to use modularity with Java 9 will help you enhance your technology career and give you very precious technical skills.

Once you read this book, you'll be able to develop scalable and modular Java 9 applications and migrate existing Java applications to Java 9.

Here are some of the most important things you'll learn in this book:

- What modularity is in general and what advantages it brings
- What Java 9 modularity is and what its goals are
- How the new layout for JDK and JRE looks
- What strong encapsulation and reliable configuration are and how to apply and take advantage of them
- Which JDK-internal APIs have been encapsulated in Java 9 and which have remained accessible
- How the JDK was divided into a group of modules
- What the new accessibility rules in JDK 9 are
- How to define a module together with its dependencies
- How to create a modular JAR file and a JMOD file
- How to compile, package, and run a modular Java application using Jigsaw
- How to use the JDeps tool to audit the code, search for dependencies between the libraries, or generate module descriptors
- How to migrate an application to the modular system
- How to solve migration issues like encapsulated JDK internal APIs, not resolved modules, split packages, cyclic dependencies, and more
- How to perform top-down migration
- How to define, configure, and use services for modules
- How to combine the module path and the class path to provide backward compatibility

- How to create custom modular runtime images using the Jlink linking tool
- How to define and use a layer in Java 9
- How to perform operations on a Module using the new Module API
- How to use qualified exports
- How to improve the maintainability and performance of Java applications
- How to handle the unit testing of a modular application
- How to patch a module
- How to integrate Jigsaw with different build tools like Maven
- How to check whether a Java application is compatible with JDK 9
- How to make a Java application compatible with JDK 9
- How to assure runtime compatibility when switching to Java 9
- How to choose the best design patterns when modularizing a Java application

Errata

Everybody involved in publishing this book is strongly committed to providing an error-free book. That's why the errata of this book is continuously updated as soon as even a minor issue is found. You can submit errata at www.apress.com/us/services/errata.

Contacting the Author

The source code for this book can be accessed by clicking the Download Source Code button on its apress.com product page, located at www.apress.com/9781484227121.

Downloading the Code

The source code from this book can be found on GitHub. You can also download it from the book's product page at www.apress.com/us/book/9781484227121.

CHAPTER 1

Modular Programming Concepts

You've almost surely had to deal with complexity in your software projects. The complexity level of a software application is usually low when development begins, but after a while the complexity begins to increase due to the changes performed on the platform. Complexity keeps increasing constantly as new features are added and the existing functionality is customized. The more changes and customizations are performed, the more complex the system becomes—it may get so complex that it becomes difficult for a new developer to ramp up the project and be able to understand all its inner workings. And if the documentation of the system software isn't good enough, then understanding the system becomes even harder. A high level of complexity requires more energy, resources, and time to be spent in order to understand the inner structure of the application.

What is it that causes software systems to become so difficult to maintain? The answer is related to the fact that there are a lot of existing dependencies throughout the code. That happens when a piece of code depends on many other pieces of code, and it can generate a lot of issues. Enhancing such a system becomes painful because making a change on one place may affect many other parts of the application. By modifying an application in many different areas, the risk of introducing new errors grows. Besides that, reaching a satisfactory level of reuse becomes very difficult. The software has so many dependencies that simply reusing a component can become costly in terms of time and it could also *further* increase complexity. This also hinders the desire to enhance the system. By having a system with many dependencies, adding new functionality becomes a nightmare. Furthermore, the testing process also becomes more difficult because testing separate components is almost impossible to achieve. For you as a developer, understanding every part of the system is hard due to its complexity. As new features are added on a regular basis, and the software system evolves, keeping up to date with the changes can be challenging. In order to mitigate and reduce the negative effects of rising complexity, maintaining the system is mandatory, although maintaining itself becomes demanding in terms of time, effort, and cost.

What do we need in order to get rid of these problems? The answer is modularity.

General Aspects of Modularity

Modularity specifies the interrelation and intercommunication between the parts that comprise a software system. *Modular programming* defines a concept called the module. *Modules* are software components that contain data and functions. Integrated with other modules, together they form a unitary software system. Modular programming provides a technique to decompose an entire system into independent software modules. Modularity plays a crucial role in modern software architecture. It divides a big software system into separate entities and helps reduce the complexity of software applications while simultaneously decreasing the development effort.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_1

The goal of modularity is to define new entities that are easy to understand and use. Modular programming is a style of developing software applications by splitting the functionality into different modules — software units that contain business logic and have the role of implementing a specific piece of functionality. Modularity enables a clear separation of concerns and assures specialization. It also hides the module's implementation details. Modularity is an important part of agile software development because it allows us to change or refactor modules without breaking other modules.

Two of the most important aspects of modularity are maintainability and reusability, both of which bring great benefits.

Maintainability

Maintainability refers to the degree to which a software system is upgraded or modified after delivery. A big, monolithic software application is hard to maintain, especially if it has many dependencies inside the code.

The architecture of the system and the design patterns used help us create maintainable code. Maintainability is often ensured by simplicity. For instance, one of the simplest ways to improve maintainability is to provide a reference only to the interface that is implemented by the class as a substitute of the class itself. Low maintainability is a consequence of technical debt. Duplicating code may sometimes decrease the level of maintainability. For example, if one piece of code is altered, then other pieces of code that are similar to it also require the same sort of modifications. Because the code is in are many locations, it's easy to omit some of the code segments that have to be modified, and this introduces new software issues into the system. The level of maintainability is associated with the quality of the software: the higher the degree of maintainability, the higher the quality of software. Maintainability is enhanced as a result of splitting a monolithic application into a set of modules that present well-defined boundaries between them. In a modular software application, changing a module is easier when it has fewer incoming and outgoing dependencies.

Reusability

Object-oriented programming can be used to obtain reusability, especially via inheritance. In order to reuse the functionality encapsulated in an object, a second object must inherit the first object.

How do modules relate to reusability? It should be possible to reuse a module elsewhere in the same application or in other application. *Reusability* is the degree to which we can reuse or replace a module. Reusability avoids duplicating code and reduces the number of lines of code, which has a positive impact on the number of software defects. It not only improves software quality, it also helps in developing software faster and makes performing updates on it easier. By applying reusability, the functionality is replicated in a coherent form throughout the entire software system.

Reusability makes the developer's job easy because it increases their productivity when developing software components. Modules can be reused because they implement a well-defined interface that makes communication with other modules possible. The interface, which is specified as a contract, allows modules to be exchanged. The module interface is expressed in a standard way so that it may be understood and recognized by other modules. In order to achieve a high degree of software reusability, a module should perform a well-defined function. A "design once, deploy many times" software architecture is realized by taking advantage of source code reusability. As a property of good software design, reusability is increased by reducing the dependencies between the modules.

Reusability plays an important role in the migration of applications and libraries. Migration becomes simpler when you can reuse software components or modules. Reusability is not easy to achieve because it is challenging to design software that must be successfully used to fit somewhere else.

Module Definition

A software module is an independent and deployable software component of a larger system that interacts with other modules and hides its inner implementation. It has an interface that allows inter-modular communication. The interface defines which components it provides for external use and which components it requires for internal use. A module determines a boundary by specifying which part of the source code is inside the module. It also provides flexibility and increases the reusability of the software system.

Modules can be discovered starting from compile-time. A module can expose some of its classes to outside or can encapsulate them in order to prevent external access. Figure 1-1 illustrates this concept with an example of a module containing classes that are exposed to outside (classes in green color) and classes that are not exposed to outside (classes in red color).

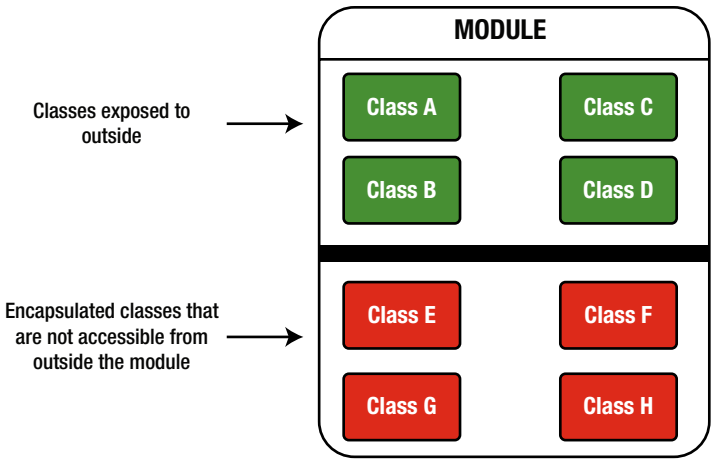

Figure 1-1. *A module specifies the non-encapsulated (green) and the encapsulated classes (red)*

A module can also be viewed as a *black box*. It has an input and an output and performs a specific function. It takes the input, applies some business logic on it, and returns an output, as illustrated in Figure 1-2.

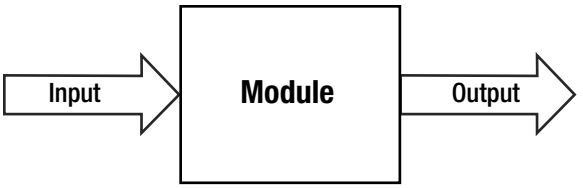

Figure 1-2. *A module seen as a black box*

A software module is reusable, testable, manageable, and deployable. Multiple modules can be combined together to form a new module. Modular programming is the key to reducing the number of bugs in complex software systems to a minimum. By dividing the application into very small modules, each modules will have fewer defects because its functionality is not complex. Assembling these less error-prone modules results in an application with fewer errors.

One of the key facets of modularity is breaking the application down into small, thin modules that are easy to implement because they don't possess a high level of complexity. The modules can be interconnected earlier at compile time or later at runtime. Each module must be able to be bound to the core application.

Figure 1-3 shows the general structure of a module.

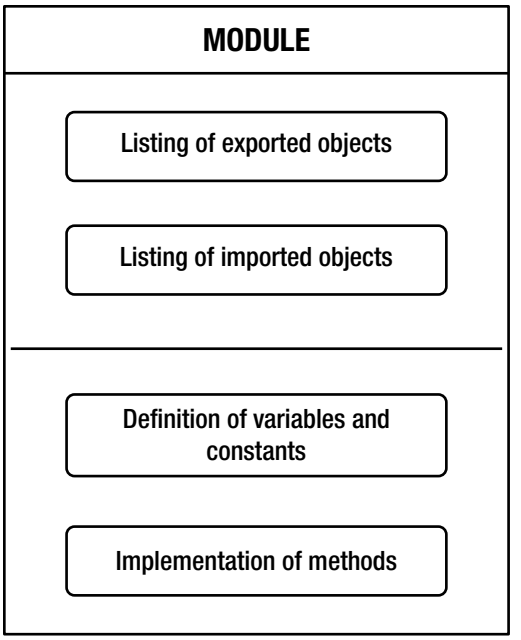

Figure 1-3. *General structure of a software module*

A module generally consists of two parts: the module interface and the module implementation. The *module interface* defines the objects that it exports and the objects that it imports. The *exported* objects are the objects that are suited to being available outside of the module. The *imported* objects are the objects that the module requires from outside for internal use. The *module implementation* defines the variables, constants, and implementation of methods.

Using a module as a variable, instance variable, constant, or function isn't allowed. A module can consist of objects that can be used only internally inside the module and objects that can be exported to the other modules for external use. *Data abstraction*, a core concept of modularity, is achieved by hiding information so that it won't be accessible from outside unless it's explicitly specified by an export. By default, the internal structure and internal implementation of a module are hidden from other modules.

A change performed on a specific module should not have an impact on other modules. Additionally, it should be possible to add a new module to the core system without breaking the system. Because only the interface of a module is visible from outside the module, it should be possible for developers to change the module's internal implementation without breaking the code in the application. The structure of a modular software application is basically defined by the connections and correlations between modules.

Some of the characteristics of a module include the following:

- A module must define interfaces for communication with other modules.
- A module defines a separation between the module interface and the module implementation.
- A module should present a set of properties that contain information.
- Two or more modules can be nested together.
- A module should have a clear, defined responsibility. Each function should be implemented by only one module.
- A module must be able to be tested independently from other modules.
- An error in a module should not propagate to other modules.

Let's give a short example. If we have two Jigsaw modules called A1 and A2 and one package in module A2 called P2 that we want to be accessible in module A1, then the following conditions have to be met:

- Module A1 should depend on module A2; module A1 should specify in its declaration that it "requires" module A2.
- Module A2 should export the package P2 in order to make it available to the modules that depend on it. In our case, in the declaration of the module A2 we should specify that it "exports" package P2.

Package P2 will be accessible in module A1 only if both of conditions are met at compile-time. If none or just one of the conditions mentioned above is met, then package P2 won't be accessible in module A1. That's part of the reliable configuration concept introduced in JDK 9. We cover reliable configuration later in this chapter and in upcoming chapters.

The following sections look at four concepts that build the foundation of a modular application:

- Strong encapsulation
- Explicit interfaces
- High module cohesion
- Low module coupling

Strong Encapsulation

Encapsulation defines the process of preventing data access from outside by allowing a component to declare which of its public types are available to other components and which aren't. Encapsulation improves code reusability and diminishes the number of software defects. It helps obtain modularity by decoupling the internal behavior of each object from the other elements of the software application.

Related to modularity, encapsulation designates a technique that hides the details of the module implementation. Only the important characteristics of a module should be visible and accessible from other modules. Source code in one module should be able to access a type in another module only if the first module reads the second module and at the same time the second module exports the package that encloses that corresponding type.

In Java prior to version 9, we took advantage of encapsulation by setting the variables and methods of classes to private. In this way, they were accessible only inside the class. We used to define accessor methods like setters and getters as public in order to allow the instance variables to be read or modified from outside of the class.

You will see in the following sections how we can achieve strong encapsulation in Java 9 using modules and the new types used for accessibility.

Explicit Interfaces

The interfaces of a modular system should be as small as possible. If an interface is too big, it should be divided into several smaller interfaces. An interface should make available to a module only the methods that the module really needs in order to be able to fulfill its business requirements.

A modular system typically provides module management and configuration management. *Module management* refers to the capacity to install, uninstall, and deploy a module. The installation could be done from a module repository, for example. In some cases, a module could be deployed instantly without requiring that the system is restarted. *Configuration management* specifies the capacity to dynamically configure modules and specify the dependencies between them.

High Module Cohesion

Cohesion measures how the elements of a module are residing together. *Module cohesion* denotes the module's integrality and coherence in regard to its internal structure. It expresses the rate to which the module's items are defining only one functionality.

The highest module cohesion is achieved when all the elements from the module are grouped together to form a piece of functionality. When designing a module, the focus should be on having a high level of cohesion, and this can be accomplished in many different ways:

- By reducing the complexity of the module (for instance by using fewer methods or less code)
- By reducing the complexity of the methods described in the module
- By using related groups of data
- By defining only one predefined scope for the module

Cohesion describes not only the capability of the module to act like a standalone component in the entire ecosystem, but also the homogeneity of its internal components. High cohesion provides better maintainability and reusability because loosely coupled source code can be altered more simply and with less pain than source code that isn't loosely coupled.

During the conception of modules, one significant aspect is choosing the right degree of complexity for them. If the functionality of a module is small, then the module might not be very helpful in the entire module ecosystem. If its functionality is complex and it performs a lot of tasks, then it might be troublesome to reuse it. It's a trade-off, and it's up to you to make the right decision.

Low Module Coupling

Coupling specifies the level of interdependence between modules. *Module coupling* refers to the dependency between modules and the way they interact. The objective is to reduce module coupling as much as possible, and this is achieved by specifying interfaces for the inter-modular communication. The interfaces have the role of hiding the module implementation. The resulting modules are independent and can be modified or swapped without fear of breaking other modules—or worse, breaking the entire application.

Low coupling usually corresponds to high cohesion. This is the result we want to achieve in the context of modularity. The reverse—high coupling and low cohesion—is the opposite of what a modular system should normally aim to accomplish.

Tight Coupling vs. Loose Coupling

Tight and loose coupling can refer to classes or modules. *Tight coupling* between classes is when a class uses logic from another class. It basically uses another class, instantiates an object of it, and then calls the object to access methods or instance variables.

Loose coupling is encountered when a class doesn't directly use an instance of another class but uses an intermediate layer that primarily defines the object to be injected. One framework that defines loose coupling is the Spring framework, where the dependency objects are being injected by the container into another object. Loosely coupled modules can be altered with less effort. Loose coupling is generally accomplished by using small or medium-sized modules. Replacing a module won't affect the system if the new module has the same interface as the module being replaced.

Tight coupling means classes are dependent on other classes and it doesn't allow a module to be replaced so easily because it has dependencies on the implementation of other modules.

Listing 1-1 shows an example of tight coupling. The listing defines one class called `Customer` that has dependencies on objects of types `CurrentAccount`, `DepositAccount`, and `SavingsAccount`. In the `Main` class, we create one object of type `Customer`. This object further creates three other objects. The `Customer` class contains an object of type `CurrentAccount` and calls the method `depositMoney(amount)` on this object. This is a tight coupling between class `Customer` and class `CurrentAccount`, and because class `CurrentAccount` is completely tied to class `Customer`, it depends on it. The class `Customer` creates objects of types `CurrentAccount`, `DepositAccount`, and `SavingsAccount` in order to execute some business logic that is defined in these three classes.

Listing 1-1. Defining Three Classes That Are Similar to Each Other

```
// CurrentAccount.java
package com.apress.tightcoupling;

public class CurrentAccount {

        long deposit;

        public void depositMoney(long amount) {
                deposit = amount;
        }

        public long getDeposit() {
                return deposit;
        }
}
```

```
// DepositAccount.java
package com.apress.tightcoupling;

public class DepositAccount {

        long deposit;

        public void depositMoney(long amount) {
                deposit = amount;
        }

        public long getDeposit() {
                return deposit;
        }
}

// SavingsAccount.java
package com.apress.tightcoupling;

public class SavingsAccount {

        long deposit;

        public void depositMoney(long amount) {
                deposit = amount;
        }

        public long getDeposit() {
                return deposit;
        }
}
```

Listing 1-2 defines a class called `Customer` that initializes objects of the classes `CurrentAccount`, `DepositAccount`, and `SavingsAccount`.

Listing 1-2. The Customer Class

```
// Customer.java
package com.apress.tightcoupling;

public class Customer {

        private CurrentAccount currentAccount;
        private DepositAccount depositAccount;
        private SavingsAccount savingsAccount;

        public Customer() {
                currentAccount = new CurrentAccount();
                depositAccount = new DepositAccount();
                savingsAccount = new SavingsAccount();
        }
```

```
        public void depositMoneyIntoCurrentAccount(long amount) {

                currentAccount.depositMoney(amount);
        }

        public void depositMoneyIntoDepositAccount(long amount) {

                depositAccount.depositMoney(amount);
        }

        public void depositMoneyIntoSavingsAccount(long amount) {

                savingsAccount.depositMoney(amount);
        }

        public CurrentAccount getCurrentAccount() {
                return currentAccount;
        }

        public DepositAccount getDepositAccount() {
                return depositAccount;
        }

        public SavingsAccount getSavingsAccount() {
                return savingsAccount;
        }
}
```

Listing 1-3 defines the `Main` class, which creates three objects of type `Customer` and calls methods on them.

Listing 1-3. The Main Class

```
// Main.java
package com.apress.tightcoupling;

public class Main {

    public static void main(String[] args) {

                Customer firstCustomer = new Customer();
                firstCustomer.depositMoneyIntoCurrentAccount(50);

                Customer secondCustomer = new Customer();
                secondCustomer.depositMoneyIntoDepositAccount(100);

                Customer thirdCustomer = new Customer();
                thirdCustomer.depositMoneyIntoSavingsAccount(200);

                System.out.println("First Customer current account amount: "
                                + firstCustomer.getCurrentAccount().getDeposit());
                System.out.println("Second Customer deposit account amount: "
                                + secondCustomer.getDepositAccount().getDeposit());
                System.out.println("Third Customer savings account amount: "
                                + thirdCustomer.getSavingsAccount().getDeposit());
    }
}
```


www.allitebooks.com

The previous three listings show tight coupling in which the `Customer` class instantiates objects of other classes and subsequently accesses methods on them. This results in a very high level of dependency between the `Customer` class and the other classes it uses. The main problem is that a change in `CurrentAccount`, `DepositAccount`, or `SavingsAccount` classes could eventually obligate us to adapt the class `Customer`. For example, if the constructor of `CurrentAccount` changes, we have a problem. To decouple the classes in this example, we should modify the code so that class `Customer` is not dependent any more on the implementation of classes `CurrentAccount`, `DepositAccount`, and `SavingsAccount`. As a result, we'll use an interface in order to make class `Customer` dependent only on the interface. And we'll instantiate the other dependencies only in the `Main` class and not in the `Customer` class anymore, as we saw before.

Listing 1-4 defines the interface `AccountInterface`, which contains the definitions of the methods.

Listing 1-4. The Interface AccountInterface

```
// AccountInterface.java
package com.apress.looseCoupling;

public interface AccountInterface {

        void depositMoney(long amount);

        long getDeposit();
}
```

Listing 1-5 defines the three classes that implement the interface and provide implementations for the methods from the interface.

Listing 1-5. The Classes CurrentAccount, DepositAccount, and SavingsAccount

```
// CurrentAccount.java
package com.apress.looseCoupling;

public class CurrentAccount implements AccountInterface {

        long deposit;

        public CurrentAccount() {
        }

        @Override
        public long getDeposit() {
                return deposit;
        }

        @Override
        public void depositMoney(long amount) {
                deposit = amount;
        }
}
```

```
// DepositAccount.java
package com.apress.looseCoupling;

public class DepositAccount implements AccountInterface {

        long deposit;

        public DepositAccount() {
        }

        @Override
        public long getDeposit() {
                return deposit;
        }

        @Override
        public void depositMoney(long amount) {
                deposit = amount;
        }
}

// SavingsAccount.java
package com.apress.looseCoupling;

public class SavingsAccount implements AccountInterface {

        long deposit;

        public SavingsAccount() {
        }

        @Override
        public long getDeposit() {
                return deposit;
        }

        @Override
        public void depositMoney(long amount) {
                deposit = amount;
        }
}
```

Listing 1-6 presents the `Customer` class, which uses the interface called `AccountInterface` inside of its constructor.

Listing 1-6. The Customer Class

```
// Customer.java
package com.apress.looseCoupling;

public class Customer {

        private AccountInterface account;

        public Customer(AccountInterface account) {
                this.account = account;
        }

        public void deposit(long amount) {
                account.depositMoney(amount);
        }

        public AccountInterface getAccount() {
                return account;
        }
}
```

Listing 1-7 shows the `Main` class, which creates three objects of type `Customer`.

Listing 1-7. The Main Class

```
// Main.java
package com.apress.looseCoupling;

public class Main {

    public static void main(String[] args) {

                CurrentAccount currentAccount = new CurrentAccount();
                Customer firstCustomer = new Customer(currentAccount);
                firstCustomer.deposit(10);

                DepositAccount depositAccount = new DepositAccount();
                Customer secondCustomer = new Customer(depositAccount);
                secondCustomer.deposit(100);

                SavingsAccount savingsAccount = new SavingsAccount();
                Customer thirdCustomer = new Customer(savingsAccount);
                thirdCustomer.deposit(200);
```

```
                System.out.println("First Customer current account amount: "
                                   + firstCustomer.getAccount().getDeposit());
                System.out.println("Second Customer deposit account amount: "
                                   + secondCustomer.getAccount().getDeposit());
                System.out.println("Third Customer savings account amount: "
                                   + thirdCustomer.getAccount().getDeposit());
    }
}
```

The `Customer` class is no longer dependent on the other classes. It doesn't create new classes of type `CurrentAccount`, `DepositAccount`, or `SavingsAccount`, but uses an interface in its constructor. The interface is implemented by all of the three classes. Because `CurrentAccount`, `DepositAccount`, and `SavingsAccount` implement the interface `AccountInterface`, they're injected into the `Customer` object. In the `Main` class, we create objects of type `interface` and then pass these objects to the constructor of the `Customer` class. At the end we call the method `deposit()` on the `Customer` objects, which further calls the method `depositMoney()` from the interface.

In this simple example we've seen how to switch from a tightly coupled application to a loosely coupled one by programming to an interface instead of programming to an implementation.

Modular Programming

This section discusses the principles and benefits of modular programming. It compares modular programming and object-oriented programming (OOP) and talks about the differences between a monolithic application and a modular one.

Principles of Modular Programming

The principles of modular programming are continuity, understandability, reusability, combinability, and decomposability. I've already talked about reusability, so now let's focus on the other four.

- *Continuity*: Refers to the situation when a requirement to change the functionality of the software system should cause changes in as few modules as possible.
- *Understandability*: Refers to the fact that each module should be comprehensible as a standalone single unit. Its role should be clear and concise. It's definitely easier to understand the inner workings of a particular module—which presents a lower level of functionality—than of an entire application. You should avoid situations in which a module fulfills its role only in correlation with some other modules. A module should not cause side issues for other modules.
- *Combinability*: Allows us to recombine modules so that a new software application results.
- *Decomposability*: Allows us to decompose a monolith into smaller and simpler parts, which should be independently packaged into a different software unit. The resulting software unit should have a simpler structure and a lower level of complexity than the initial monolith. By breaking a system down into logical modules, we can understand the system much better and adjust it more easily.

Benefits of Modular Programming

We stress some important advantages of modular programming throughout this book, especially when we talk about Project Jigsaw, which brings modularity to Java 9. As you know by now, modular programming reduces the complexity of software applications. And it facilitates the reuse of software components. Moreover, it generally helps make debugging applications easier because only a single module can be debugged and not the entire monolith.

Modular programming allows a team to work together on the same project at the same time with fewer problems related to source code conflicts. If every developer works on their own module, there won't be any conflicts at all. Having to write less code is another benefit of using modular programming and it is a direct consequence of the reuse capability. By using modular programming techniques, developers gain better productivity and performance by taking advantage of parallel development.

Faster development is another key aspect of modular programming. The time required for development decreases because modules can be designed and implemented independently. The development process can be scaled because it's possible to develop more modules simultaneously by involving a larger team that can work on different modules at the same time. If a module is being modified, then the other modules will continue to work.

Modules should be easily interchanged with other modules. By defining an interface for the other modules of the application, changing or replacing a module implies only assuring that the new interface is equivalent to the old one. The internal implementation of the new module can differ.

Another important aspect refers to the testing process. Instead of testing an entire application as a whole monolith, the application is divided into modules, and each module is tested separately. Because the modules are independent, multiple modules can be tested at the same time, which speeds test execution and assures the integrity of the modules. By testing each module separately as a unit, better test coverage is achieved. Integration testing is performed by connecting the modules and looking at them as black boxes.

Now that you know what the principles and benefits of modular programming are, it's time to make a comparison between modular programming and object-oriented programming.

Modular Programming vs. Object-Oriented Programming (OOP)

The similarities between modular programming and OOP include the fact that both break large software applications down into *fragments* or *concerns*. Modular programming is not object-oriented. The core principles of OOP, such as polymorphism and inheritance, don't exist in modular programming. In OOP, polymorphism is used to dynamically change the properties of classes at runtime. This isn't possible in modular programming because the modules aren't dynamic. Besides that, in OOP classes use inheritance to enable other classes to inherit variables and method implementations from them. In modular programming, a module can't inherit another module.

One of the main differences between modular programming and OOP is the fact that in OOP objects can be created from classes. In modular programming, deriving objects from modules isn't possible.

Monolithic Application vs. Modular Application

A *monolithic application* is a software application with a high level of complexity that executes an entire group of tasks in order to implement a whole use case. It doesn't execute only a specific task or function and it doesn't consist of any logical units that can be identified. It has the role of executing entire functions, not just particular tasks inside these functions. Monolithic applications are constructed without modularity.

Figure 1-4 shows a possible architecture of a monolithic application.

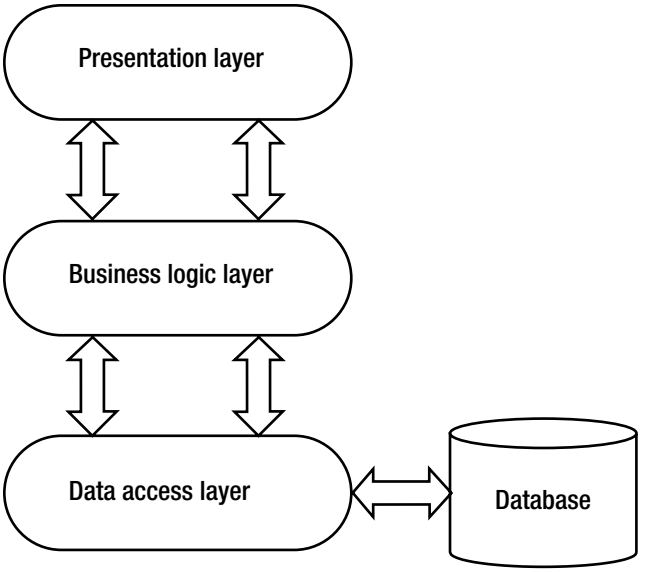

Figure 1-4. *Architecture of a monolithic application*

A monolithic application typically consists of many *layers*: presentation layer, business layer, data access layer, and database. With this system of layers, using multiple technologies for a single layer is difficult. We don't claim that it's absolutely impossible—it depends on the technologies you're using. Sometimes it's possible to use multiple technologies, and sometimes not. But having such constraints is definitely a drawback for every developer. If they find a third-party library that can solve a specific problem easier, they might not be able to use it because it's incompatible with the technology used on that layer. It would have been easier for them to be able to split the layers into different microservices and have the freedom to use, for each microservice, whichever technology better suits the task. Being forced sometimes to continue to use the existing technology is bad because the technology may already be old. For a modular system, because you may not have so many dependencies to care about, it is obviously not so time-consuming to update the technology and install the latest version of it, when compared to a monolithic system.

Due to the complexity of the system and the technical know-how required, there may be many cases when multiple teams are working on different layers of the application. To add a new feature to the application, every layer has to be addressed, which means in most cases, when a new feature is added, more than one team has to be involved. That can increase the time needed to develop and test new features because the teams will need to coordinate their work. There may also be more integration work to do—teams will not only have to coordinate with each other, they'll sometimes also have to wait for certain features to be ready so they can integrate their own piece of developed functionality.

As for scalability, a monolithic system can be scaled only as a whole. It's impossible to scale only specific parts of the system, and it's not a good idea to try to scale the entire system if only a part should be actually scaled. Scaling the entire system could mean additional infrastructure costs. This is why monolithic applications aren't so frequently upgraded and patched compared to modular applications.

A monolithic application *can* be separated into a set of modules, but that's not the only way to get a modular application. It can also be designed as modular from the beginning. Redesigning a monolithic application may be not a trivial task, especially when the system is very complex.

Summary

This chapter presented some general aspects on modularity. It discussed two important aspects of modularity: maintainability and reusability. It explained the concept of a software module and underlined the basics of the module declaration. It went over the four main concepts upon which a modular application is built: strong encapsulation, explicit interfaces, high module cohesion, and low module coupling. It presented the benefits of modular programming and compared modular and object-oriented programming. And the chapter illustrated what tight and loose coupling are and showed an explanatory example using Java.

Here's a short summary of when we should use modular programming:

- When we have a large program with lots of classes and methods and dependencies between them—such programs are always good candidates for modularization
- When we want to make a software application more suitable for future developments
- When we want to get out of monolithic applications
- When we want to make a software application more understandable

The next chapter explores the basics of Project Jigsaw, the new Java Platform Module System introduced in Java 9.

CHAPTER 2

Project Jigsaw

This chapter introduces Project Jigsaw. It describes what Project Jigsaw is all about, shows some problems encountered in the past in Java, and discusses the Java Enhancement Proposals that constitute Project Jigsaw. It explains the goals of Project Jigsaw so that we can get a grasp of the reasons that made the JCP team decide to introduce a new module system for the Java platform. It also goes over concepts like strong encapsulation and reliable configuration.

Weaknesses in Java Prior to JDK 9

Figure 2-1 shows the JDK 7 *module graph*, from the official OpenJDK website at `http://openjdk.java.net/projects/jigsaw/doc/jdk-modularization.html`. The base module is displayed right in the middle. Because there were a lot of dependencies between the classes, there was no way to split the monolith into smaller pieces. Besides that, the access delimiters didn't provide sufficient means to completely hide the implementation of the classes. Their scope was limited. For instance, in JDK 7 if we wanted to print "Hello world!" using the System console, we would need a great number of packages besides the base module.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_2

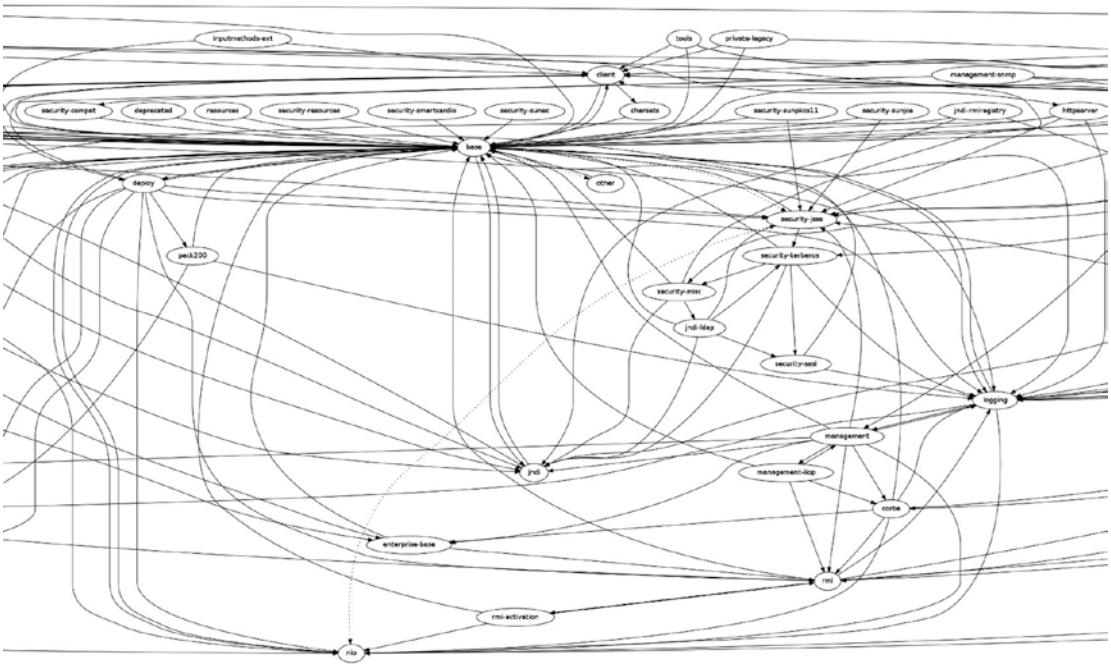

Figure 2-1. *The module graph in JDK 7*

JAR (Java Archive) files are ZIP files that contain class files and other resources. One or more JAR files are placed on the class path, which doesn't provide encapsulation for the JAR files it contains. This means that every class in a JAR file is accessible by every other class, and that represents a security vulnerability. You can imagine the class path like being a row where the JAR files are being searched linearly from left to right. JAR files are not components that can provide modularity for the classes that they contain.

Introduced in JDK 8, *compact profiles* are a subset of the full Java SE platform. The three compact profiles are called compact 1, compact 2, and compact 3. A list of the packages contained in each compact profile is listed on the Oracle website. The compact profiles were just a small step toward the modularization of the platform. Being able to get three different versions of the JDK by aggregating the standard packages was not what the community expected with respect to modularization. Besides that, compact profiles don't hide their internals and therefore brought no improvements to the security of the Java Platform.

Explicit dependencies were also a huge problem before Java 9. By packaging source code in a JAR file and putting it on the class path, there was no way of programmatically defining which other JARs were needed in order for the actual JAR to be able to run. Because Java didn't solve this problem, a few popular build tools have emerged, including Maven and Gradle.

Weak Encapsulation

For achieving encapsulation, Java prior to version 9 used the well-known access modifiers: private, protected, public, and no modifier. The *private access modifier* is the most restrictive. It makes the data inside unavailable from outside. The *protected access modifier* indicates that the member can only be accessed by a subclass of its class in another package or within its own package. The *public access modifier* makes data available everywhere. Using *no access modifier* means that availability is granted only inside the same package.

However, encapsulation has some limitations. It's impossible to make a type accessible to an external package and at the same time restrict access to it from all other existing packages. To make the type available to an external package, the only way would be to mark it as public, although defining it as public breaks encapsulation and makes it public for all the existing packages. There is no way to reach the desired level of encapsulation using Java prior to version 9.

JAR Hell Problem

Prior to JDK 9, the standard style for developing Java applications was to insert all the necessary libraries and JAR files directly on the class path. This approach could give rise to the JAR hell problem.

Before JDK 9, the runtime environment searched on a couple of locations in order to load a class. One of the searched locations is the class path, which contained a list of class files that were loaded by the Java Virtual Machine. Searching for a class on the class path was straightforward. The class loader searched a class by exploring all the JAR files listed on the class path. It didn't take into account a pre-definite order but just searched from the first to the last. It also didn't take into account aspects of the internal structure of the classes on the class path. Java wasn't able to take the boundaries between the JAR files into consideration. All the classes from all the JAR files were placed on the class path, and the boundaries between the JAR files disappeared. Each type in a JAR could access all the public types from any other JAR files. Therefore, the code couldn't be encapsulated in order to hide it from external use.

We'll give you an example. Suppose an author of a library had some internal code in the library that was never intended to be used from outside. Because there was no encapsulation, everyone could access this internal code—and worse, provide an own implementation that depended on the internal code. If the author of the library decided to make some changes in the internal code of his library, then code that depended on this library might encounter issues.

JAR hell is a common problem encountered prior to JDK 9. If there were more libraries on the class path with different versions and each library depended on another library, then there was said to be a JAR hell problem on the class path. This so-called *dependency hell* happened when a package had a dependency not on another package, but only on a version of that package. There were different variations of dependency hell, taking into account the environment being used. The problem with JAR hell is that you could have conflicts on the class path, especially when it contained many JARs. For instance, one library could have two or more different versions of a specific class on the class path. The class path wasn't the best solution because JAR files aren't components, and therefore we can't exactly know if something is missing or is conflicting.

If a specific class wasn't found on the classpath, a runtime exception was triggered—not during the launch of the application but at a later point when, due to an action performed by the user, the missing class was invoked. The runtime didn't have the capacity to identify all the existing dependencies until it had to access them. It would have been preferable to have all the errors displayed right during the start of the application and not at a later point.

What Is Project Jigsaw?

Project Jigsaw represents the implementation of the new scalable module system introduced in Java 9. It was developed under Open JDK, which is the free, open source implementation of the Java Platform Standard Edition. The goal of the newly designed module system for the Java SE Platform is to modularize the JDK and apply the module system to the JDK itself. Jigsaw modularizes the Java SE platform.

The process of modularizing the Java platform was a complicated and tremendous effort. A great number of difficult design decisions had to be made. The modularization of the platform is an enormous change with a major impact on the entire ecosystem. It introduces the new concept of modules and significantly changes the way we develop software applications using the Java programming language. Modules are placed in the foreground and are the key concept upon which Project Jigsaw is based. Entire programming techniques have to be adjusted to match the newly introduced concept.

Project Jigsaw started back in 2008 in an exploratory phase. The JEPs that constitute the Java Platform Module System were created starting with year 2014. Project Jigsaw was initially planned for the Java 7 release, but due to its complexity it was not included in the JDK 7 release and was postponed for the JDK 8 release. Then the Java Community Process deferred it for Java 9. Although the official release of Project Jigsaw is at the time of this writing planned for September 2017, early access builds have been available for a long time on the Open JDK website so that the community can test and provide valuable feedback to the JDK developers.

Project Jigsaw consists of six JEPs and a JSR. JSR 376 is called the Java Platform Module System. It designates a standard specification for building a modular version of the Java platform. Table 2-1 lists the other six JEPs that are part of Project Jigsaw.

Table 2-1. *JDK Enhancement Proposals (JEP) for the Development of the Java Platform System*

JEP Number	JEP Name	Scope
JEP 200	Modular JDK	Standard Edition
JEP 201	Modular Source Code	Implementation
JEP 220	Modular Run-Time Images	Standard Edition
JEP 260	Encapsulate Most Internal APIs	Java Development Kit
JEP 261	Module System	Standard Edition
JEP 282	jLink: The Java Linker	Java Development Kit

Following are short descriptions of each of the JEPs. They're covered in greater depth in the chapters that follow:

- *JEP 200—the Modular JDK*: This Java Enhancement Proposal divided the JDK into a set of modules. The JDK was modularized, and the source code was organized into modules. There are two different categories of modules: *standard* modules, with names that start with *java.*, and *JDK modules*, that start with *jdk*. A new module graph emerged as the modular format of the JDK changed (shown in Chapter 3). More information about the JDK modularization can also be found in Chapter 3.
- *JEP 201—Modular Source Code*: This JEP defines how the JDK build and the source code were reorganized around modules (covered in detail in Chapter 3).
- *JEP 220—Modular Run-Time Images*: JEP 220 presents the new modular runtime image and the enhancements added so that we can build custom modular runtime images. The binary structure of the JRE and JDK was changed. The JEP is discussed in Chapter 5.

- *JEP 260—Encapsulate Most Internal APIs*: JEP 260 refers to the process of encapsulating the non-critical internal APIs. This JEP is covered in many chapters throughout this book.
- *JEP 261—Module System*: JEP 261 represents the implementation of the new module system.
- *JEP 282—jLink: The Java Linker*: This JEP creates a tool that assembles a set of modules into a custom runtime image (covered in Chapter 7).

Downloading and Installing

As of September 2017, an early-access version of Project Jigsaw can be downloaded from the following URL address: `http://jdk.java.net/9/`. Project Jigsaw is contained into JDK 9. It cannot be used separately.

JDK 9 is available for download for the following platforms:

- Windows 32-bit
- Windows 64-bit
- Linux 32-bit
- Linux 64-bit
- Solaris SPARC 64-bit
- Solaris x86 64-bit
- Mac OS

Project Jigsaw was merged into JDK 9, so if you download JDK 9, you will have Jigsaw included by default.

The installation is straightforward. You have to set the environment variables on your PC to point to the new JDK. For this, choose the root folder where your Java 9 installation resides. If you use Windows and JDK 9 is in the PATH, you can verify that the environment variables have been successfully set by opening a command line and typing `java -version`.

Documentation

There is plenty of documentation available for Project Jigsaw on the official Open JDK website at `http://openjdk.java.net/projects/jigsaw`. You'll find descriptions for each of the JEPs that constitute the Java Platform Module System. The specification document can be found at `http://openjdk.java.net/projects/jigsaw/spec/reqs/`.

To get more deep insight into Project Jigsaw, you can access the Jigsaw Development mailing list, which contains comprehensive information about the internals of Jigsaw, at `http://mail.openjdk.java.net/pipermail/jigsaw-dev/`. Other mailing lists that might interest you are the Expert Group mailing list at `http://mail.openjdk.java.net/pipermail/jpms-spec-experts/` and the Adoption Discuss mailing list at `http://mail.openjdk.java.net/pipermail/adoption-discuss/`.

The API Specification for the Java 9 Standard Edition can be found at `http://download.java.net/java/jdk9/docs/api/overview-summary.html`.

Goals of Project Jigsaw

The goals of Project Jigsaw, as listed on the Open JDK website at http://openjdk.java.net/projects/jigsaw/, are as follows:

1. To make the Java SE Platform and the JDK more easily scalable down to small computing devices
2. To improve the security and maintainability of Java SE platform implementations in general and the JDK in particular
3. To enable improved application performance
4. To make it easier for developers to construct and maintain libraries and large applications, for both the Java SE and EE platforms

The module system should be powerful enough to modularize the JDK and other large legacy code bases, yet still be approachable by all the developers.

The module system split the Java platform into modules that can be managed by users. Modules can hide their internal implementation but still interact together efficiently. They have the ability to manifest explicitly if they're entirely accessible to other modules, if only some parts of their types are accessible to other modules, or if they're not accessible to other modules at all. They can also specify the list of modules to which they're available. These Java 9 features are called reliable configuration and strong encapsulation and are achieved in an easy way.

The problems related to JAR hell on the class path are solved by using the new module path instead of the class path. The JDK maintenance and administration is reduced by encapsulating the internal APIs. Project Jigsaw masks some of the internal APIs of the JDK by taking advantage of the strong encapsulation mechanism. The public types in the JDK internal APIs' packages are a lot harder to access, which leads to breaking code in some applications that are using internal APIs.

Let's look at the motivation behind the decision to develop Project Jigsaw. At the beginning of the project, the goal was, according to Open JDK, "to design and implement a module system focused narrowly upon the goal of modularizing the JDK, and to apply that system to the JDK itself."

Before Java 9, JDK was a big, indivisible monolith with more than 5.500 classes. It was impossible to split it into more pieces. The only way to use it was to install it entirely on the target platform. The Java runtime represented by the rt.jar file was also monolithic and couldn't be split into more parts. JDK consisted of rt.jar, which contained almost all the compiled classes for the base Java runtime. rt.jar grouped together all the runtime class files and had to be placed on the class path in order for the user to be able to access the Java API classes. Inside rt.jar there were—besides the popular java.* and javax.* packages— other packages such as com.oracle.*, com.sun.*, jdk.internal.*, jdk.management.*, jdk.net.*, sun.*, and more. There was no way to split rt.jar into different files. In Java 9, the focus was to break the monolithic JDK into modules and to completely remove the rt.jar file.

The JDK had to be modularized because, after more than 22 years since its first release, it had grown so much that it effectively became too big and too complex. Installing JDK on small devices can be cumbersome in certain situations, because not all small devices have enough CPU, memory, or disk space to be able to hold the entire JDK. Besides that, it's a huge waste of memory to install the entire monolithic JDK and use only a small portion of it in your application. This problem relates not only to small devices, but to big devices as well, like the ones used to hold the applications in the cloud. Important additional costs could occur using the cloud because the use of hardware resources isn't optimized.

JDK 1.0, released at the beginning of 1996, was extremely small and tiny in comparison to the actual release of the JDK. The first version of Java had only a few standard packages: java.lang, java.io, java.applet, java.awt, java.net, and java.util. There were a total of 8 packages and 212 classes and interfaces. Every release after JDK 1.0 added more and more complexity. JDK 1.1 had 504 classes and interfaces. JDK 1.2, released in 1998, tripled the number of classes and interfaces to 1,520. During the next JDK releases, this increase continued: JDK 1.4 had 2,991 classes and interfaces, and JSE 8.0 had 4,240 classes and interfaces. Compared

to JDK 1.0, the release of Java 8 contained exactly 20 times more classes and interfaces. Besides the big size, the JDK prior to version 9 was very complex due to the dependencies between the APIs.

Previous to JDK 9, for a simple program that prints a string in the console, a great number of classes had to be loaded. For example, the base module in JDK 7 depended on a lot of other modules such as logging, security-smartcardio, security-sunec, security-resources, resources, charsets, client, security-misc, security-jsse, security-kerberos, and others. All these modules had to be loaded in order to print a basic and very simple "Hello world!" in the console.

Taking into consideration all the facts previously mentioned, the decision to work at splitting the JDK into modules was absolute necessary. A modular JDK reduces the number of classes loaded because there aren't any dependencies or connections between classes from different modules anymore. Java 9 also improves the startup time of Java applications, and the memory footprint in Java 9 is better than in previous releases.

New Concepts Introduced in Jigsaw

Project Jigsaw introduces the new concept of *module* as a central software component that is built inside the Java platform. A module represents a collection of packages. It has a module descriptor that specifies the modules upon which the module depends and also specifies its exported packages that are made available for external use. A module can be packaged into a new format called *modular JAR*, which is a JAR file that also contains a module-info.class file. A modular JAR file can function as a module in Java 9, but also as a regular JAR file on the class path in Java 8 or earlier. There's another new format called JMOD, which is similar to a modular JAR but can also contain native code. A module can be open or not. Chapter 4 describes modules in detail.

The new notion of *module path* is introduced in Project Jigsaw. Module path is the module equivalent of the class path and consists of a list of directories that contain modules. Upcoming chapters cover the module path and show you how to use it alone or in combination with the class path.

Jigsaw also introduces a *linking phase* in which a group of modules is assembled by a new linking tool, called Jlink (covered in Chapter 7), into a custom binary runtime image. Linking can create a full Java development environment and can also create a Java runtime system incorporated in a program.

Java 9 adds many new options for both the Java compiler and the Java launcher in order to allow the compilation and running of modules. Throughout this book you'll find plenty of examples of compiling and running modules using different command-line options. Jigsaw also introduces new notions like *unnamed module*, *open module*, and *automatic module*, all covered in Chapter 4.

Strong Encapsulation

According to the official Jigsaw specification, "strong encapsulation allows a component to declare which of its public types are accessible to other components and which are not." Strong encapsulation's role is to forbid code from accessing classes in packages that aren't exported by their containing modules, or in packages whose containing modules aren't needed by the module that contains the code.

Strong encapsulation couldn't be achieved without having a concept like modules, because in Jigsaw the modules represent the base on which the principles of strong encapsulation are applied. Jigsaw allows modules to export only specific packages. The accessibility of modules is provided by their boundaries. Strong encapsulation is accomplished in Jigsaw using the definitions of the modules, where we are able to specify what types are accessible. Strong encapsulation hides module's internals and prevents them from external access. It also makes it more difficult to achieve reflective access.

In Java 9, calling the method `setAccessible()` won't work unless the object is accessible before the class. To be accessible, the corresponding package has to be exported, and the module has to be read. If both conditions are met, then it is accessible, so the method can be applied to make, for instance, a private field available. Strong encapsulation restricts access even when the accessing class in the target class is in the same class loader. By the way, strong encapsulation is not dependent on class loaders.

Reliable Configuration

Reliable configuration is a strong feature introduced in JDK 9. Open JDK states that "reliable configuration replaces the class path mechanism with a means for program components to declare explicit dependences upon one another." Reliable configuration is based on the capacity to declare dependencies between modules. It allows us to know at compile-time if a module is missing or a dependency isn't fulfilled. This is something we could't achieve in versions before Java 9. In JDK 9, modules can manifest their dependencies on other modules, and the module system certifies that every module dependence is achieved.

The ground for reliable configuration is represented by the readability connections that exist in the module system. Dependencies are analyzed and enforced at both compile-time and runtime. In chapter 4 you'll learn how reliable configuration is achieved in Jigsaw by the `requires` clause and how strong encapsulation is achieved by the `exports` clause in the module declaration.

Enhancements Provided by Jigsaw

Jigsaw also provides enhancements in three important areas: security, scalability, and performance.

Security

Java had a considerable amount of security issues in the past. Prior to JDK 9, as mentioned, there was no encapsulation across package boundaries. Hence, security is a very important subject in Jigsaw, being one of the key factors in some of the implemented design considerations. In Java 9, some portions of the code can't be accessed directly anymore. Jigsaw improves security significantly and greatly reduces the security risks by hiding the JDK internal APIs. We call this the *encapsulation of the JDK internal APIs*. They can now be handled only within the JDK itself. To improve security, it was not enough to only encapsulate the JDK internal APIs—its number of uses was decreased too. By specifying module boundaries, code is no longer reachable from outside of the module unless it's explicitly defined so. By default, it's not reachable from outside.

Security is improved by the newly introduced strong encapsulation mechanism, which hides module's internals. Critical source code is hidden and is not accessible from outside unless absolutely necessary. Attempting to access a public JDK internal type results in an access error. This is why code that uses internal APIs no longer works starting with Java 9.

In Jigsaw, the mechanisms that allow access to internal classes using reflection have been hardened. This is a great improvement because in the past the benefit of accessing internal JDK classes resulted in many security incidents in the Java platform. As the number of internal JDK classes decreases in Java 9, so do the number of potential breaches.

Prior to JDK 9, Java had a serious issue regarding the fact that its classes were accessible from external code running in the same environment. It had very limited ways of restricting the access to its code from outside. In order to restrict package access, Java used the method `checkPackageAccess(String packageName)` of class `java.lang.SecurityManager`. This method gets a list of restricted packages from calling `java.security.Security.getProperty("package.access")` and checks whether the parameter `packageName` is between the retrieved packages. If not, then the method throws a `SecurityException`. If `packageName` is found, then the method `checkPermission()` is called. Some of these security problems from the past were related to the fact that the software developers sometimes forgot to call the `checkPackageAccess()` method in code everywhere it was necessary. If this check isn't done everywhere, then the code can be accessed from outside and a big security breach is opened, causing potential damage. It was the responsibility of every JCP developer to be careful and not forget to put the `checkPackageAccess()` call everywhere it was necessary.

Scalability and Performance

Jigsaw allows developers to create their own Java Runtime Environments (JREs) that contain only the modules they need. A great number of small devices benefit from the prospect of being able to group only the functionality that is strictly required by the running Java software application.

Performance is enhanced during the class loading process because the Java Virtual Machine now knows where the location of a class is. Because we know in advance all the classes that a class refers to, the JVM can eventually perform optimizations that will result in a performance increase. Before Java 9, the JVM had to open every JAR file and perform a linear search in order to find a class, which imposed a huge cost on performance.

The removal of rt.jar in Java 9 was a good design decision with respect to performance because it allowed the introduction of a new productive storage system. The performance of Java applications has been improved in Java 9, especially at startup time. For this, the structure of Java runtime has been modified. There's now enough potential for future performance optimizations because portions of code are reached only by the modules they depend on.

The degree of scalability of the Java platform is increased by allowing developers to create smaller and more optimized deployments that help reduce the amount of memory needed on the corresponding running device. The new custom runtime images contain only the specific libraries and the minimum number of dependencies needed to run a Java application. There's no longer a requirement to install the entire JDK. It's possible to select exactly the modules needed by an application.

Other Generalities

New Keywords in Java 9

`Module` is a restricted keyword that acts like a keyword only in relation to a module declaration. When a module isn't used in connection with a module declaration, then the word `module` can further be used as an identifier. This means that if we used the word `module` to define the name of a variable, an instance variable, or a method, we don't have to change it.

Other restricted keywords introduced in Java 9 include `exports`, `requires`, `provides`, `uses`, `with`, `to`, `transitive`, and `opens`.

No Versioning in Jigsaw

Versioning is not supported in Project Jigsaw. The JCP team included versioning in the first releases of Jigsaw but then decided to leave it out due to the complexity and complications that subsequently occurred. The decision was based on the fact that build tools like Gradle or Maven have better mechanisms to handle this complicated problem. Project Jigsaw relies on these build tools for solving versioning resolution or dealing with different conflicts. Jigsaw allows you to declare a version in the meta information of a module, but this version isn't taken into account by the module system. Chapter 10 talks about layers, and you'll learn how Jigsaw can load two different versions of a module. Hence, these are all the features that Jigsaw brings for versioning. For instance, declaring in a module declaration that a module dependends only on a specific version of another module isn't supported.

Backward Compatibility

Having backward compatibility to older versions of the JDK was a critical topic during the design of JDK 9. The JCP team states, "if an application uses only supported APIs and works on release X, then it should work on release X+1 even without performing a recompilation." Fixing the incompatibilities inside the source code is mandatory in order to make a Java software application work after moving to Java 9.

The requirements of Project Jigsaw, published on the Open JDK page, state: "It must be possible to divide an existing Java Platform, Java SE or Java EE, into a set of modules such that existing libraries can run without change, so long as they use only standard platform APIs." Java applications will be backward-compatible with versions prior to JDK 9 as long as they use only standard platform APIs. If they're using other APIs besides the standard ones, there's no guarantee they'll work in JDK 9.

Some compatibility problems could appear for applications or libraries that use core reflection to gain access to JDK internal types. For the builds to work, using the command-line flag `--add-exports` is mandatory in order to break encapsulation. This command-line option is covered in Chapter 8.

Another issue can occur when an existing library from the class path has a reference to a type in a non-exported package that belongs to an explicit module. In order to solve it, using the class path in combination with the module path is required. In upcoming chapters you'll learn how to achieve that. It's important to remember that, for having backward compatibility, the class path can be further used in JDK 9.

Nevertheless, in order to provide backward compatibility, the three class loaders are still present in JDK 9. Chapter 10 talks about them.

Platform Modularization

One of the most important roles of Project Jigsaw is to split the JDK into modules, explained in detail in Chapter 3. The resulting modules can be divided into three distinct categories: standard modules, JDK-specific modules, and JDK-internal modules.

Before Java 9, rt.jar contained many publicly accessible APIs that were planned for public use. It was possible to use them when writing your own code. Among the publicly accessible APIs there are plenty that are part of the standard Java SE. These packages start with java.* and javax.* and are specified by JCP. Other packages that form the publicly accessible APIs, but are not part of the standard Java SE, are the jdk.* and com.sun.* packages. These packages are not part of the standard Java SE because they're intended to be used by tools that interact with the Java Virtual Machine, for instance. It doesn't make sense to make them part of the standard Java Standard Edition.

Besides the supported APIs, there are also unsupported APIs. Most of them reside in the sun.* package. They're not meant to be used publicly. A survey organized by Oracle revealed that the most popular unsupported APIs are sun.misc.Base64Encoder, sun.misc.Unsafe, and sun.misc.Base64Decoder. Oracle classified the APIs based on their usage and organized them into *critical* and *non-critical*. The non-critical APIs have very little usage outside the JDK.

New Structure of the JRE and JDK

In order to provide the means for creating runtime images, the binary structure of the JDK and JRE was changed in Java 9. Due to the introduction of modules, there's no difference between the JDK and the JRE. Every tool that depends on rt.jar had to be changed in order to work further properly in Java 9.

Figure 2-2 illustrates the old structure of the JDK and JRE, prior to Java 9.

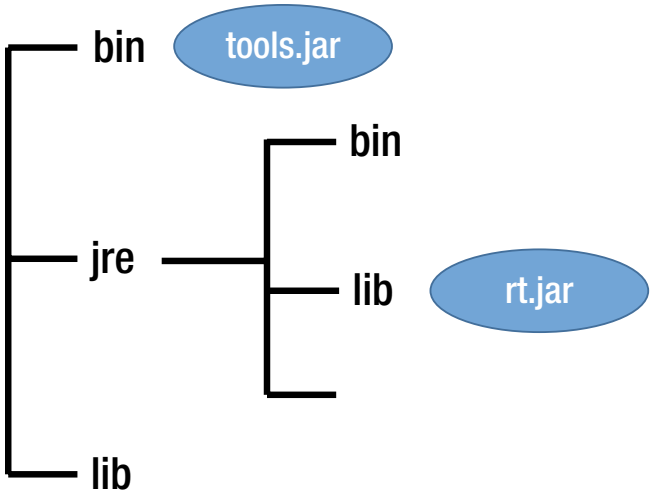

Figure 2-2. *The structure of the JDK prior to Java 9*

Before JDK 9, there were two bin and two lib directories. The lib directory from the top level contained classes for tools, and the lib directory from the jre directory contained the runtime classes. The lib directory also contained configuration files, security policy files, and other types of files.

Figure 2-3 shows which files and directories have been completely removed in JDK 9: the jre directory, the tools.jar file, and the rt.jar file.

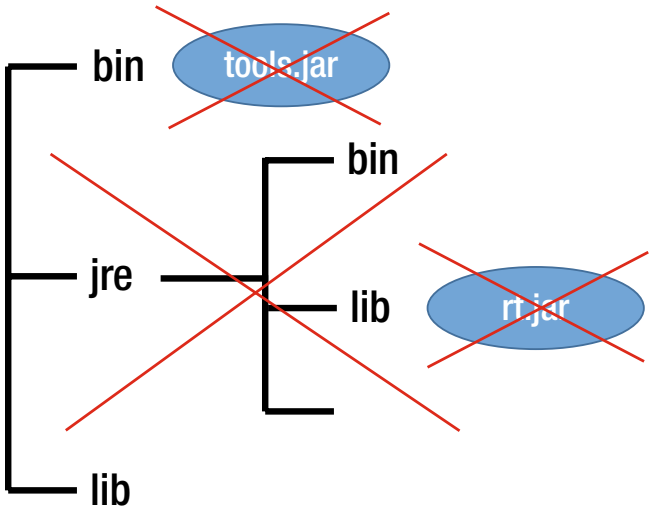

Figure 2-3. *The jre directory, tools.jar, and rt.jar were removed in JDK 9*

Figure 2-4 depicts the final structure of the JDK 9.

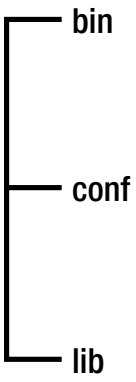

Figure 2-4. *The new layout of JDK 9*

As you can see, the jre directory doesn't exist anymore, and a new conf directory was added. The conf directory contains the configuration files that customize the JDK or the runtime. It contains only the files that should be edited. The files that should not be edited are not located in the conf directory anymore. This is important because the new layout provides a clear separation between the configuration files that are allowed to be changed and the ones that aren't. In the past it was a risk to change a configuration file because you couldn't know in advance if you were even allowed to change it, meaning the application might not start anymore.

The rt.jar and tools.jar files have been completely removed. The bin directory, which is now a single one, contains all the launchers. The new format of the JDK is more suitable for future optimizations than the old format.

How to Prepare for Jigsaw

There are some steps that you should perform in order to prepare for Project Jigsaw. As you already know, the JDK-internal classes aren't accessible any more, unless they're part of the package jdk.unsupported. Code that relies on JDK-internal classes will break.

First of all, you should check your code for uses of JDK-internal APIs using JDeps. JDeps is a tool for analyzing and finding statical dependencies (it's covered in detail in Chapter 8). If you find uses of JDK-internal classes throughout your code, you should provide replacements for them. JDeps gives you hints and proposes alternatives for your JDK-internal classes, but it's your responsibility to get rid of these classes and replace them with supported ones.

The Java compiler command-line option `--add-exports` is an alternative if you can't provide replacements for the JDK-internal classes. This command breaks the encapsulation and makes the JDK-internal classes accessible in your code. In this way, you don't have to modify your source code—you just have to adapt your build scripts to include this option during the compilation of your code.

Besides the uses of the JDK-internal classes, you should also check your code for dependencies on the rt.jar and tools.jar files. Both files were removed in Java 9, and code that relies on them will stop working. You can't use them throughout your code anymore in Java 9.

Another important topic refers to the *split packages*, which should be definitely avoided. Split packages arise when two or more loaders designate classes for a single package. You have to get rid of the split packages before migrating to Java 9. Chapter 8 covers the split packages problem in detail and shows you how to get rid of them.

Migrating to Java 9 can be challenging under some circumstances. That's why we cover this topic in more detail in Chapter 8. The list just presented isn't complete. If you want to know more about how to prepare for using Jigsaw, go directly to Chapter 8.

Differences Between OSGi and Jigsaw

OSGi (Open Service Gateway Initiative) is a well-known framework that allows developing modular applications in the Java programming language. The specification for implementing module systems in Java using OSGi can be found in the document JSR 291 – Dynamic Component Support for Java SE, which was released in August 2007.

We won't go deep into detail regarding OSGi because OSGi is beyond the scope of this book, but we will present some important distinctions between it and Jigsaw. One major difference between OSGi and Jigsaw is the fact that OSGi supports versioning, but Jigsaw does only at a low degree. Jigsaw allows you to define a version as a meta attribute or to use multiple versions for a module in a layer. But the versioning system offered by Jigsaw is far less powerful than the one offered by OSGi. OSGi also has some features related to dynamic life-cycle that Jigsaw does not have. Besides this, OSGi provides a dynamic service registry and an upgraded security model.

Jigsaw is more secure than OSGi because its security mechanism can't be bypassed. The security mechanism from OSGi can be bypassed. The OSGi bundles don't give the same level as security compared to the Jigsaw modules.

Jigsaw isn't intended to be a replacement for OSGi. OSGi can operate very well on top of JDK 9. JCP aims to make both systems able to work in parallel and to cooperate. It should even be possible for OSGi to treat a Jigsaw module as an OSGi bundle.

There may be specific cases when OSGi is more suitable to the needs of an application than Jigsaw. Jigsaw is more suited for software applications that don't present an extremely high level of complexity. OSGi takes advantage of true isolation because it's built on top of the platform. Both OSGi and Jigsaw provide isolation, but the way that's achieved differs. In OSGi, isolation is achieved automatically because OSGi is built on top of the platform. In Jigsaw, the modules have been built inside the platform, not on top of it. They present isolation programmatically by the manner in which they are designed into the platform.

In general, Jigsaw has fewer features than OSGi. For instance, Jigsaw doesn't offer the possibility to dynamically download and load modules from a repository when an application and the virtual machine are running. This kind of feature doesn't exist in Jigsaw, and OSGi should be used instead for this specific case.

Project Jigsaw also offers important features that don't exist in OSGi, such as modularity at compile-time and built-in support for native libraries. Jigsaw, in contrast to OSGi, modularizes the Java platform and introduces the new concept of modules as a central program element.

Summary

We started this chapter by presenting some weaknesses and problems that occurred prior to Java 9, such as weak encapsulation and the Jar Hell problem. Then we introduced Project Jigsaw, the new module system introduced in Java 9, and described the goals of Project Jigsaw and some of the problems it solves. The chapter briefly presented a couple of new concepts introduced in Jigsaw.

We talked about the strong encapsulation and reliable configuration mechanisms introduced in Jigsaw, which make a public type not accessible from outside of its module unless it's in an exported package. We also described other enhancements provided by Jigsaw in the fields of security, scalability, and performance. Prior to JDK 9, it was much more difficult to maintain our code because we couldn't encapsulate it in order to hide our internal implementation from external use.

www.allitebooks.com

The next topics were backward compatibility and platform modulatization, and we provided insights into the new categories of APIs in the JDK. Then we presented the new structure of the JRE and JDK in Java 9 and talked about the removal of rt.jar and tools.jar. Next we outlined some of the most important steps you have to take in order to prepare for using Jigsaw in your projects. The end of the chapter illustrated some of the most important differences between OSGi and Jigsaw.

Chapter 3 describes the JDK modularization process, the resulting modular JDK, and the way source code was modularized in JDK 9.

CHAPTER 3

Modular JDK and Source Code

This chapter focuses on describing the JDK modularization process that resulted in a new structure of the JDK and its source code. According to Open JDK, the aim of Java Enhancement Proposal 200 – The Modular JDK is to "divide the JDK into a set of modules that can be combined at compile time, build time, or runtime into a variety of configurations." These configurations can have any size. They can represent one or more modules together with their transitive dependencies, but they can also comprise the entire JDK.

The JDK Module Summary consists of comprehensive information related to the modules that currently exist in the Java Platform. For each module it specifies the following:

- The number of classes and resources it contains
- The total size of the module together with the total size of its dependencies
- The modules it requires
- The types it exports
- The services it uses and the services it provides

Note The JDK Module Summary can be found online at `http://cr.openjdk.java.net/~mr/jigsaw/ea/module-summary.html`.

As of September 2017, Project Jigsaw introduced 73 new modules in the Java platform, with a total size of more than 170 MB. Taking the number of classes in each module, the biggest module is java.desktop, which contains 5,900 classes and 284 resources. Its size is more than 26 MB, and the total size of its dependencies is about 55 MB. The second largest module is module java.base, which contains 5,684 classes and 17 resources. It has no dependencies because it's the base module.

Modular JDK

In Java 9, the JDK is modularized. In order to list all modules that exist in the runtime system, the Java launcher can be used with the command-line option `--list-modules`. By running the following command, we get a full list of the existing modules in our runtime:

```
$ java --list-modules
```

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_3

Table 3-1 displays the results.

Table 3-1. *The Modules of the Java Runtime System*

java.activation	java.xml.crypto	jdk.jfr
java.base	java.xml.ws	jdk.jsobject
java.compiler	java.xml.ws.annotation	jdk.localedata
java.corba	javafx.base	jdk.management
java.datatransfer	javafx.controls	jdk.management.agent
java.desktop	javafx.deploy	jdk.naming.dns
java.instrument	javafx.fxml	jdk.naming.rmi
java.jnlp	javafx.graphics	jdk.net
java.logging	javafx.media	jdk.pack
java.management	javafx.swing	jdk.plugin
java.management.rmi	javafx.web	jdk.plugin.dom
java.naming	jdk.accessibility	jdk.plugin.server
java.prefs	jdk.charsets	jdk.scripting.nashorn
java.rmi	jdk.crypto.cryptoki	jdk.scripting.nashorn.shell
java.scripting	jdk.crypto.ec	jdk.sctp
java.se	jdk.crypto.mscapi	jdk.security.auth
java.se.ee	jdk.deploy	jdk.security.jgss
java.security.jgss	jdk.deploy.controlpanel	jdk.snmp
java.security.sasl	jdk.dynalink	jdk.unsupported
java.smartcardio	jdk.httpserver	jdk.xml.dom
java.sql	jdk.incubator.httpclient	jdk.zipfs
java.sql.rowset	jdk.internal.le	oracle.desktop
java.transaction	jdk.internal.vm.ci	oracle.net
java.xml	jdk.javaws	
java.xml.bind	jdk.jdwp.agent	

Table 3-2 contains a short description of each of the standard Java SE modules, as described in the JDK 9 API documentation.

Table 3-2. *The Standard Modules According to the Java Platform Standard Edition 9 API Specification*

Module name	Description
java.activation	Represents the JavaBeans Activation Framework API
java.base	Represents the primary APIs of the Java SE platform
java.compiler	Expresses the Annotation Processing, Language Model, and Java Compiler APIs
java.corba	Defines the RMI-IIOP API and the OMG CORBA APIs
java.datatransfer	Defines an API for exchanging information between applications
java.desktop	Comprises the AWT and Swing user interface toolkits and also APIs for printing, audio, imaging, and more
java.instrument	Contains the services that permit agents to instrument programs that execute on the Java Virtual Machine
java.logging	Expresses the Java Logging API
java.management	Represents the Java Management Extensions API
java.management.rmi	Represents the RMI connector for the Java Management Extensions API
java.naming	Contains the Java Naming and Directory Interface API
java.prefs	Specifies the Preferences API
java.rmi	Holds the Remote Method Invocation API
java.scripting	Represents the Scripting API
java.se	Represents the core Java SE API
java.se.ee	Represents the full API of the Java SE platform
java.security.jgss	Includes the Java binding of the Generic Security Services API
java.security.sasl	Contains Java support for the Simple Authentication and Security Layer
java.sql	Represents the Java DataBase Connectivity API
java.sql.rowset	Determines the JDBC RowSet API
java.transaction	Specifies a subdivision of the Java Transaction API
java.xml	Includes the Java API for XML Processing, the Streaming API for XML, the Simple API for XML, and the W3C Document Object Model API
java.xml.bind	Represents the Java Architecture for XML Binding API
java.xml.crypto	Describes the XML Cryptography API
java.xml.ws	Specifies the Web Services Metadata API and the Java API for XML-based Web Services
java.xml.ws.annotation	Specifies a part of the Commons Annotations APIs to support programs running on the Java SE platform

Platform Modules

The JCP team put a lot of effort into modularizing the Java platform. The most difficult tasks were to investigate and evaluate the dependencies between different parts of the libraries and to split all the classes from the JDK and put them into modules.

The *platform modules* are the modules that resulted after splitting the JDK. They completely replace the monolithic JDK and enable us to create custom runtime images. These can consist of a specific configuration that contains a subset of modules together with their transitive dependencies. This subset of modules can represent one module or more than one module. It can also represent all the modules, which is the equivalent of the entire JDK. It's also possible to combine platform modules together with our own created modules in order to form a runtime image.

Each module has a determined functionality and can define dependencies upon other modules. A platform module is part of the Java runtime and contains source code. Platform modules are able to export their packages so they can be accessible from other modules that read them. When we generally talk about modules, we mean not only the platform modules but also the modules created by application programmers. These modules don't have a special definition. We could call them "developer modules" or "programmer modules" in order to make a clear distinction between them and the modules that are by default part of the platform, which are the platform modules.

There are two different kinds of platform modules: standard modules and non-standard modules.

Standard Modules

The standard modules are managed by the Java Community Process (JCP). The names of the standard Java SE modules start with java.*. These names are explicit enough, so it's quite easy to imagine what the role of the module is. For example, the module named java.rmi defines the Remote Method Invocation API, and the module named java.logging defines the Java Logging API. A standard module can consist of standard API packages as well as non-standard API packages. It can also depend upon one or more non-standard modules.

Non-standard Modules

The non-standard modules are specific to JDK. Their names start with jdk.*. Non-standard modules contain packages and specific JDK code, which may be distinct between various implementations of the Java Development Kit. Some JDK modules, such as tools or service providers, don't export anything, which means they're not visible outside the module.

Two things are very import to remember: first, that standard API packages must not be exported by non-standard modules, so their visibility remains hidden from outside. Second, and certainly the most important thing to keep in mind, is that the source code that depends only upon Java SE modules will depend only upon standard Java SE types. This is a great advantage because the code becomes portable to all the existing implementations of the Java SE platform, as stated in the official description of the JEP 200 under `http://openjdk.java.net`.

Turning a JDK-specific API into a Java standard API is possible, but special focus is required regarding compatibility. When considering doing this, you should take into account whether it's feasible and necessary by looking at the way it's used. For example, the reason why the Java Debug Interface wasn't turned into a standard API is because it's used only by tools and debuggers, so it definitely doesn't make sense to enhance it to be part of the Java Standard API.

Every platform module contains a folder called classes inside a folder called share. The classes folder contains all the classes that compose the module, along with the module descriptor in a file called module-info.java. Some modules, such as the java.base module, have native code for different operating systems like Windows, Linux, macOS, and so on.

The JDK Module Graph

The modularization of the Java 9 platform can be well represented as a module graph. Figure 3-1 shows an excerpt of the new module graph of the JDK containing only the standard SE modules. It is resulted after splitting the JDK into modules.

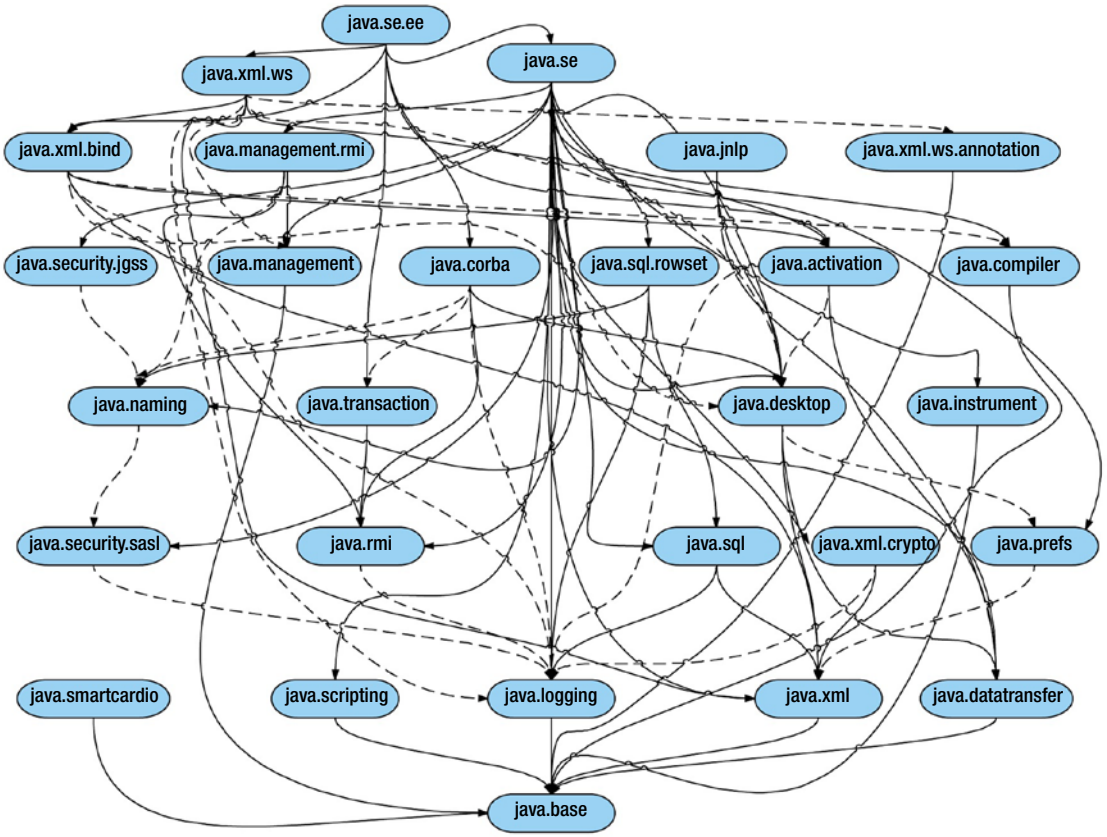

Figure 3-1. *A part of the module graph of JDK 9 representing only the standard SE modules*

Only the standard Java SE modules are shown in this graph (the non-Java SE modules aren't shown due to lack of space). In the graph, the modules are represented by the nodes, and the dependencies between modules are expressed with arrows. If a module depends on another module, there's a direct arrow from one module to another.

We have two categories of lines between the modules. The solid lines illustrate an implied readability between modules, and the dashed lines mean there is only simple readability between modules but no implied readability. But in both cases, the module reads another module, meaning the module depends on the other module. For instance, there's a solid line between module java.transaction and module java.rmi. This means that module java.transaction `requires transitive` module java.rmi. There's also a dashed line between module java.xml.ws and module java.xml.ws.annotation, which means that module java.xml.ws `requires` module java.xml.ws.annotation. In other words, module java.xml.ws uses types from module java.xml.ws.annotation.

The graph is hierarchical and clean, has no cycles, and contains no split packages. It has no circular dependencies, because they're not allowed. The java.base module is right at the bottom of the graph. It doesn't depend on any other module. All the rest of the modules depend directly or indirectly on module java.base (not shown in the module graph due to lack of space). Therefore, there are lines to module java.base only for the modules that require only java.base and nothing else. For the modules that require at least one more module besides java.base, there's no line to module java.base.

The java.se.ee module is located at the top of the module graph. It acts like an aggregator module and consists not only of all the Java SE modules, but also of the modules that overlap with the specification of Java EE. Module java.se.ee adds none content of its own. It has only a module descriptor that gathers the contents of the following modules:

- module java.se
- module java.activation
- module java.xml.ws.annotation
- module java.corba
- module java.transaction
- module java.xml.bind
- module java.xml.ws

In chapter 4, we show what a module descriptor is. The module java.se.ee consists of all the Java SE APIs. Compared to module java.se.ee, the java.se module is an aggregator that consists of the parts of the Java SE that don't overlap with Java EE. The java.se module gathers the contents of the following modules: java.datatransfer, java.logging, java.sql, java.instrument, java.security.jgss, java.security.sasl, java.prefs, java.xml.crypto, java.rmi, java.xml, java.naming, java.compiler, java.desktop, java.scripting, java.management.rmi, java.sql.rowset, java.management, and java.base.

■ **Note** The module graph in Figure 3-1 shows only the standard SE modules. The non-standard SE modules (with names that start with jdk.*), the Java FX modules (names start with javafx.*), and the Oracle modules (names start with oracle.*) aren't displayed in this module graph.

More on Modules

Now we will learn how to read the description of a module and to present the module java.base.

Read the Description of a Module

To get the entire description of a module, we can use the `--describe-module` command-line option of the Java launcher, followed by the module name:

```
$ java --describe-module <module_name>
```

By running the `--describe-module` option on module java.naming, we get the following output:

```
java.naming@9
exports javax.naming
exports javax.naming.directory
exports javax.naming.event
```

```
exports javax.naming.ldap
exports javax.naming.spi
requires java.base mandated
requires java.security.sasl
uses javax.naming.spi.InitialContextFactory
uses javax.naming.ldap.StartTlsResponse
provides java.security.Provider with sun.security.provider.certpath.ldap.JdkLDAP
qualified exports com.sun.jndi.toolkit.ctx to jdk.naming.dns
qualified exports com.sun.jndi.toolkit.url to jdk.naming.dns jdk.naming.rmi
contains com.sun.jndi.ldap
contains com.sun.jndi.ldap.ext
contains com.sun.jndi.ldap.pool
contains com.sun.jndi.ldap.sasl
contains com.sun.jndi.toolkit.dir
contains com.sun.jndi.url.ldap
contains com.sun.jndi.url.ldaps
contains com.sun.naming.internal
contains sun.security.provider.certpath.ldap
```

The preceding code displays the entire information contained in the module-info.java file of the java.naming module. It additionally contains the `contains` clauses, which aren't displayed in the module-info.java file:

- The `exports` statements denote that module java.naming make the packages javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap, and javax.naming.spi available to any other module that depends on it (on module java.naming).
- The `requires` statements from the preceding code denote that module java.naming depends on module java.base and also on module java.security.sasl, meaning that the exported types in those two modules are used inside the java.naming module. For instance, the class `LDAPCertStore` from the java.naming module imports all the java.security subpackages that are part of module java.security.sasl.
- The `uses` statement takes as an argument a type name that represent a service type. In our case, module java.naming consumes instances of `InitialContextFactory` and `StartTlsResponse`.
- The `provides` statement specifies that the module provides the implementation of java.security.Provider (from module java.base) with sun.security.provider.certpath.ldap.JdkLDAP.
- The statement `qualified exports com.sun.jndi.toolkit.url to jdk.naming.rmi` means that package com.sun.jndi.toolkit.url from the java.naming module should be accessible only in the module jdk.naming.rmi. If we take a look in the jdk.naming.rmi module, we find there a class called rmiUrlContext that imports the class com.sun.jndi.toolkit.url.GenericURLContext from module java.naming. This is why module java.naming needs to specify that it exports the package to module jdk.naming.rmi.
- The `contains` clauses list all the packages from the module that aren't part of the standard API.

Chapter 4 explains in detail what the `exports`, `requires`, `uses`, and `provides` clauses mean and how they can be used.

Module java.base

With more than 5,600 classes and a size of more than 43 MB, java.base is one of the largest modules of the Java Platform Module System. Every module depends on java.base by default, as the java.base module is located at the bottom of the module graph, representing the core of the system.

The module java.base contains the core APIs and encapsulates the Java runtime. It's not mandatory for a module to explicitly declare that it depends on module java.base because it depends on it automatically. Writing `requires java.base` is allowed, but it's not necessary because the compiler inserts it anyway by default.

The module java.base contains the following packages:

- java.io
- java.lang.*
- java.math
- java.net.*
- java.nio.*
- java.security.*
- java.text.*
- java.time.*
- java.util.*
- javax.crypto.*
- javax.net.*
- javax.security.*

■ **Note** I used *to indicate the package and all its corresponding sub-packages.

The module java.base represents the root of the module system because it contains classes like `java.lang.Object`, `java.lang.Class`, `java.lang.String`, `java.lang.System`, and reflection classes like `java.lang.reflect.Constructor` and `java.lang.reflect.Method`.

All the Java platform packages are exported by the java.base module, meaning they're accessible to any other modules:

```
// module-info.java (module java.base)
module java.base {
        exports java.io;
        exports java.lang;
        exports java.lang.module;
        ...
        ...
        exports java.text;
        exports java.time;
        exports java.util;
        exports javax.net;
}
```

The module java.base, called the base module of the system, depends upon no other module. We say that it has no dependencies upon other modules, meaning that its module-info.java file contains no `requires` clauses. Module java.base also contains the new java.lang.module package introduced in JDK 9, which is part of the new module API. Chapter 9 covers the java.lang.module package. It's also important to remember that java.base is not an aggregator module.

There will be situations when java.base will be the only module you need in order to compile and run a simple Java application. If all the types you need are in packages contained in the java.base module, then java.base is all you need. Due to the fact that java.base has no dependencies on other modules, there aren't any other modules that have to be utilized together with java.base. This is a great advantage because before JDK 9 we had to take the entire JDK since even a simple class like `java.lang.Object` previously had to be used with a large number of classes. A module also contains a new file called module-info.java, which represents the module descriptor, which Chapter 4 describes in detail.

In addition, java.base contains lots of JDK internal packages like jdk.internal.util, sun.io, sun.text, sun.util, com.sun.crypto.provider, com.sun.net.ssl, and more.

■ **Note** Use the Java launcher with the option `--describe-module java.base` to find out all the available information related to the java.base module descriptor.

We've seen how the JDK was modularized and what the module graph looks like. In the next section, you'll learn about the modularization of the source code.

Modular Source Code

Whereas JEP 200's role is to divide the JDK into a set of modules, JEP 201's role is, according to Open JDK, "to reorganize the source code in the JDK into modules, enhance the build system to compile modules, and enforce module boundaries at build time." The layout of the source code was entirely changed in JDK and replaced by modules. The whole source code of a module is now inside a single directory. Therefore, in JDK 9 we have a new scheme of the source code, and that's the focus of this section.

New Scheme for the Source Code

Figure 3-2 illustrates the new scheme of the source code in the JDK.

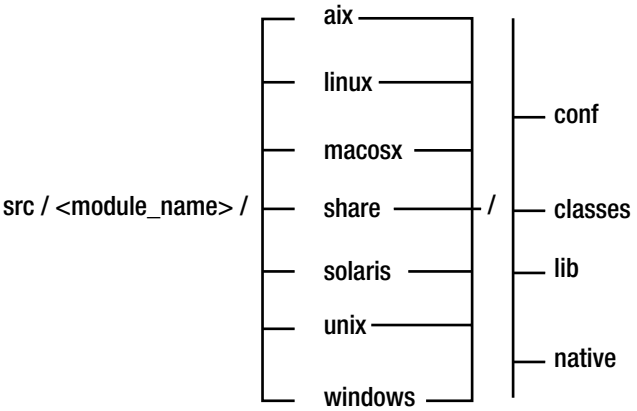

Figure 3-2. *New scheme of JDK 9 source code*

The src directory of the JDK contains a list of directories that represent modules names. Each module has its own directory. The directories' modules names start with java.* or jdk.* and represent the names of the modules.

Every module directory contains a share directory, which consists of cross-platform source code. Additionally, a module directory can also contain other directories related to operating systems, like aix, linux, macosx, solaris, unix, and windows. These directories contain source code pertaining to a single operating system only. Not all the modules contain all the operating-system directories just listed. There are modules that contain operating-system-specific source code for all the operating systems (like java.base or java.desktop) or only for a part of them. For instance, the module java.prefs consists only of three operating-system directories: macosx, unix, and windows—meaning that this module doesn't have any source code specific to Linux or Solaris inside it.

■ **Note** The aggregator modules like java.se.ee and java.se don't have any operating-system-specific source code because they don't have any source code inside their directories.

The next level of directories inside a module contains directories with names like classes, conf, lib, native, and doc. Except for the conf directories, the other four directories can also be found in JDK 8 under the share directory.

The classes directories in JDK 9 consist of Java source files grouped into directories that designate the structure of their packages. We want to point out two main differences to JDK 8: first, in JDK 9 there's a classes directory for each of the existing modules. Even the aggregator modules have a classes directory. Second, in JDK 9 a module-info.java file is placed in the root of the classes directory. The module-info.java file represents the module descriptor and was introduced in Java 9. Each module has a module-info.java file inside its classes directory. The classes directory can contain java, javax, jdk, sun, com, or org directories, depending on the packages contained in it.

The conf directory includes configuration files, which may be properties files, security policy files, policy files, and so on. This is a new directory that didn't exist in JDK 8.

The lib directory is present only in java.base and consists of the file default.policy in the directories share, solaris, and windows.

The native directory holds C and C++ source files, native classes, and procedures. It can contain some of the following directories: include, launcher, common, libfdlibm, libjava, libjimage, libjli, libnet, libnio, libverify, libzip, and more. According to Open JDK, the names of the directories correspond "to the names of the shared libraries into which the compiled code will be linked." The include folder represents an exception to this rule because it has C/C++ header files in it.

■ **Note** The classes and native directories weren't renamed in JDK 9 because doing so could create confusion and slow the adoption of JDK 9.

Table 3-3 shows where some of the most important classes in Java are now located.

Table 3-3. *Location of Some of the Most Important Java Classes*

Class name	Location
java.lang.Object	src / java.base / share / classes / java / lang
java.lang.String	src / java.base / share / classes / java / lang
java.lang.Exception	src / java.base / share / classes / java / lang
java.lang.Class	src / java.base / share / classes / java / lang
java.util.ArrayList	src / java.base / share / classes / java / util
java.util.Date	src / java.base / share / classes / java / util
java.io.File	src / java.base / share / classes / java / io
java.net.URL	src / java.base / share / classes / java / net
java.text.Format	src / java.base / share / classes / java / text
java.util.logging.Logger	src / java.logging / share / classes / java / util / logging
java.sql.DriverManager	src / java.sql / share / classes / java / sql

■ **Note** As you can see in Table 3-3, the most important and useful Java classes are in module java.base, the base module.

The next subsection describes the new structure of the source code in JDK 9.

Comparison Source Code Structure

Here's a short comparison between the structure of the source code in JDK 8 and JDK 9. In JDK 8, the structure of the source code looks like this:

```
jdk / src / share / {back; bin; classes; demo; doc; instrument; javavm; lib; native; npt;
sample; transport}
```

In JDK 9, the structure of the source looks like this:

```
jdk / src / <module_name> / share / {classes; conf; lib; native; doc}
```

As you can see, an intermediary directory containing the module name was added between the directories src and share. Another important difference between JDK 8 and 9 is that in JDK 9 for each classes directory of each module there's a module-info.java file at the root level. In JDK 9, under the classes directory and its subdirectories are only the packages and the classes that belong to the corresponding module. In JDK 8, the classes directory and its subdirectories contain all the packages and classes that comprise the Java platform.

Build Process Adjustments

Not only the source code, but the build itself was organized around modules. There are a new proposed layout and new build targets. The output emitted during the build is different in Java 9 compared to previous versions. The build was changed so that it builds everything as modules. Besides that, the make files have been split into module specific files.

Table 3-4 shows the new structure of the build system in JDK 9 compared to the one that existed in JDK 8.

Table 3-4. *Comparison Between the Structure of the Build System in JDK 8 and JDK 9*

Structure of the Build System in JDK 8	Structure of the Build System in JDK 9
jdk / classes / *.class	jdk / modules / <module_name> / *.class

Table 3-5 shows a list of target commands together with their descriptions that are used to build the JDK 9. These are described in the official JDK 9 API specification.

Table 3-5. *Target commands Used to Build the JDK 9*

Target Command	Description
`make java`	Compiles all Java classes from the system
`make java.sql`	Compiles Java code as well as native code in the java.sql module together with all its dependencies
`make java.sql-java`	Compiles only the Java classes in the java.sql module together with all its dependencies
`make [default]`	Compiles everything
`make all`	Builds everything (JARs, docs, images, and so on), executes a verification tool on the Java classes which finds broken module boundaries
`make images`	Same functionality as JDK 8
`make hotspot`	Same functionality as JDK 8
`make docs`	Builds the entire documentation
`make docs-javadocs`	Builds only the javadoc
`make gensrc`	Executes all the steps involving the generation of source code

The prospect of compiling only one module at a time is one of the most important changes to the build system. A module can be compiled together with its dependencies, and the compiled classes are divided into modules. During the build process, the modules that are independent can be compiled at the same time. If module boundaries are violated, the build process won't succeed.

The first module to be compiled is the java.base module because it's required by all other modules. The module graph is traversed in a reversed order (starting from bottom to top) during the compilation. This makes sense because by compiling the modules successively from java.base up to the top of the module graph, we avoid the situation of compiling a module that has a dependency on a module that has not been compiled.

■ **Note** A great advantage of being able to compile modules lies in the fact that the source code in the JAX-WS, JAXP, and CORBA repositories can now use the new Java language APIs. In the Java versions prior to version 9, that wasn't possible because those repositories were compiled before the JDK repository.

The JCP team performed significant modifications to the build system in order for modules to be built independently. For instance, a change performed in the java.logging module won't determine a new build of the java.base module. This is a great achievement because it boosts productivity.

Summary

This chapter covered two of the most important JEPs of the Java Platform Module System: the Modular SDK and the Modular Source Code.

First, we showed how to list all the modules of the Java runtime system using the `--list-modules` command-line option. We then gave a brief explanation of the standard modules that are part of the Java Platform Standard Edition 9 API Specification. We explained what the platform modules are and talked about the characteristics of the standard and non-standard modules that comprise the Java Platform Module System. We also showed the new JDK module graph that resulted after the JDK modularization. And we showed how to get the entire content of the module descriptor of module java.naming using the `--describe-module <module_name>` command-line option. The end of the first part discussed the module java.base, the most important module of the Java Platform Module System.

The second part of the chapter focused on describing the changes performed at the source code level due to the implementation of JEP 201 (the Modular Source Code). We started by presenting the new scheme of the JDK 9 source code and then described how the source code was organized into directories and what each directory represents. We pointed out the differences between the source code layout in JDK 8 and JDK 9. The chapter finished by talking about the way the build system was enhanced in order to meet the requirements of the newly introduced modules that are now first-class components of the Java platform.

Chapter 4 explains what a module is and shows how you can define and use your own modules.

CHAPTER 4

Defining and Using Modules

In this chapter we'll start to develop modular applications in Java 9 using some of the features offered by Project Jigsaw. We'll begin by explaining the new concept of a Jigsaw module together with its module declaration, the module-info.java file. You'll also learn about the five types of directives that can be used inside the module declaration: `requires`, `exports`, `uses`, `provides`, and `opens`. Then we'll look at compiling and running modules using JDK 9, and for that we'll introduce the new module path in detail.

The accessibility changes introduced in Java 9 are also covered in this chapter. They have a great impact on the platform because they differ almost entirely from the old accessibility rules that were in place before in Java. In Java 9 we can have more types of modules: normal modules, automatic modules, named modules, observable modules, open modules, and unnamed modules. Each is briefly covered in this chapter.

The Concept of Module

As you know by now, Java 9 introduces a new sort of first-class components called *modules*. A Jigsaw module, also a fundamental part of the Java 9 Platform, represents a container of packages. It contains packages, resource files, and native code. The packages can contain Java classes, enumerations, and interfaces.

Figure 4-1 displays the general structure of a module.

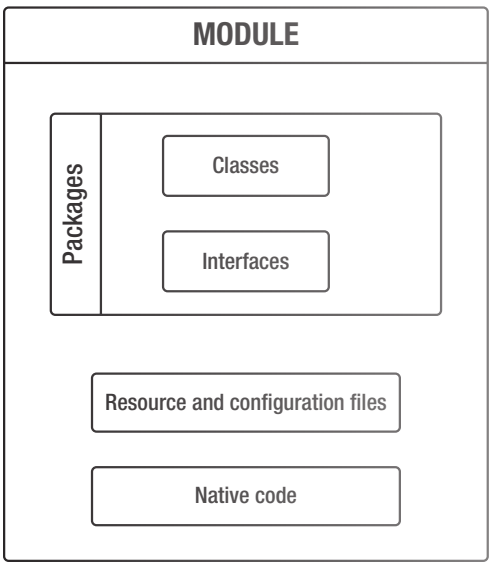

Figure 4-1. *General structure of a module*

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_4

A module consists of the source files together with the module declaration represented by the module-info.java file. Here's a typical directory structure of a module named com.apress.moduleA:

```
src/
    com.apress.moduleA/
                            module-info.java
                            com/apress/moduleA/
                                            Main.java
                                            // other files
```

A directory having the same name with the name of the module is located right at the top. Inside it is the module-info.java file as well as a structure of directories representing the format of the package. In our case, the package has the same name as the module name. The .java files are located inside the directories of the package.

■ **Note** It's also possible to define a module that has no source files and no packages except the module descriptor module-info.java.

We mentioned the module declaration module-info.java. In the next subsection you'll learn about it.

Module Declaration

Each module has a module declaration located into a special new file called module-info.java, located in the top level of the directory. To define a module in Java 9, we create the file module-info.java and put inside it the new keyword `module` followed by the module name and the module declaration in curly brackets.

Declaring a module is easy and straightforward. In this example, a module called com.apress.moduleA is declared inside the module-info.java file:

```
module com.apress.moduleA {}
```

In this case, the module declaration doesn't contain anything besides the module heading. Module com.apress.moduleA doesn't require any modules, nor does it export any packages, and it doesn't provide or consume any services.

As already mentioned, each module must have a module declaration and thus must contain an own module-info.java file. This rule applies no matter whether it's a platform module or a module created by developers. If the module-info.java file isn't present, the Java compiler doesn't treat the source code as a module. The module-info.java file gets compiled in exactly the same way as a Java file.

If we change the name of the module-info.java file to something else, the compiler will interpret the file as a normal file rather than as a module descriptor. In this case, the module system won't be able to recognize the module anymore.

■ **Note** In order to move the source code into a module, a module descriptor is mandatory. Otherwise, the source code won't be part of a module.

Let's describe some cases to see what we can put in a module-info.java file and what we can't. Module-info.java can't contain anything besides the module definition. The Java compiler can recognize syntax errors in the module declaration.

If the module declaration isn't in the module-info.java file, the following error message will be displayed by the Java compiler, and the compilation will fail:

```
Error: module declarations should be in a file named module-info.java
```

A Java class can't be put into the module-info.java file instead of a module. If you try to do so, the compilation will also fail:

```
error: cannot access module-info
  bad source file: src\com.apress.moduleA\module-info.java
    file does not contain module declaration
    Please remove or make sure it appears in the correct subdirectory of the sourcepath.
```

Besides that, attempting to write two module declarations in a single module-info.java file will also result in a compilation error. As we already observed in the examples presented earlier, the Java compiler gives concrete indications about the cause and the location of the errors. In this way, we can go directly to the line of code that generates the issues and provide a fix for it.

The module-info.java module descriptor is compiled together with the source code. As a result, .class files, including a module-info.class file, are generated. All these compiled files can be packaged as a modular JAR file (we cover modular JAR files later in this chapter). The compiler treats module-info.java like any other Java file and translates it into a module-info.class file that we can put in a JAR file. The result is a modular JAR.

■ **Note** The name of file module-info.java was chosen by the JCP team after the already existing name of the package-info.java file. The compiler can make use of the module-info.java file even if it contains an illegal Java identifier (the dash) in its name definition.

Platform modules consist of a module-info.java file by default. If we create our own module, in most cases we have to create and write the content of the module-info.java file on our own. But there are two cases when we don't write a module-info.java file on our own:

- When we put a JAR file on the module path, a module-info.java file will be automatically generated.
- When we automatically generate a module-info.java file for a specific JAR file using the JDeps tool and the option `--generate-module-info`.

Don't worry if notions like *module path* and *JDeps* are unfamiliar to you. You'll find out later in this book what they are.

Module Name

Specifying a name for the module is mandatory. Two modules within the same code base can't have the same name. It's good practice to name our modules the same way we name packages: by using the domain names in reverse order. The name of the module could therefore be a prefix of the names of its exported packages, but we can name our modules how we want because we don't have any constraints regarding the format of the module name. Nevertheless, the name of the module complies with the general rules of identifiers in Java. A module can have the same name as a Java class or an interface, because the names of the modules have their own namespace.

■ **Note** There is an exception to the rule: when compiling multiple modules at the same time, it's mandatory that the module name has the same name as the directory where the module descriptor module-info.java is located.

Inside a module declaration we can have a total of five types of clauses, discussed next.

Five Types of Clauses

A module declaration can consist of up to five types of clauses:

- `requires` clauses specify the module that's required by the current module.
- `exports` clauses specify the packages that are exported by the current module
- `provides` clauses specify the service implementations that the current module provides
- `uses` clauses specify the services that the current module consumes
- `opens` clauses specify the packages that the current module opens for deep reflection

Table 4-1 describes the syntax of these five clauses.

Table 4-1. *The Five Clauses from a Module Descriptor*

Directive Keyword	Description
`requires <module_name>`	Expresses which other modules the current module depends on.
`exports <package_name> (to <module_name>)`	Expresses which packages from the current module are exported outside the module. The optional to clause lists the modules to which the packages are exported.
`opens <package_name>`	Makes the `<package_name>` available for deep reflection at runtime.
`provides <service_name> with <service_name_implementation>`	Specifies that the current module provide the implementation of `<service_name>` with `<service_name_implementation>`.
`uses <service_type>`	Specifies that the current module consumes instances of `<service_type>`.

This chapter covers only the first three clauses: `requires`, `exports`, and `opens`. The last two, `provides` and `uses`, are covered in Chapter 6 because they're related to services.

Let's continue by exploring in detail the most common clauses typically used inside a module declaration: the `requires` clause and the `exports` clause.

The requires Clause

The `requires` clause is used inside the module declaration (module-info.java) to express the module that the actual module needs upon in order to fulfill its dependencies. It's used to express the module's dependencies.

Figure 4-2 shows the syntax of the `requires` clause.

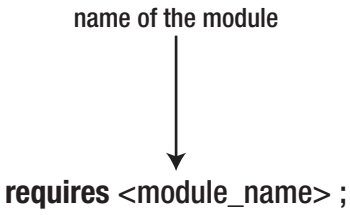

Figure 4-2. *Syntax of the requires clause*

The syntax is simple and concise. The `requires` directive specifies the name of the module it depends upon, followed by a semicolon. Inside the curly brackets of the module declaration we can put one or more `requires` clauses, each of them followed by the name of the module.

In the following example, the module com.apress.moduleA requires two modules, module com.apress.moduleB and module com.apress.moduleC:

```
module com.apress.moduleA {
        requires com.apress.moduleB;
        requires com.apress.moduleC;
}
```

In this example, two dependencies are expressed using the `requires` clauses. Module com.apress.moduleA has a dependency on module com.apress.moduleB and also has a dependency on module com.apress.moduleC. In this case, we say that module com.apress.moduleA requires (or reads) module com.apress.moduleB and requires (or reads) module com.apress.moduleC.

Figure 4-3 shows a module graph that illustrates these dependencies.

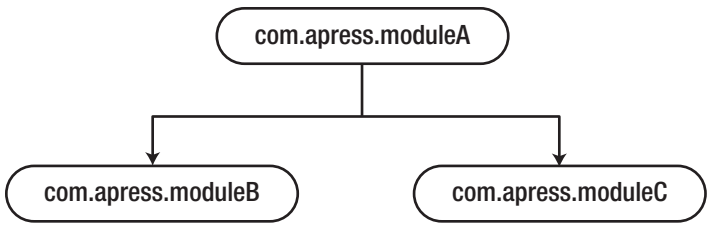

Figure 4-3. *Module graph expressing dependencies between the three modules*

In the module graph we have an arrow from module com.apress.moduleA to module com.apress.moduleB and also an arrow to module com.apress.moduleC. There is no arrow between module com.apress.moduleB and module com.apress.moduleC because those two modules don't have dependencies between them. The direction of the arrow is straightforward: from com.apress.moduleA to com.apress.moduleB, because module com.apress.moduleA reads the module com.apress.moduleB and not inversely.

What does it mean that module com.apress.moduleA has a dependency on module com.apress.moduleB and on module com.apress.moduleC? It means that module com.apress.moduleA uses types that are part of module com.apress.moduleB and of module com.apress.moduleC. Because module com.apress.moduleA uses types from those modules, it has dependencies on them that must be explicit declared in the module descriptor module-info.java. In this way, the Java compiler knows at compile-time what the dependencies of a module are and doesn't allow the compilation if a single dependency is not fulfilled.

■ **Note** Compared to the class path, the situation when a module dependency isn't fulfilled is detected right at compile-time using the Java 9 Module System. It would have not been allowed for module com.apress.moduleB to also read module com.apress.moduleA at the same time at compile-time. We would then have had a circular dependency, which is in JDK 9 forbidden by the Java compiler.

If we try to run module com.apress.moduleA, a resolution is first performed. A *resolution* represents a process that searches and discovers the modules required by a module. All the modules found on the host system are being searched, and the modules found are searched again for dependencies. This process continues and runs until every required module has been covered and until every dependency of every required module has been solved. In our case, the resolution is simple, because module com.apress.moduleA requires only two modules: module com.apress.moduleB and module com.apress.moduleC. We suppose that these last two modules don't have any dependencies upon other modules. In this case, the resolution process is successfully finished after all three modules are added to the module graph. For instance, if module com.apress.moduleB has had other dependencies, these would have been resolved and also added to the module graph. The result of the resolution process contains the entire data required for compiling and running the root module, com.apress.moduleA.

Every module implicitly requires java.base, as we already know. Mentioning `requires java.base` in the module descriptor is unnecessary because module java.base is by default required by every module. The module java.base will always be located right at the bottom of the module graph because every module depends upon it. Figure 4-4 shows the previous module graph with module java.base included at the bottom.

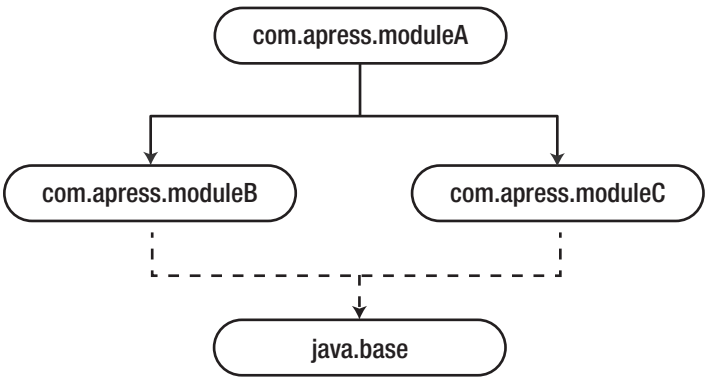

Figure 4-4. *Module graph expressing dependencies between modules, including module java.base*

If a module descriptor doesn't contain any `requires` clauses, the module doesn't have any dependency on any module except for module java.base. Module java.base doesn't have any `requires` directives because it doesn't depend upon any other module.

Until now, we've looked at only the positive cases. Let's also explore some cases when something goes wrong and the compilation fails. The compilation will fail if the module used in the `requires` clause isn't found. The following module declaration states that it requires module com.apress.moduleB, but if this module hasn't been defined, compiling com.apress.moduleA will result in an error, because its dependencies can't be fulfilled:

```
module com.apress.moduleA {
        requires com.apress.moduleB;
}
```

We state that a module isn't defined if it doesn't have a module-info.java file or if its module-info.java file doesn't contain the right name of the module. In the previous example, the output of the compilation of module com.apress.moduleA results in an error:

```
error: module not found: com.apress.moduleB
```

We get the same results if we have a qualified export to a module that isn't found.

Loops in module declarations aren't allowed, as in the following example:

```
module com.apress.moduleA {
        requires com.apress.moduleA;
}
```

This module declaration has a cyclic dependence and results in a compilation error:

```
error: cyclic dependence involving com.apress.moduleA requires com.apress.moduleA
```

■ **Note** Cyclic dependencies aren't allowed by the module system at compile-time.

Listing 4-1 shows an example of a simple circular dependency between three modules.

Listing 4-1. Defining Three Module Descriptors for Three Distinct Modules

```
// module-info.java (module com.apress.moduleA)
module com.apress.moduleA {
        requires com.apress.moduleB;
}

// module-info.java (module com.apress.moduleB)
module com.apress.moduleB {
        requires com.apress.moduleC;
}

// module-info.java (module com.apress.moduleC)
module com.apress.moduleC {
        requires com.apress.moduleA;
}
```

A circular dependency is present because of the following conditions:

- module com.apress.moduleA depends on module com.apress.moduleB.
- module com.apress.moduleB depends on module com.apress.moduleC.
- module com.apress.moduleC depends on module com.apress.moduleA.

The compilation of these three modules fails because it results in the same cyclic dependence error as in the previous example.

Every `requires` statement must contain only one module name. It can't enumerate two module names using a comma in a single `requires` statement. In this case, we'll get an error during compilation. It's also forbidden to duplicate two `requires` statements in the same module declaration. The compilation error message would then be as follows:

```
error: duplicate requires: <module_name>
```

What happens when a module depends on other module but doesn't declare this dependency inside its module descriptor? In this next example, the descriptor for module com.apress.moduleA reflects that it doesn't require any other module. Listing 4-2 shows the module descriptor of this module:

Listing 4-2. The Module Descriptor of module com.apress.moduleA

```
// module-info.java
module com.apress.moduleA {

}
```

In Listing 4-3, module com.apress.moduleA contains a class called `Main` that imports and makes use of types from module com.apress.moduleB.

Listing 4-3. The Main Class of module com.apress.moduleA

```
// Main.java (module com.apress.moduleA)
package com.apress.moduleA;
import com.apress.moduleB.*;

public class Main {
        public static void main(String[] args) {
                Employee employee = new Employee("John", "Albert");
                System.out.println("First name is : " + employee.getFirstName());
                System.out.println("Last name is : " + employee.getLastName());
        }
}
```

Listing 4-4 defines the module com.apress.moduleB, which has an empty module declaration.

Listing 4-4. The Module Descriptor of module com.apress.moduleB

```
// module-info.java
module com.apress.moduleB {

}
```

Listing 4-5 defines a POJO class as part of the com.apress.moduleB module.

Listing 4-5. Class Employee from module com.apress.moduleB

```
// Employee.java (module com.apress.moduleB)
package com.apress.moduleB;
public class Employee {

        private String firstName;
        private String lastName;

        public Employee() {
        }
```

```
        public Employee(String firstName, String lastName) {
                this.firstName = firstName;
                this.lastName = lastName;
        }

        public String getFirstName() {
                return firstName;
        }

        public String getLastName() {
                return lastName;
        }
}
```

We compile all the .java files from both modules at the same time using the following command:

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

■ **Note** This compilation is done using Cygwin in Windows. Cygwin is a Unix-like command-line interface that runs in Windows. Throughout this book all the operations are performed using Cygwin.

For compilation we use the `--module-source-path` command-line option in order to indicate to `javac` the location of the source code of the modules. In our example, the `--module-source-path src` option defines that the subdirectories of the src directory comprise the code for various modules.

The compilation fails, and we're informed that the package com.apress.moduleB doesn't exist:

```
.\src\com.apress.moduleA\com\apress\moduleA\Main.java:3: error: package com.apress.moduleB
does not exist
import com.apress.moduleB.*;
^
.\src\com.apress.moduleA\com\apress\moduleA\Main.java:8: error: cannot find symbol
                Employee employee = new Employee("John", "Albert");
                ^
  symbol:   class Employee
  location: class Main
.\src\com.apress.moduleA\com\apress\moduleA\Main.java:8: error: cannot find symbol
                Employee employee = new Employee("John", "Albert");
                                        ^
  symbol:   class Employee
  location: class Main
3 errors
```

Module com.apress.moduleA has an empty module declaration. It doesn't require any other module, so it can't access types from other modules according to the strong encapsulation mechanism introduced in Jigsaw. This is why attempting to access types from the module com.apress.moduleB inside com.apress.moduleA results in a compilation error.

■ **Note** You can find the source code for this example in the directory /ch04/requiresClause.

Let's edit module-info.java of the module com.apress.moduleA and add the dependency to module com.apress.moduleB. Listing 4-6 shows its new module descriptor.

Listing 4-6. The Module Descriptor of module com.apress.moduleA

```
// module-info.java
module com.apress.moduleA {
        requires com.apress.moduleB;
}
```

Now we try to compile the source code again using the same options. Unfortunately, exactly the same compilation error as the one that we previously had. That's because in Java 9 it's not enough to specify that a module requires another module in order to access types from that module.

Additionally, the second module must export some of its types in order to make them accessible to the modules that depend upon it. In our case, for the compilation to successfully work, we must modify module-info.java of module com.apress.moduleB and specify that it exports all the types from package com.apress.moduleB. Listing 4-7 shows the new definition of its module-info.java file.

Listing 4-7. The Module Descriptor of Module com.apress.moduleB

```
// module-info.java
module com.apress.moduleB {
        exports com.apress.moduleB;
}
```

To recap, we've learned up to now how to define dependencies upon other modules using the `requires` clause. The property of a module to specify the modules that it requires represents the base of reliable configuration. Until now we've used only the simple form of the `requires` clause. Hence, the `requires` clause can also include the `static` keyword as well as the `transitive` keyword. We explain the `requires transitive` clause later in the "Accessibility" section.

The `requires static myModule` clause indicates that the module myModule should be present only at compile-time. At runtime, its presence isn't mandatory. In this way, we must have a compile-time dependency, but no runtime dependency.

Let's find out how to make a module express that it makes its packages available for other modules that depend upon it. The next section explains how the `exports` clause can be used inside the module declaration.

The exports Clause

The `exports` clause has the role of exporting a package at compile-time as well as at runtime. It allows a module to specify which packages it exports. Only an `exported` package can be available to other modules, provided that the other conditions regarding reliable configuration are met. The reverse is also true. A package that isn't exported isn't available for any other modules.

■ **Note** A module doesn't export any package by default. This means that by default, no package from the current module is available to other modules for access.

The `exports` clause is specified in the module descriptor by using the keyword `exports` followed by the package name. It's forbidden to separate packages or modules using a comma. For each package, a separate `exports` clause must exist.

Figure 4-5 shows the syntax of the `exports` clause.

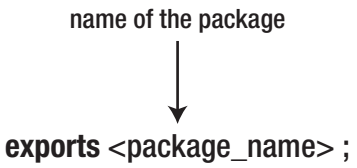

Figure 4-5. *Syntax of the exports clause*

Similar to the `requires` clause, the `exports` clause has some constraints. For example, duplicating the `exports` statements inside the module declaration isn't allowed and results in a compilation error:

```
error: duplicate export: <module_name>
```

We suppose that module com.apress.moduleB wants to make two of its packages, com.apress.moduleB.packageB1 and com.apress.moduleB.packageB2, available to other modules that require it. In this case we say that module com.apress.moduleB exports those packages. Listing 4-8 illustrates this in its module declaration.

Listing 4-8. The module-info.java of module com.apress.moduleB

```
module com.apress.moduleB {
        exports com.apress.moduleB.packageB1;
        exports com.apress.moduleB.packageB2;
}
```

Listing 4-9 states that module com.apress.moduleC also exports a package called com.apress.moduleC.packageC1.

Listing 4-9. The Module Descriptor of module com.apress.moduleC

```
module com.apress.moduleC {
        exports com.apress.moduleC.packageC1;
}
```

In the following module graph, module com.apress.moduleB has two packages that are both exported. Module com.apress.moduleC also consists of two packages, but only one is exported according to its module definition. Package com.apress.moduleC.packageC2 isn't exported and therefore will never be accessible outside the module com.apress.moduleC. Any other module attempting to make use of package com.apress.moduleC.packageC2 will not only fail, it will also not be compiled successfully.

Figure 4-6 shows a new enhanced type of the module graph where we have also inserted the packages that the module contains. The packages com.apress.moduleB.packageB1, com.apress.moduleB.packageB2 and com.apress.moduleC.packageC1 are exported, and the com.apress.moduleC.packageC2 package is not exported.

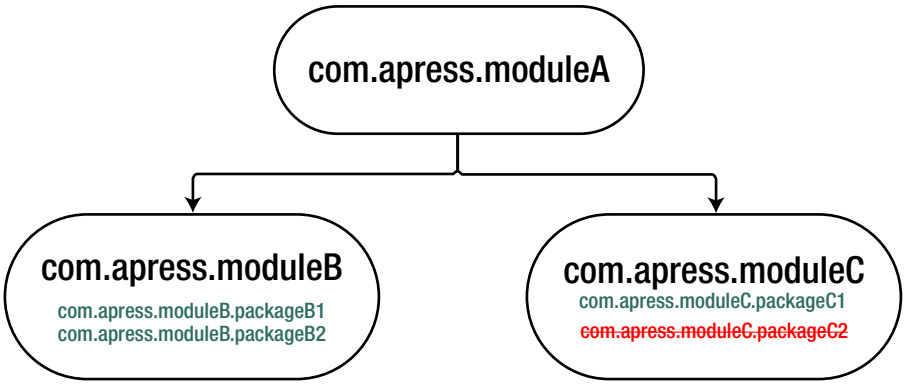

Figure 4-6. *Module graph showing which packages are exported and which aren't*

Module com.apress.moduleA can successfully access types in both packages packageB1 and packagesB2 for the following reasons:

- It reads module com.apress.moduleB.
- The packages packageB1 and packageB2 are being exported by module com.apress.moduleB.

However, if a new package were added into module com.apress.moduleB, it would not be accessible to module com.apress.moduleA unless it were declared as exported in the module description of module com.apress.moduleB. Module com.apress.moduleA can access types from com.apress.moduleC, but only from package packageC1 because this is the only package that is being exported by module com.apress.moduleC. Package packageC2 is not being exported and therefore can't be accessed by module com.apress.moduleA.

We've learned so far how to set up a module declaration file and how to use the `requires` and `exports` clauses. Listing 4-10 presents a simple module with both `requires` and `exports` clauses.

Listing 4-10. The Module Descriptor of module com.apress.moduleA Using Both requires and exports Clauses

```
module com.apress.moduleA {
    requires com.apress.moduleB;
    exports com.apress.moduleA.packageP1;
}
```

In this example, we defined a module called com.apress.moduleA that depends upon another module called com.apress.moduleB and also exports the package called com.apress.moduleA.packageP1.

The opens Clause

Until now we've seen how we can achieve strong encapsulation using the `exports` directive. But what happens if we need to access some types using reflection?

■ **Note** The `exports` clause just shown doesn't allow its non-public types to become accessible by using deep reflection.

There are two different situations for using reflection in Java 9:

- Code in the unnamed module (the class path) can access code in any named modules using reflection. This is possible due to a flag called `--illegal-access` that's set by default and that was added by the JCP team in order to ease migration. Many frameworks such as Hibernate and JPA need reflective access to code in named modules. These frameworks typically reside on the class path. Chapter 8 discusses the `--illegal-access` flag in more detail.
- Code in a named module can't access code in any named modules using reflection.

This is where the new `opens` clause and the `--add-opens` command-line option come in play.

In order to solve this problem, a new directive called `opens` was introduced. Its role is to provide reflective access to the types passed as parameter.

■ **Note** Using `opens` to export a package that has some internal implementation isn't recommended. Doing so will expose the internal implementation and break the strong encapsulation rules.

Figure 4-7 illustrates the syntax of the opens clause.

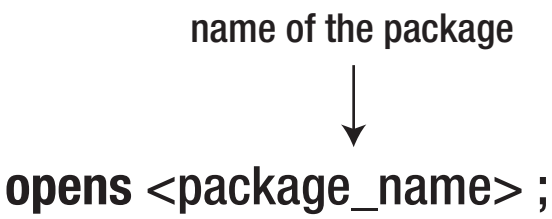

Figure 4-7. *The syntax of the opens clause*

The `opens` clause is another clause that can exist in a module descriptor, besides the `requires` and `exports` clauses discussed earlier in this chapter. The `opens` clause is used inside the module declaration to define the packages that are available for deep reflection at runtime for all modules. Therefore, both the public and the private types of the package are accessible using deep reflection by code in other modules.

■ **Note** Packages inside a module are by default available for deep reflection only by code in any unnamed module.

The opens directive can also be qualified by specifying a list of target named modules, as illustrated in Figure 4-8.

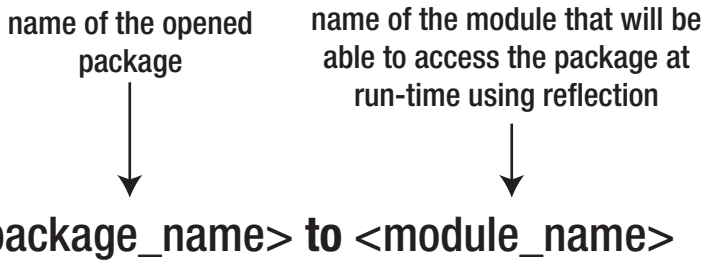

Figure 4-8. *The syntax of the qualified opens clause*

There are some characteristics of the opens clauses you should know. First, it's important to know that it is possible to use both exports and opens directives for the same package. In this case, the package is exported for access at compile-time and runtime and also available for deep reflection at runtime. As a result, its public types can be accessed at compile-time and at runtime, and both its public and private types can be accessed at runtime using reflection. Second, the opens directive can't be used inside an open module. Open modules are covered later in this chapter.

■ **Note** The opens directive can't use wildcards and cannot contain more than one package.

Other Clauses

The other two directives that can be used inside a module descriptor are the uses and provides clauses. These clauses are used to define (provides) and consume (uses) services and will be described in Chapter 6 – Services.

■ **Note** None of the five directives already mentioned (requires, exports, opens, uses, and provides) are mandatory inside a module declaration. There is no restriction in selecting which directive to use and which not to use. We can create our own module and use any combination of the five directives.

We've now seen the basics of the module declaration. It's time to learn how to compile and run modular applications in JDK 9.

Compiling and Running Modules

The compilation of a modular Java 9 application is different than in Java 8 or 7. All the examples of compiling and running modules are expressed in this book using the command-line running a Linux-like environment. However, this is an unusual practice in day-to-day work. Build tools like Maven or Gradle are far more productive, suitable, and easy to use for compling and running Java applications. Chapter 12 covers the integration of Java 9 with Maven and shows how to use it to build, package, compile, and run Java 9 modular applications.

Compile a Single Module

Here's a very simple example of compiling a single module using JDK 9. Suppose we have a module com.apress.moduleA that has no dependencies. It contains a Main.java file inside its package com.apress.moduleA. The structure of the folder is as follows:

```
src/com.apress.moduleA/
                module-info.java
                com/
                    apress/
                          moduleA/
                                Main.java
```

Listing 4-11 shows the content of class `Main`. It prints a message on the console.

Listing 4-11. The Main class from the module com.apress.moduleA

```
// Main.java
package com.apress.moduleA;
public class Main {
        public static void main(String[] args) {
                System.out.println("Here is Java 9!");
        }
}
```

Listing 4-12 shows the module-info.java file of module com.apress.moduleA. It doesn't define any clause.

Listing 4-12. The Module Descriptor for moduleA

```
// module-info.java
module com.apress.moduleA {

}
```

First we create the destination directory where the compiler outputs to using the `mkdir` command:

```
mkdir -p outputDir
```

Then we use the Java compiler to compile the Main.java file and the module-info.java file. The compilation will create a .class file for each .java file:

```
$ javac -d outputDir/com.apress.moduleA src/com.apress.moduleA/module-info.java src/com.
apress.moduleA/com/apress/moduleA/Main.java
```

The `javac` command gets the files that is has to compile. The `-d` option specifies the directory where the compiler outputs to. In this case, it will output to the directory outputDir/com.apress.moduleA. The last two parameters represent the path to the files that we want to compile, module-info.java and Main.java.

The corresponding class files are generated during the compilation in the directory passed as parameter to the option `-d`. The compilation of a single module is done very much the same as in Java 7 or 8 without necessarily needing to use other compiler flags than the ones used in the older versions of Java. Compared to Java 8, the single distinction here is that module-info.java was also compiled.

■ **Note** The module declaration from the module-info.java file is compiled along with the rest of the source code, and a module-info.class file is generated.

Run an Application Containing a Single Module

To run the previously compiled classes, we use the Java launcher with the following command:

```
$ java --module-path outputDir --module com.apress.moduleA/com.apress.moduleA.Main
```

As a result, the string "Here is Java9!" is printed at the console. We recognize in the previous listing that the `java` command uses new flags for handling modules. The `--module-path` option, introduced in Java 9, gets as parameter a directory or a list of directories containing the location of the already compiled files. In our case, these are in the outputDir directory. The module path is used in order for the compiler to be able to locate the modules at runtime.

The differences between the module path and the class path are listed later in this chapter, in the section "The Module Path."

■ **Note** We exploded the files on the file system as .class files. Another possibility would have been to package them as modular JARs. Later in this chapter we show how.

The second option used by the Java launcher is the command-line option `--module`. It's used to specify the main class and the main module by taking a parameter in form of `<module_name> / <main_class>`. The location of the `main` class is mandatory information that the `java` command needs to be aware of.

The Java launcher will load the root module com.apress.moduleA, resolve all its dependencies and transitive dependencies by running the resolution process, and finally run its `Main` class, which was passed to the option `--module`. At the end, the message is printed in the console.

Congratulations! You just learned how to successfully compile and run your first module in Java 9.

If we had tried to run the `java` command without the `--module-path` option, like this

```
$ java –m com.apress.moduleA/com.apress.moduleA.Main
```

the following error message would have been displayed:

```
Error occurred during initialization of VM
java.lang.module.ResolutionException: Module com.apress.moduleA not found
        at java.lang.module.Resolver.fail(java.base@9-ea/Resolver.java:796)
        at java.lang.module.Resolver.resolveRequires(java.base@9-ea/Configuration.java:370)
        at java.lang.module.Configuration.resolveRequiresAndUses(java.base@9-ea/
        ModuleDescriptor.java:2081(
        at jdk.internal.module.ModuleBootstrap.boot(java.base@9-ea/ModuleBootstrap.java:263)
        at java.lang.System.initPhase2(java.base@9-ea/System.java:1925)
```

What does this error mean? Module com.apress.moduleA isn't found because we didn't inform the Java launcher about the location of the compiled modules. The directory where the compiled modules are located has to be specified using the `--module-path` option. The Java launcher can't find the modules unless we explicitly specify their location.

Compile Multiple Modules

Until now, we've compiled and executed a single module. But for the compilation of two or more modules, some extra compiler flags have been introduced in Java 9. In the following example you'll learn how to compile multiple modules at the same time.

Suppose we have a total of three modules: module com.apress.moduleA, module com.apress.moduleB, and module com.apress.moduleC. The structure of the folders for the three modules is like this:

```
src/com.apress.moduleA/
                          module-info.java
                          com/
                                apress/
                                      moduleA/
                                                Main.java
    com.apress.moduleB/
                          module-info.java
                          com/
                                apress/
                                      moduleB/
                                                ClassB1.java
                                                ClassB2.java
    com.apress.moduleC/
                          module-info.java
                          com/
                                apress/
                                      moduleC/
                                                ClassC1.java
                                                ClassC2.java
```

Further, suppose that for each module the module-info.java doesn't contain any clause. Listing 4-13 shows the `javac` command used to compile only the module com.apress.moduleA.

Listing 4-13. Compile module com.apress.moduleA Using the --module-source-path Flag

```
$ javac –d outputDir --module-source-path src src/com.apress.moduleA/module-info.java
src/com.apress.moduleA/com/apress/moduleA/Main.java
```

The compilation of multiple modules at the same time is done using the new `--module-source-path` command-line option introduced in Java 9. It's used to tell `javac` about the sources in the directory. In this case we point the `--module-source-path` to the src directory and output our compiled modules into the outputDir directory.

The compilation generates the following classes inside the outputDir directory:

```
outputDir/com.apress.moduleA/
                                com/
                                    apress/
                                          moduleA/
                                                  Main.class
                                module-info.class
```

The other two modules, com.apress.moduleB and com.apress.moduleC, haven't been generated because we didn't specify them in the `javac` command. We compiled only the module com.apress.moduleA.

Let's specify that module com.apress.moduleA has a dependency on module com.apress.moduleB and on com.apress.moduleC. Listing 4-14 shows the module-info.java file of module com.apress.moduleA where we define the dependency using the `requires` clause.

Listing 4-14. The Module Descriptor of module com.apress.moduleA

```
// module-info.java
module com.apress.moduleA {
        requires com.apress.moduleB;
        requires com.apress.moduleC;
}
```

If we run the `javac` command from Listing 4-13, we get the following structure of the outputDir directory:

```
outputDir/com.apress.moduleA/
                            com/
                                apress/
                                        moduleA/
                                             Main.class
                                module-info.class
          com.apress.moduleB/
                                module-info.class
          com.apress.moduleC/
                                module-info.class
```

In the `javac` command, we specified to compile only the Main.java and the module-info.java files from module com.apress.moduleA. But due to the fact that module com.apress.moduleA requires both modules com.apress.moduleB and com.apress.moduleC, the module descriptors from com.apress.moduleB and com.apress.moduleC have also been compiled to class files.

We compile all the classes that end with .java using the following command:

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

The `--module-source-path` option specifies the location of non-compiled files. We search for all the files that end with the extension .java, including the module descriptor, of course. All the files ending with .java from all the modules have been compiled to .class files. The structure of outputDir is shown in the following code. Both Java classes and the module-info.java files have been compiled to .class files:

```
outputDir/com.apress.moduleA/
                                com/
                                      apress/
                                             moduleA/
                                                      Main.class
                                module-info.class
            com.apress.moduleB/
                                com/
                                      apress/
                                               moduleB/
                                                        ClassB1.class
                                                        ClassB2.class
                                module-info.class
            com.apress.moduleC/
```

```
com/
     apress/
            moduleC/
                                ClassC1.class
                                ClassC2.class
 module-info.class
```

Run an Application Containing Multiple Modules

To run the previously compiled classes, we use the Java launcher with the following command:

```
$ java --module-path outputDir --module com.apress.moduleA/com.apress.moduleA.Main
```

This is the same `java` command used in the previous example, when we ran an application consisting of only one module. But now we have three modules inside the application that we want to run, so why do we specify a single module?

The `--module` option needs to get only the name of the root module. It doesn't need to get all the modules. By getting the root module, it starts a resolution process and finds the other modules according to the information present in the module descriptor.

■ **Note** The `--module` command-line option gets as parameter only the name of the `Main` class from the root module.

The resolution process starts with module com.apress.moduleA and then finds the modules com.apress.moduleB and com.apress.moduleC. After it finishes, the `Main` class of the root module is executed.

In this section, we've learned how to compile and run multiple modules using Java 9. Now let's see an example where compilation fails due to the broken accessibility rules. Here, we import ClassB1 and ClassB2 into Main.java. Listing 4-15 shows the `Main` class of com.apress.moduleA.

Listing 4-15. The Main Class

```
// Main.java
package com.apress.moduleA;
import com.apress.moduleB.ClassB1;
import com.apress.moduleC.ClassC1;
public class Main {
        public static void main(String[] args) {
                System.out.println("Here is Java 9!");
        }
}
```

We know from the previous example that module com.apress.moduleA requires modules com.apress.moduleB and com.apress.moduleC. By compiling all the Java files

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

the following error is thrown:

```
Main.java:3: error: ClassB1 is not visible because package com.apress.moduleB is not visible
import com.apress.moduleB.ClassB1;

Main.java:4: error: ClassC1 is not visible because package com.apress.moduleC is not visible
import com.apress.moduleC.ClassC1;

2 errors
```

The compilation fails as a result of strong encapsulation. The classes `ClassB1` and `ClassC1` can't be imported into the `Main` class. Even if module com.apress.moduleA requires the other two modules, it can't access types from them because those types aren't exported by these modules. The public access modifier used to define the classes `ClassB1` and `ClassC1` isn't able to enforce accessibility any more in Java 9.

In order to make the classes `ClassB1` and `ClassC1` available to the `Main` class, we must explicitly specify the following two things:

- In the module descriptor of module com.apress.moduleB, we specify that the package containing the `ClassB1` is exported.
- In the module descriptor of module com.apress.moduleC, we specify that the packages containing the `ClassC1` is exported.

By doing this, all classes from the exported package from module com.apress.moduleB and from the exported package from module com.apress.moduleC will be accessible in this way to module com.apress.moduleA, and the compilation will succeed without any error.

Private vs. Public Methods

We extend our example to be able to call a method from a different module. A private method won't be able to be accessed from other module even if the module is read and the corresponding types are exported.

■ **Note** It's important to remember that a private method is always package private.

Listing 4-16 shows the class `ClassB1` from package com.apress.moduleB which has a new private static method.

Listing 4-16. Class ClassB1 Contains a Private Method

```
// ClassB1.java
package com.apress.moduleB;
public class ClassB1 {
        private static String getInfoForClassB1() {
                return "ClassB1 from ModuleB";
        }
}
```

Listing 4-17 shows how the `Main` class from module com.apress.moduleA calls the static method from `ClassB1`.

Listing 4-17. Class Main

```
// Main.java
package com.apress.moduleA;
import com.apress.moduleB.ClassB1;
import com.apress.moduleC.ClassC1;

public class Main {
        public static void main(String[] args) {
                System.out.println(“Here is Java 9!”);
                System.out.println(ClassB1.getInfoForClassB1());
        }
}
```

The compilation fails with the following error message:

```
Main.java:10: error: getInfoForClassB1() has private access in ClassB1
```

By setting the `getInfoForClassB1()` method to public in ClassB1.java, the compilation will succeed. Running the application

```
$ java --module-path outputDir –m com.apress.moduleA/com.apress.moduleA.Main
```

results in the following being printed:

```
Here is Java 9!
ClassB1 from ModuleB
```

In this example, we showed that having an access identifier as `private` makes the type inaccessible, as in the previous versions of Java.

We've learned how to compile and run modular applications in Java 9. In the next section we discover the new modular JAR files introduced in Jigsaw.

Modular JARs

The modular JARs were introduced in JDK 9. They represent module artifacts that contain compiled module definitions. A modular JAR resembles a regular JAR and contains .class files and also a module-info.class file. The difference between a modular JAR and a regular JAR consists only of the module-info.class that a modular JAR additionally has.

The module-info.class file lies in a modular JAR at the top level of its directory. Every modular JAR must contain such a file. If it doesn't, it's just a regular JAR file, not a modular one.

■ **Note** The module-info.class is located at the root of the directory, inside the modular JAR file. It's not located inside the packages.

A modular JAR can be used on the module path as well as on the class path. When used on the class path, the module-info.class file isn't taken into consideration.

■ **Note** A modular JAR file is compatible with older versions of the JDK. It works as a regular JAR file on the class path for all Java versions prior to Java 9.

Because the module descriptor is compiled in a module-info.class file in Java 9, a modular JAR can work as a module by being placed on the module path. If we place the modular JAR on the class path, the module-info.class file will be simply ignored. But all the other files of the modular JAR, except the module-info.class file, will be taken into consideration. In this way, the modular JAR will act as a normal JAR file.

It's a great advantage to have a JAR file that can be used either on the class path or on the module path. We could compile a library and be able to use it on the class path for JDK 8 (or earlier) or compile it with JDK 9+ and use it on the module path.

■ **Note** A modular JAR can incorporate only one module. It can't be composed of more than one module.

The alternative to modular JARs would be to explode the compiled modules on the file system. Both solutions work, but it's definitely better and more suitable to have a single modular JAR instead of a group of files.

Let's imagine a situation where the module path contains two modular JAR files that are located in the same directory. If the module that's part of the first modular JAR file has the same name as the module that's part of the second modular JAR file, there will be an error at compile-time.

In order to create a modular JAR, we can use the jar tool, which was enhanced in Java 9. We'll learn more about it later in this section. Next, we talk about the structure of a modular JAR file.

Structure of a Modular JAR

The structure of a modular JAR file is similar to the one of a normal JAR file, except a module-info.class file is present. Here's an example of a modular JAR file:

```
META-INF/
META-INF/MANIFEST.MF
module-info.class
com/apress/moduleA/Main.class
com/apress/moduleB/ClassB1.class
com/apress/moduleB/ClassB2.class
...
```

There is also a MANIFEST.MF file located in the META-INF directory. The module-info.class file is located in the root directory. All our compiled .class files are present in the modular JAR.

Up to now, we've learned about modular JAR files and about their structure. But how can we create one? We'll find the answer in the next section when we talk about packaging and present the jar tool used to create modular JAR files.

Packaging

Java 9 allows a module to be packaged, besides in the already known normal JAR file, in a modular JAR file, multi-release JAR file, JMOD file, or JIMAGE file. The JMOD files and multi-release JAR files are discussed in Chapter 10. It's important to mention that the Java Platform Module System doesn't force in any way a module to be packaged as a modular JAR.

Let's start by discussing how we can package a module in a modular JAR. You can do this using the jar tool that can be found under the JDK_HOME\bin directory.

Package as a Modular JAR Using the jar Tool

The jar tool has been enhanced to support the newly introduced concept of modules. Table 4-2 displays the new options that have been added in JDK 9 to the jar tool, according to the official JDK 9 API specification.

Table 4-2. *New Options Added to the JAR tool in JDK 9*

Short Format	Long Format	Description
`-d`	`--describe-module`	Prints the module descriptor
	`--module-version=VERSION`	Specifies the module version when creating a modular JAR or when updating a non-modular JAR
	`--hash-modules=PATTERN`	Computes and records the hashes of modules matched by the given pattern and that depend directly or indirectly upon a modular JAR being created or upon a non-modular JAR being updated
`-p`	`--module-path`	Specifies the location of module dependence for generating the hash
	`--release VERSION`	Puts the files in a versioned directory of the JAR file

The options presented in Table 4-2 can be used when we create modular JARs or when we update a non-modular JAR. The `jar --help` command has also been enhanced in JDK 9. It contains more detailed descriptions about each command.

Using the jar tool, we create a new modular JAR file called moduleA.jar in the lib directory by packaging everything that exists in the modules directory:

```
$ jar --create --file lib/moduleA.jar --main-class com.apress.moduleA.Main -c modules
```

- The `--create` option creates the modular JAR file.
- The `--file` option indicates the name of the modular JAR that will be created. It also specifies the location where it will be created (in our case, the lib directory).
- The `--main-class` option sets the `Main` class while the module is being packaged.
- The `-c` option specifies the location where the compiled modules are (in our case, they're in the modules directory, which contains the compiled class files for the module com.apress.moduleA).

■ **Note** In order to be able to package a module as a modular JAR, the module must have been compiled before.

Adding a Module Version

When creating a modular JAR, a module version can also be added. The option `--module-version` can be used during packaging to add some metadata regarding the version of the module. The metadata isn't added to the module declaration and won't be processed at runtime. It's important to make this distinction.

Printing the Module Descriptor

The module descriptor is printed with the jar tool's command-line option `--describe-module`:

```
$ jar --describe-module
```

Or simply:

```
$ jar -d
```

By printing the module descriptor we can see what the corresponding JAR file contains. The following information is displayed:

- The name of the module that's contained in the JAR file
- A list of modules that the module contained into the JAR file depends on
- The name and package of the main class

An important change is the introduction of a long format for the specification of the options. In Java 8, we had only the short format composed of a letter. In Java 9 it's possible to use both the short and long formats, as described in Table 4-3. However, there are situations when there's no corresponding for the long format.

■ **Note** The Java Archive Tool, also called the jar tool, is an archiving tool that creates a JAR file by archiving a couple of files. Java 9 improved the jar tool by adding support for modules.

The next section describes a fundamental feature in JDK 9: the introduction of the module path.

The Module Path

Project Jigsaw introduces a new concept for replacing the class path: the module path, which can represent one of the following:

- A path to a sequence of directories that contain modules
- A path to a modular JAR file
- A path to a JMOD file

In contrast to the module path, the class path represents a sequence of JAR files. The module path is used by the compiler to find the modules in order to resolve them. The module path can be mixed together with the class path. In this case, the classes that are part of the modules are able to depend on anything that exists on the class path.

Every existing artifact from the module path must have a module declaration. On the module path, we can't have artifacts that don't have a module declaration.

■ **Note** Even if the module path was introduced in Java 9 to replace the class path, the class path still exists and can be used standalone or in combination with the module path.

There are three types of module paths introduced in Jigsaw:

- Application module path
- Compilation module path
- Upgrade module path

We already worked with the application module path and the compilation module path when we compiled and ran multiple modules. The next subsections go over these types of module paths.

Application Module Path

The application module path is used by the Java launcher to mark the directory that incorporates the application modules. It's expressed using the new command-line option `--module-path` or its short-form, `-p`.

Figure 4-9 shows the command-line syntax of the module path flag:

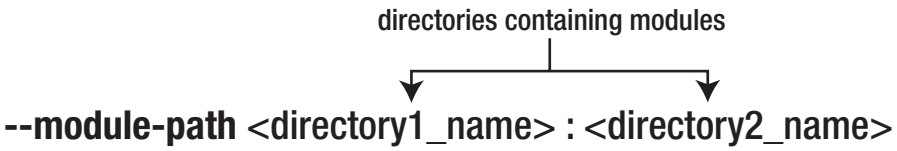

Figure 4-9. *Syntax of the --module-path command-line option*

The module path flag gets a list of directories as parameters separated by a colon. There can be an unlimited number of directories listed, but for each of them there has to be a colon separating them.

■ **Note** In the previous example, we used a colon (:) to separate directories, but a colon is used only for Linux environments. For Windows environments, we must use a semicolon (;) instead.

The modules contained in the directories that form the module path can be:

- Packaged as modular JAR files.
- Exploded as standalone class files.

The Java launcher can load exactly the module that it needs from the module path because it knows this information due to the configurations that exist in the module declaration.

■ **Note** The module path allows specifying modules instead of JAR libraries as the class path.

At runtime, using the module path is possible to specify the different types of modules that we've built together with the project. A module can be only in one place. If it's in more than one place, the first occurrence will be retained, and the other occurrences won't be taken into consideration.

■ **Remember** The module path can contain only modules.

Besides the application module path, there are two more types of module path: the compilation module path and the upgrade module path.

Compilation Module Path

The compilation module path, which contains definitions of modules in source form and is used together with `javac`, is specified using the new Java option `--module-source-path` on the command-line. It's used during the compilation to inform the Java compiler of the location of the modules that must be searched.

Figure 4-10 illustrates the syntax of the command-line `--module-source-path` option:

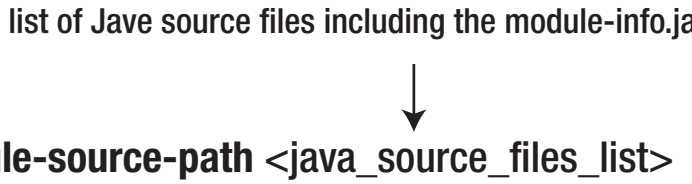

Figure 4-10. *Syntax of the --module-source-path command-line option*

The `--module-source-path` flag specifies a list of Java source files to be compiled. They can be listed one after another separated by a blank space.

■ **Note** In most situations, the list of Java files will be huge. Linux helps you in this respect. You can type `$(find. -name '*.java')` to get the entire list of directories having the extension .java.

Generally, we can think of the `--module-source-path` command-line option as the module correspondent of the `--sourcepath` option.

Upgrade Module Path

The upgrade module path is specified using the Java compiler option `--upgrade-module-path` on the command-line. According to OpenJDK, "it contains compiled definitions of modules intended to be used in place of upgradeable modules built-in to the environment." The upgrade module path isn't covered in this book.

In this section we'll talk about the module resolution process.

Module Resolution

Module resolution is a process introduced in Java 9 that checks the correctness of the module path and also resolves the dependencies that exist throughout a module system. It takes place at both compile-time and runtime. The goal of the module resolution process is to end up with a minimum necessary set of resolved modules in order to be able to run the application.

■ **Note** Modules are resolved during build and installation. They're not resolvable during runtime.

The `requires` clauses from the module declaration located in the module-info.java file provide the module system with valuable information about the dependencies that have to be solved. These dependencies are nothing more than modules that our current module depends upon. In Java 9, they're called *observable modules*, covered later in this chapter in the section "Types of Modules."

After all the observable modules of our current module are found, the module system doesn't stop. It searches further for the observable modules of the most recently found modules. This process continues until every dependence of every module is fulfilled and until the base module java.base is reached.

■ **Note** At compile-time, during the module resolution, Jigsaw searches to see whether there are any cyclic dependencies. If it finds any, the application won't compile.

Root Module

The root module is the module that the resolution process starts with. It's specified in the `java` command using the `--module` option, as we've seen in the examples when we ran the modular application.

First, the root module is added to the group of resolved modules. Second, the module system scans the module descriptor of this module and adds all the dependencies (modules) to the group of resolved modules. The process continues, and the Java Platform Module System tries to find the dependencies on the other modules. When all the modules searched are found and the module java.base is reached, the process stops. After the resolving process is finished, we have all the necessary modules for running our software application.

In some cases, a module is on the module path but wasn't found during the resolution process so that it can be added to the module graph. Here, we have to manually add the module to the module graph. This can be done using the `--add-modules` command-line option. We discuss this option in more detail in Chapter 8.

It's important to know that the module resolution process detects any modules that could eventually be missing. If a mandatory module is missing, the module resolution process stops, and an exception is thrown.

■ **Note** If we have an incomplete module path at compile-time, the compiler will give us a warning. Both platform modules and developer modules are searched for during the resolution process.

Another important topic in Jigsaw is accessibility, covered in the next section.

Accessibility

The accessibility rules changed fundamentally in Java 9. A type declared as public but not exported will be available only inside the module where it resides. This a major change compared to older versions of Java. Prior to Java 9, it was enough to mention that a type is public and therefore was accessible anywhere. In Java 9, declaring a type as public doesn't imply that it will be accessible everywhere.

Simply reading a module doesn't guarantee access to its packages. In addition, in order to be accessible, modules have to export some of their packages. Only the public types from the exported packages will be accessible from another module. By taking advantage of strong encapsulation and reliable configuration, we can explicitly define which of the module's types are available for external access. In this way, we can very easily hide our implementation internals.

Hiding the implementation details becomes the standard in Java 9. It's obtained by default—we don't have to opt for it. It's enough to omit to export a type in order to make that type strong encapsulated and invisible from outside of the module. A package can quickly benefit from the power of strong encapsulation by simply being placed inside a module. In Java 9, we're able to decide which types should be accessible from outside by enumerating them in the module declaration. Very simple and concise.

To recap, in order for a module A to read package P from module B, two conditions have to be met simultaneously. The first is that module A should read (`requires`) module B. The second condition is that module B should export its package P.

■ **Note** In Java 9, accessibility is imposed at both compile-time and at runtime. An error of type `IllegalAccessError` is thrown at runtime if the accessibility rules are breached. The accessibility checks are enforced in the Java Virtual Machine.

In Java 9, simply setting a `public` modifier to a type doesn't mean that access is granted. In the Java versions before 9, setting an accessibility type of `public` to a type conferred him global accessibility, but in Java 9 three conditions have to be simultaneously met in order to make a type called T accessible outside the module:

- The package in which type T resides has to be exported.
- The module that needs to access type T has to read the module that contains the type T.
- Type T must have a public identifier.

These conditions are illustrated in Table 4-3, which displays a comprehensive list of the cases when the accessibility is granted or not:

- The "Module Is Read" column has the value Yes if the module that contains the type T is read by the module that wants to access the type T.
- The "Package Is Exported" column has the value Yes if the module that contains the type T exports the package of type T.
- The "Access Modifier of Type T" column represents the access modifiers for type T.
- The "Accessible in the Other Module" column specifies Yes if the type T is accessible from the other module, and No otherwise.

Table 4-3 shows the new accessibility cases in Java 9 together with the results.

Table 4-3. *Accessibility Cases in Java 9*

Module Is Read	Package Is Exported	Access Modifier of Type T	Accessible in the Other Module
Yes	Yes	Public	**Yes**
Yes	Yes	Protected	**No**
Yes	Yes	(default)	**No**
Yes	Yes	Private	**No**
Yes	No	Public	**No**
Yes	No	Protected	**No**
Yes	No	(default)	**No**
Yes	No	Private	**No**
No	Yes	Public	**No**
No	Yes	Protected	**No**
No	Yes	(default)	**No**
No	Yes	Private	**No**
No	No	Public	**No**
No	No	Protected	**No**
No	No	(default)	**No**
No	No	Private	**No**

As a rule, a public component of an exported package can be accessed from outside of the module provided that the module that makes use of it reads the origin module. In contrast, a public element of a non-exported package isn't accessible from outside of the module. It's accessible by default to all the source code from the module where it resides, but it won't be accessible from outside of the module.

The Java compiler throws exceptions when we attempt to access types that aren't accessible. The most common is a `ClassNotFoundException`. At runtime the most common errors are `IllegalAccessError` or `InaccessibleObjectException`.

Readability vs. Implied Readability

Readability is the relation between two modules that refers to the fact that a module that reads another module can access the types from its exported packages. In this case, we say that a module *reads* another module.

We touched the subject of readability in previous examples when we described the `requires` and `exports` directives inside of a module declaration. To recap the concept of readability, you can go back to the "Module Declaration" section, subsection "The requires Clause." What we haven't covered yet is the new concept of implied readability.

Implied Readability

Implied readability refers to the situation when

- The first module reads the second module.
- The second module reads the third module.
- The first module logically reads the third module as a result of the two conditions just met.

Suppose we have a module B that uses a type from a module C (B reads C). If another module A reads module B and uses types from module C, then without implied readability, module A should explicit specify that it also requires module C. By using implied readability, it's not necessary to specify this in module A. It's enough to specify in the module-info.java of module B that it `requires transitive` the types from module C. As a result, every module that reads types from module B will automatically be able to access the types from module C.

Figure 4-11 illustrates the corresponding module graph of the three modules and shows the readability relations between them.

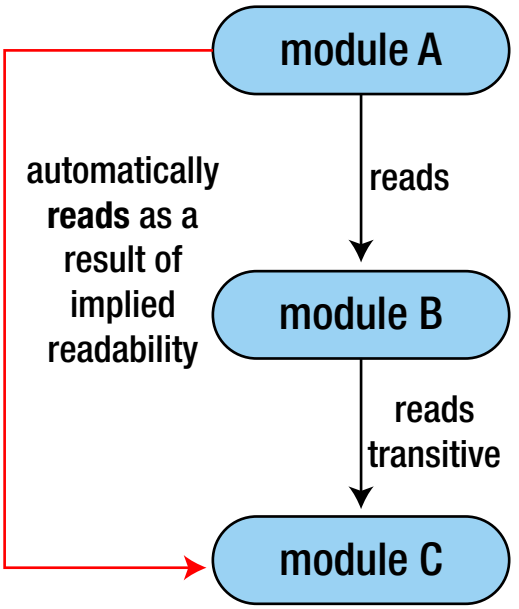

Figure 4-11. *Module graph showing implied readability*

Listing 4-18 illustrates the module descriptor of module A, which requires module B.

Listing 4-18. Module Descriptor of Module A

```
// module-info.java
module A {
        requires B;
}
```

Listing 4-19 shows the module descriptor of module B, which requires transitive module C.

Listing 4-19. Module Descriptor of Module B

```
// module-info.java
module B {
        requires transitive C;
}
```

It's not necessary for module A to `require` module C, because module B `requires transitive` module C. As a result, module A can automatically access types from module C.

■ **Note** Implied readability is achieved by adding the statement `requires transitive` in the module declaration, followed by the name of the module that the current module depends upon.

Figure 4-12 illustrates the syntax of the `requires transitive` clause. It takes a module name as parameter.

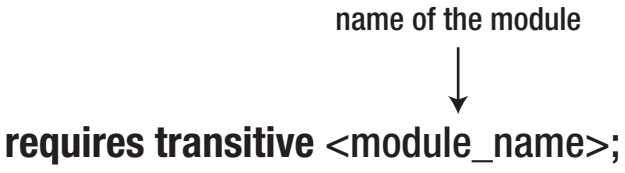

Figure 4-12. *Syntax of the requires transitive clause*

Let's look at an example of implied readability with platform modules in order to understand this better. Listing 4-20 shows the module declaration of the platform module java.desktop that defines `requires transitive` clauses in order to take advantage of implied readability.

Listing 4-20. The Module Descriptor of module java.desktop

```
// module-info.java
module java.desktop {
        requires transitive java.datatransfer;
        requires transitive java.xml;
        requires java.prefs;

        exports java.applet;
        ...
}
```

Listing 4-21 is an excerpt of the class `DocumentHandler` located in module java.desktop in the package com.sun.beans.decoder:

Listing 4-21. Class DocumentHandler from Module java.desktop

```
// jdk/src/java.desktop/share/classes/DocumentHandler.java

package com.sun.beans.decoder;
```

```
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParserFactory;
...

public final class DocumentHandler extends DefaultHandler {

...

  public void parse(final InputSource input) {
        if ((this.acc == null) && (null != System.getSecurityManager())) {
            throw new SecurityException("AccessControlContext is not set");
        }
        AccessControlContext stack = AccessController.getContext();
        SharedSecrets.getJavaSecurityAccess().doIntersectionPrivilege(new
        PrivilegedAction<Void>() {
            public Void run() {
                try {
                    SAXParserFactory.newInstance().newSAXParser().parse(input,
                    DocumentHandler.this);
                }
                catch (ParserConfigurationException exception) {
                    handleException(exception);
                }
                catch (SAXException wrapper) {
                    Exception exception = wrapper.getException();
                    if (exception == null) {
                        exception = wrapper;
                    }
                    handleException(exception);
                }
                catch (IOException exception) {
                    handleException(exception);
                }
                return null;
            }
        }, stack, this.acc);
    }
}
```

As you can see, the class `DocumentHandler` from module java.desktop uses `SaxParserFactory` from the java.xml module. This denotes that module java.desktop uses a type from module java.xml, so a readability relation occurs between module java.desktop and module java.xml.

If we add a dependency to the java.desktop module in our module descriptor and try to use the `parse()` method from the `DocumentHandler`, we're attempting to access types not only from module java.desktop but also from module java.xml. In order to access the method from our own module, it's mandatory that the module java.xml is *required transitive* by the module java.desktop. In this way our module can take advantage of implied readability and have access to the types from the module java.xml without explicitly requiring them. Any module that requires module java.desktop automatically requires the modules java.datatransfer and java.xml, because both are present in the module descriptor of module java.desktop. By requiring the module java.desktop, we get access to the exported packages from modules java.desktop, java.datatransfer, and java.xml.

If we omit the `transitive` keyword and use only the `requires` directive, we have the situation where our module is able to access types only from the module java.desktop, but not from java.xml.

Suppose we create a simple module called myModule that requires module java.desktop. Listing 4-22 shows the module descriptor of module myModule.

Listing 4-22. Module Descriptor of Module myModule

```
// module-info.java
module myModule {
        requires java.desktop;
}
```

Figure 4-13 shows the module graph of the module myModule and expresses the readability together with the implied readability relations between modules.

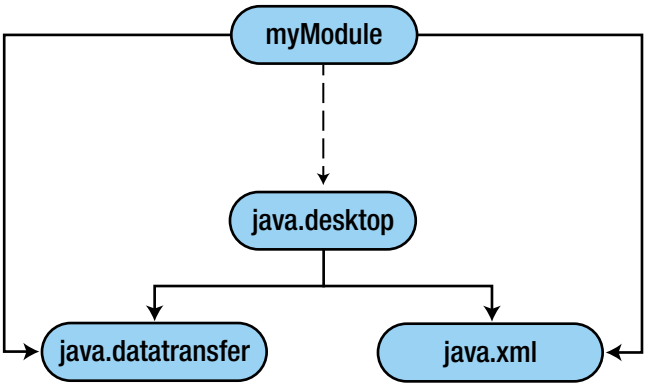

Figure 4-13. *The module graph of myModule*

We have the following situation:

- myModule requires java.desktop (readability illustrated with a dashed line in the graph).
- java.desktop module requires transitive module java.datatransfer and module java.xml (implied readability illustrated with a solid line).

Our module myModule gets readability to modules java.datatransfer and java.xml without needing to explicitly require them. As a result, it can use types from these two modules without having to worry about the need to specifically declare the dependencies to them.

Now that we've learned what implied readability is, let's see what a qualified export means.

Qualified Exports

A module can export all its packages or a group of its packages to all the modules. The `exports` clause has been enhanced to specify that a module can export a group of its packages only to a set of named modules.

Listing 4-23 is an excerpt of the module descriptor from the module java.rmi.

Listing 4-23. Excerpt from Module Descriptor of module java.rmi

```
// module-info.java
module java.rmi {
    ...
    exports com.sun.rmi.rmid to java.base;
    exports sun.rmi.registry to
        java.management;
    exports sun.rmi.server to
        java.management,
        jdk.jconsole;
    exports sun.rmi.transport to
        java.management,
        jdk.jconsole;
}
```

Package com.sun.rmi.rmid is exported using a qualified export to module java.base. Therefore it's accessible only inside the module java.base. The other modules can't access it. Only the modules specified after the `to` clause will be able to access the package.

■ **Note** A module can't access an exported package from another module if it doesn't read that module. This is true even if the package is exported using a qualified export.

The syntax used for defining qualified exports inside the module-info.java file is shown in Figure 4-14.

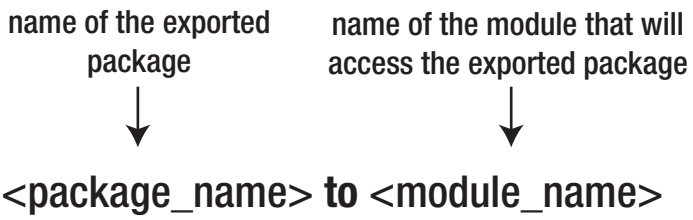

Figure 4-14. *Qualified exports syntax*

■ **Note** A qualified export directive can define multiple modules separated by comma. In contrast, a simple export directive can define only one module.

Duplicating the names of the modules in a qualified export declaration is prohibited, as shown in Listing 4-24.

Listing 4-24. Qualified Export with a Duplicate Module's Name

```
// module-info.java (com.apress.moduleA)
module com.apress.moduleA {

}
```

```
// module-info.java (com.apress.moduleB)
module com.apress.moduleB {
        exports com.apress.moduleB to com.apress.moduleA, com.apress.moduleA;
}
```

A compilation failure will occur in this case:

```
Error: duplicate export: com.apress.moduleA exports com.apress.moduleB to com.apress.
moduleA, com.apress.moduleA
```

We've learned what the qualified exports directives are and how they're different from standard exports directives. It's a great advantage to be able to specify only the specific modules that are allowed to access the module's data. Modules shouldn't be obligated to expose their packages to all the existing modules.

Qualified exports are a great benefit of strong encapsulation. They've been extensively used during the process of modularizing the JDK and are present in a couple of module-info.java files inside the JDK.

We talked about accessibility and discussed the concepts of readability, implied readability, and qualified exports. Next let's look at the different types of modules introduced in Jigsaw.

Types of Modules

Jigsaw defines two primary types of modules: *named* modules and the *unnamed* module. The named modules are divided into *normal* modules and *automatic* modules. The normal modules are also separated in *basic* modules and open modules. Figure 4-15 illustrates the classification of modules.

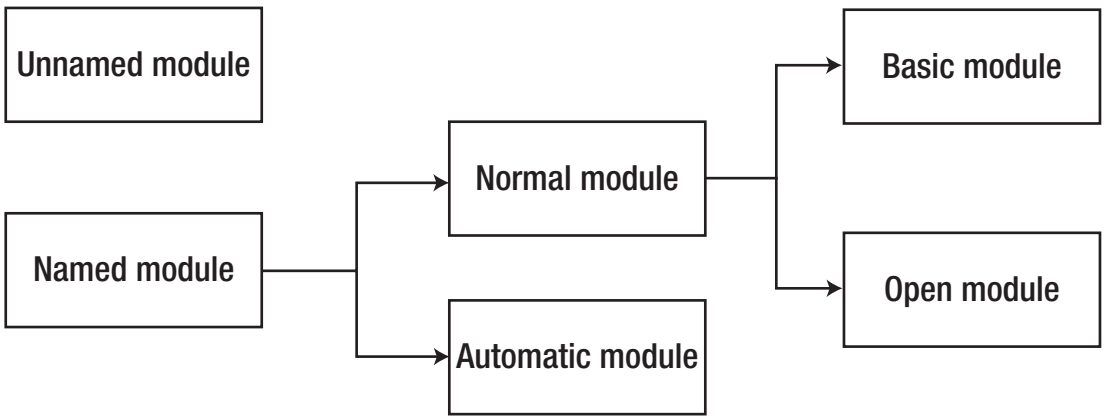

Figure 4-15. *The classification of modules in the JPMS*

First, we'll talk about the named modules together with its children, and then the unnamed module.

Named Modules

The named modules comprise all the modules from the module system except the unnamed module. There are two important things that distinguish an unnamed module from a named one. First, the unnamed module lives on the class path, whereas the named modules live on the module path. Second, the unnamed module doesn't have a name, whereas each named module has a name. A named module can be a normal module or an automatic module. Named modules are modules declared using a name in the module-info.java module descriptor file. Every module that has a declaration of form `module <module_name>` in its module-info.java is a named module. This is the single condition that has to be met for a module in order to be classified as a named module. Examples of named modules include all the platform modules, but our own modules can also be included in this category if they respect the unique condition just mentioned. Transforming a JAR file into a named module is possible by merely adding a module-info.class file to it.

Normal Modules

The notion of "normal" modules doesn't officially exist. We use this term to define a named module that isn't automatic. The main difference between a normal module and an automatic one is that a normal module has a module descriptor module-info.java, whereas an automatic module doesn't. Additionally, a normal module is explicitly declared by developers, which declares the module's dependencies in the module's module descriptor. The module descriptor of an automatic module isn't provided by developers. A normal module is declared using the keyword `module` followed by the name of the module. All the modules we've presented until now in this chapter were normal modules. A normal module doesn't export any of its packages by default. Besides that, its `exports` clauses must be explicitly specified. The `exports` clauses export packages at compile-time as well as at runtime. A normal module comprises both basic modules and open modules.

Automatic Modules

An *automatic* module is a module created after placing a JAR file onto the module path. Comparing an automatic module to a normal one, two important distinctions emerge:

- An automatic module requires by default all the existing modules from the system, which comprise all our own modules, plus all modules from the JDK image, plus all the other automatic modules.
- An automatic module exports all its packages by default.

An automatic module can access types on the class path and is useful for third-party code especially. Automatic modules are used for migrating existing applications to Java 9. Chapter 8 talks about them in more detail.

■ **Note** An automatic module isn't explicit declared by us inside a module descriptor. It's automatically created when placing a JAR file into the module path.

Basic Modules

We call every named module that isn't an open module a *basic* module. However, the term "basic" module doesn't officially exist in JDK 9. We use it to define a named module that's neither automatic nor open. A basic module has the same set of characteristics as a normal module, except that it's not opened for deep reflection.

www.allitebooks.com

Open Modules

Inside a module, packages aren't accessible to code from another module at compile-time, even when using deep reflection. However, many third-party libraries and frameworks use reflection to access the internals of JDK at runtime. As a result, all these frameworks aren't working in JDK 9 unless reflective access is granted. In JDK 9, reflective access is granted only by code in named modules to code from the class path. It's not granted by default by code in named modules to code in other named modules. As a result, if the third-party libraries or frameworks lie on the class path, they have reflective access by default in the JDK. If they live on the module path, then they don't have reflective access in the JDK. But to grant reflective access to all the packages in a module, the module should be declared as open.

An *open* module is defined by placing the identifier `open` in front of the keyword `module`, followed by the name of the module.

Open modules make all the packages inside of the module available for deep reflection. When we say "all the packages", we mean both the public and private packages. We can also opt between opening an entire module for deep reflection or opening only specific packages. When choosing the latter, we don't specify an entire module as open but only one or more packages inside the module. The keyword `open` can be placed near the module name or inside the module descriptor to open specific packages.

■ **Note** The reason behind open modules is that they permit frameworks to reflect over the module's internals, which isn't possible using basic modules. Frameworks like Spring, JPA, and Hibernate need reflective access at runtime.

Listing 4-25 defines an open module called com.apress.myModule that requires two Spring modules, spring.tx and spring.context.

Listing 4-25. Defining an Open Module

```
open module com.apress.myModule {
        requires spring.tx;
        requires spring.context;
        exports com.apress.myModule.myPackage;
}
```

Two important facts must be stressed in regard to the previous example:

- All types from all packages of the module com.apress.myModule are available for deep reflection at runtime.
- At compile-time, only the public and protected types in package com.apress.myModule.myPackage are accessible.

As a result, the Spring framework can make use of the `setAccessible()` method to access the non-public elements of the com.apress.myModule.myPackage package.

Enabling Core Reflection Using Open Modules

With respect to the principles of strong encapsulation, there are some constraints introduced in Java 9 when calling the method `setAccessible()` method of the `java.lang.reflect.AccessibleObject` class. We can't use the method `setAccessible()` to make private fields or methods from other modules accessible in our module. But there is a solution to make them accessible: by declaring the target module as an open module.

In the following example we have two modules. Class `Employee` from module target contains a private `String` field called `employeeName`. We want to set this field to be accessible from our second module called testReflection.

Listing 4-26 shows the declaration of module target, which exports its package.

Listing 4-26. The module-info.java of Module target

```
module target {
       exports target;
}
```

In Listing 4-27 the module testReflection reads the module target.

Listing 4-27. The module-info.java of Module testReflection

```
module testReflection {
       requires target;
}
```

Listing 4-28 shows the class `Employee` from module target that contains a private type called `employeeName`:

Listing 4-28. Definition of Class Employee

```
package target;

public class Employee {

    private String employeeName = null;

    public Employee(String employeeName) {
        this.employeeName = employeeName;
    }
}
```

The `Main` class creates an object of type `Employee` and calls its constructor, setting the value `"John"` to `employeeName`. After that, a `Field` object is returned that represents the field `employeeName` and the method `setAccessible()` with parameter `true` is called on this field in order to make it accessible throughout our testReflection module.

Listing 4-29 shows the `Main` class of the application.

Listing 4-29. Main Class from Package testReflection

```
package testReflection;

import java.lang.reflect.*;
import target.*;

public class Main {

    public static void main(String[] args) {
```

```
        Employee employee = new Employee("John");
        try {
            Field employeeField = Employee.class.getDeclaredField("employeeName");
            employeeField.setAccessible(true);
        }
        catch(NoSuchFieldException noSuchFieldException) {
        }
    }
}
```

We compile the code inside the two modules and specify the location of the compiled files to be in the out directory. Listing 4-30 shows the usage of the `--module-source-path` option to specify that all the files that have the extension .java are on the module source path.

Listing 4-30. Compile Files Using the flag --module-source-path

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

Listing 4-31 illustrates the `java` command used to run the `Main` class of our module. We pass the `Main` class to the `--module option` and the `--module-path` option points to the out directory where all the compiled class files exist.

Listing 4-31. Run the Main Class Using the `--module option`

```
$ java --module-path out --module testReflection/testReflection.Main
```

Unfortunately, an exception is thrown when attempting to run our application because the call of the method `setAccessible()` fails. Thus we can't make the private field `employeeName` accessible.

```
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to make
field private java.lang.String target.Employee.employeeName accessible: module target does
not "opens target" to module testReflection
        at java.base/jdk.internal.reflect.Reflection.throwInaccessibleObjectException
        (Reflection.java:424)
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible
        (AccessibleObject.java:198)
        at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:171)
        at java.base/java.lang.reflect.Field.setAccessible(Field.java:165)
        at testReflection/testReflection.Main.main(Main.java:14)
```

■ **Note** In this example, we have readability between the two modules target and testReflection. But due to the very powerful strong encapsulation mechanism introduced in Java9, calling the method `setAccessible()` on a private field of the other module throws an `InaccessibleObjectException`.

We can fix this issue very easily by defining the target module as an open module instead of a strong module. Calling the `setAccessible()` method on a private field of the target module will succeed. The private field `employeeName` is now accessible into the testReflection module.

Listing 4-32 defines the target module as an open module by specifying the keyword `open`.

Listing 4-32. Define the Target Module as an Open Module

```
open module target {
        exports target;
}
```

■ **Note** You can find the source code for the first example in the folder /ch04/CoreReflectionFail and the source code for the second example in the folder /ch04/CoreReflectionSucceed.

Until now we've talked about the named module and its types. Next we'll talk about the unnamed module.

The Unnamed Module

An unnamed module, as the term suggests, doesn't have a name and isn't declared. It comprises all the JAR files or modular JAR files from the class path. All these JAR files together form the unnamed module. The Java Platform Module System first searches for a specific type on the module path. The module path is searched before the class path. If the type isn't found on the module path, the search is performed on the class path. If the type is found on the class path, it will be part of the so-called unnamed module. We use the singular term *unnamed module* instead of the plural *unnamed modules* because the unnamed module is unique for each class loader. There's only one, single unnamed module for each class loader.

■ **Note** An unnamed module is bound to a class loader. There's a one-to-one relationship between a class loader and an unnamed module. The unnamed module reads all the named modules in the JDK image and on the module path. It also exports all of its packages.

By default, the unnamed module reads all the named modules from the system. In this way, according to the Java 9 accessibility rules, the unnamed module can access all packages from all the named modules that are being exported. The reverse isn't true, meaning that the named modules can't read the unnamed module. If we try to access code on the class path (in the unnamed module) from the module path, the compilation will fail. To succeed, we need to turn the code from the unnamed module into automatic modules. Therefore, we take the JAR file out of the class path and place it on the module path to become an automatic module.

■ **Note** A named module can't require an unnamed module.

All classes that aren't contained into the named modules are implicitly contained in the unnamed module. All the packages contained in the unnamed module are open by default to all the modules from the module path, which makes reflective access from the module path on the class path possible.

Observable Modules

The *observable* modules are not a separate category of modules. This is why we didn't include them in the classification of modules. The term *observable modules* is used to denominate all the modules from the system: platform modules, library modules, and our own modules. The modules from the module path are part of the observable modules too.

Summary

This chapter presented the new concept of module in Jigsaw. We learned how to define a module and described the structure of the new module-info.java file that represents the module descriptor.

Inside the module descriptor there are five types of directives that can be used: `requires`, `exports`, `opens`, `uses`, and `provides`. The first three were explained in detail throughout this chapter. We saw how we can define dependencies between modules using the `requires` directive and also how we can specify which packages a module exports using the `exports` directive. Further, we defined a couple of modules and illustrated the dependencies between them in a module graph.

We mentioned the differences between the `exports` and `opens` clauses. The `exports` clause allows compile-time and runtime access to the public types of a specific package. The `use` clause allows runtime access using reflection for both the public and private types of a specific package.

We compiled and ran a single module as well as multiple modules. To be able to do this, we used the new concept of module path together with the new command-line options `--module-source-path` and `--module-path`. Then we talked about the new modular JARs and described their internal structure. I described the enhancements added to the jar tool and showed how to package a modular JAR using the jar tool.

The chapter explained the three types of existing module paths: the application module path, the compilation module path, and the upgrade module path. It also talked about the module resolution process and about the new accessibility rules that were introduced in Java 9. We described topics like readability, implied readability, and qualified exports.

This chapter concluded by describing the different types of modules in Jigsaw: normal modules, open modules, named modules, unnamed modules, automatic modules, and observable modules. We stressed that as a consequence of the fact that a named module can't read the unnamed module, code from the module path can't access code inside the JARs placed on the class path.

In Chapter 5 you'll learn about modular runtime images.

CHAPTER 5

Modular Runtime Images

In this chapter we'll look at the structure of the new modular runtime image introduced in Java 9, which brings an important benefit in terms of improved performance and maintainability. On the other hand, the new format of the runtime image doesn't necessarily result in preserving the exactly same functionality of all the existing APIs.

Note This chapter is an informational one that describes the format of the modular runtime images introduced in JDK 9.

Modular Runtime Images

Chapter 3 showed how the source code in the JDK was restructured around modules. In this chapter we'll talk about the new modular runtime image that was implemented in the Java Enhancement Proposal 220. This JEP modified the structure of the JDK and JRE as a consequence of the introduction of modules. It also defines the layout of the modular runtime image.

The introduction of modules in Java 9 caused an important change in the structure of the JDK and the JRE. As a result, a new runtime format was introduced. The minimum possible runtime that we can have in Java 9 would consist only of the module java.base. A JDK 9 image can be accessed not only by tools that are running on Java 9, but also by tools running on Java 8, for example.

Note For accessing classes and resources in the JDK and JRE, a helper interface has been introduced in Java 9.

Another important change, not directly related to modules but more related to the JDK, is the replacement of the rt.jar file and the tools.jar file with the new runtime image.

Note Java 9 doesn't remove the JAR files and doesn't prohibit them. JAR files continue to work in Java 9.

Because JAR files can cause many problems, the intention of the JCP team was to stop using them inside the JDK and the JRE as much as possible.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_5

The Runtime Image Prior to Java 9

This section covers the structure of the runtime image prior to Java version 9. We already talked about it in Chapter 2, but now we'll get into more details. Before Java 9, the JDK build provided us with two types of runtime images: a Java Runtime Environment (JRE) image and a Java Development Kit (JDK) image.

The JRE Image Prior to Java 9

A Java Runtime Environment was a complete implementation of the Java SE Platform. A JRE image was composed of two directories: bin and lib.

The bin directory contained the `java` command for launching the runtime system and also executable binaries, like javacp, java-rmi, javaw, javaws, keytool, pack200, rmid, rmiregistry, and servertool.

The lib directory was larger than the bin directory and contained .properties and .policy files. The ext directory was placed inside the lib directory and contained JAR files like nashorn.jar, sunec.jar, zipfs.jar, and others. The most important thing to mention is that inside the lib directory we could find the rt.jar file. The lib directory comprised the runtime system's dynamically-linked native libraries on the Mac OS and Linux operating systems.

The JDK Image Prior to Java 9

On the other hand, prior to Java 9, a JDK image enclosed a JRE. It had a copy of the JRE in its jre subdirectory. A JDK image contained many directories, but the most important three of them were the lib, the bin, and the include directories.

■ **Note** A JDK image contained libraries and development tools.

The lib directory consisted of JAR files comprising the implementations of the JDK's tools. The tools.jar file, which included the classes that composed the javac compiler, was located into this lib directory. The bin directory had command-line debugging and development tools like javac, javadoc, and jconsole. The include directory contained C and C++ header files for use in compiling native code that interfaces directly with the runtime system.

Why a New Format for the Runtime Images?

According to Open JDK, there are various reasons why a new format is required for the runtime images. First of all, the new runtime format is more powerful than the old JAR format. Second, the new runtime format can be easily enhanced to hold precompiled native code for Java classes or precomputed JVM data structures. Third, the new runtime format can store class and resource files from application modules, JDK modules, and library modules.

The most important reason behind the decision to revamp the JDK and the JRE is stated by OpenJDK on their website: "to draw a clear distinction between files that developers, deployers, and end-users can rely upon and, when appropriate, modify, in contrast to files that are internal to the implementation and subject to change without notice."

OpenJDK lists three more reasons:

- "to provide supported ways to perform common operations that today can only be done by inspecting the internal structure of a runtime image such as, e.g., enumerating all of the classes present in an image"
- "to enable the selective de-privileging of JDK classes that today are granted all security permissions but do not actually require those permissions"
- "to preserve the existing behavior of well-behaved applications, i.e., applications that do not depend upon internal aspects of JRE and JDK runtime images"

The Runtime Image in Java 9

This section describes the structure of the new runtime image introduced in Java 9.

Identical Structure of the JDK and JRE

The JRE and the JDK have the same structure in Java 9. This is different from older versions of Java where, as described earlier, there was a clear distinction between the JDK and the JRE. A JDK image is simply a runtime image that contains the development tools from the JDK.

Configuration files that were once located in the lib directory are now located inside the conf directory. These are files we can edit. The files in the lib directory are implementation details of the runtime system.

■ **Note** The files from the lib directory should not be modified.

The Structure of the New Runtime Image

Figure 5-1 shows the new structure of the runtime image in Java 9.

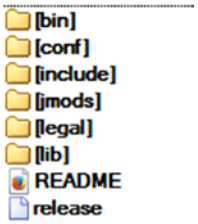

Figure 5-1. *The structure of the runtime image in Java 9*

Let's see what kind of directories the new modular runtime image contains:

- The bin directory incorporates command-line launchers represented by the modules that are linked into the image. Some of the most important command-line launchers are java, javac, javadoc, javah, javap, jcmd, jconsole, jdeps, jimage, jlink, jmod, jshell, and jstat. We are not going to describe all of them here. You can check the bin directory of the JDK 9 build.
- The lib directory consists of the runtime system's dynamically-linked native libraries.
- The conf directory contains files that can be edited. Among them are .properties files and .policy files.
- The jmods directory contains all the JMOD files.
- The legal directory contains the copyright files.

The root directory of a modular runtime image contains copyright, readme, and release files.

The release File

The structure of the release file contains information about the modules, OS version, source, operating system arch, operating system name, Java version, and the full Java version:

```
IMPLEMENTOR="Oracle Corporation"

JAVA_VERSION="9"

MODULES="java.base java.datatransfer java.logging java.activation java.compiler java.
rmi java.transaction java.xml java.prefs java.desktop java.security.sasl java.naming jdk.
unsupported java.corba java.instrument java.jnlp java.management java.management.rmi java.
scripting java.xml.ws.annotation java.xml.crypto java.security.jgss java.sql java.sql.rowset
java.se jdk.httpserver java.xml.bind java.xml.ws java.se.ee java.smartcardio javafx.base
jdk.jsobject javafx.graphics javafx.controls jdk.deploy jdk.javaws jdk.plugin javafx.deploy
javafx.fxml javafx.media javafx.swing jdk.xml.dom javafx.web jdk.accessibility jdk.internal.
jvmstat jdk.attach jdk.charsets jdk.compiler jdk.crypto.ec jdk.crypto.cryptoki jdk.crypto.
mscapi jdk.deploy.controlpanel jdk.dynalink jdk.internal.ed jdk.editpad jdk.hotspot.agent
jdk.incubator.httpclient jdk.internal.le jdk.internal.opt jdk.internal.vm.ci jdk.jartool jdk.
javadoc jdk.jcmd jdk.management.agent jdk.management jdk.jconsole jdk.jdeps jdk.jdwp.agent
jdk.jdi jdk.jfr jdk.jlink jdk.jshell jdk.jstatd jdk.localedata jdk.naming.dns jdk.naming.rmi
jdk.net jdk.pack jdk.packager.services jdk.packager jdk.plugin.dom jdk.plugin.server jdk.
security.jgss jdk.policytool jdk.rmic jdk.scripting.nashorn jdk.scripting.nashorn.shell jdk.
sctp jdk.security.auth jdk.snmp jdk.xml.bind jdk.xml.ws jdk.zipfs oracle.desktop oracle.net"

OS_ARCH="x86_64"

OS_NAME="Windows"

SOURCE=".:a4371edb589c+ closed:3b9bef864bcf corba:c72e9d3823f0 deploy:d12d0210bc37
hotspot:1ca8f038fceb hotspot/make/closed:0b47834a0294 hotspot/src/closed:f5870a8748c9
hotspot/test/closed:2a88d69ed789 install:6549b99d10f0 jaxp:332ad9f92632 jaxws:b44a721aee3d
jdk:80acf577b7d0 jdk/make/closed:1793d3af1ed9 jdk/src/closed:8e4a66cb15a6 jdk/test/
closed:860a0f54d259 langtools:2f01728210c1 nashorn:aa7404e062b9 sponsors:d751e23bea1e"
```

The next section discusses the rt.jar, tools.jar, and dt.jar files that were removed in JDK 9.

Removed Files

Rt.jar Removed

The rt.jar file comprised the entire compiled class files that formed the JRE. These represented all the compiled classes from the Core Java API, including the sun and com packages. rt.jar used to be located inside the lib folder of the JRE. In Java 8, the rt.jar file weighed approximatively 52 MB. In all Java versions prior to version 9, it was absolutely mandatory to include the rt.jar file into the class path in order to be able to access the core Java libaries. But the rt.jar file has been completely removed in Java 9. It doesn't exist anymore among the files that make up the JDK 9.

Tools, compilers, and integrated development environments (IDEs) using rt.jar are affected by the removal of the rt.jar file. They have to be adjusted by their authors in order to continue to work properly in Java 9.

According to the specification of JEP 220, "the class and resources files previously stored in lib/rt.jar, lib/tools.jar and lib/dt.jar, and various other internal JAR files will now be stored in a more efficient format in implementation specific files in the lib directory."

In JDK 9, the rt.jar file was replaced by the new runtime.

Tools.jar and dt.jar Removed

Before JDK 9, the tools.jar file was used by the JDK and was located in the lib directory of the JDK version 8 or lower. It contained classes used by javac and also support for tools like javah, javap, jdeps, javadoc, and more. The dt.jar file was also located inside the lib directory and contained `Swing` classes.

■ **Note** Both tools.jar and dt.jar have been removed in Java 9.

In Java 9, resources and class files that were formerly located inside the tools.jar file are visible via the bootstrap or application class loaders in a JDK image. Similarly, resource and class files that were formerly located inside the dt.jar file are visible via the bootstrap class loader in Java 9.

New URI Scheme

A new URI scheme was introduced in JDK 9. To demonstrate the new URI scheme, we'll first show a simple example. In this example, we use the `getSystemResource()` method of the `ClassLoader` class to find a URL resource of the class `String` from the search path used to load classes. The resource will be located through the system class loader.

In Listing 5-1 the `getSystemResource()` method retrieves the URL resource of the class `java.lang.String`.

Listing 5-1. The Main Class of Module com.apress.getSystemResource

```
// Main.java (module com.apress.getSystemResource)
package com.apress.getSystemResource;
import java.net.URL;
```

```
public class Main {
        public static void main(String[] args) {
           URL url = ClassLoader.getSystemResource("java/lang/String.class");
           System.out.println(url);
        }
}
```

Listing 5-2 defines the module descriptor of the module com.apress.getSystemResource.

Listing 5-2. The module-info.java File of Module com.apress.getSystemResource

```
module com.apress.getSystemResource {

}
```

We run the previous module and see the resulting URL printed in the console:

```
jrt:/java.base/java/lang/String.class
```

Let's now run the same example on Java 8. The resulting URL in this case would be as follows:

```
jar:file:/C:/Program%20Files/Java/jdk1.8.0_101/jre/lib/rt.jar!/java/lang/String.class
```

■ **Note** You can find the source code for this example in the directory /ch05/getSystemResource.

We observe that the output is completely different between Java 8 and Java 9. In Java 8, the URL has the form of a JAR file and the `String.class` is located inside the rt.jar file. It's not possible to have the same output in Java 9 because Java 9 doesn't have an rt.jar file.

To solve this problem, Java 9 introduces a new URL scheme called jrt that provides access to the content of the runtime image. The term *jrt* is derived from *java runtime*. According to Open JDK, the jrt URL scheme "is used for naming the modules, classes, and resources stored in a runtime image without revealing the internal structure or format of the image."

■ **Note** A jrt URL can have a total of four distinct forms.

Figure 5-2 shows the simplest structure of the jrt URL.

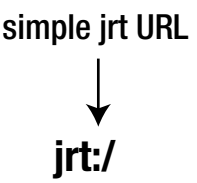

Figure 5-2. *Simplest structure of the jrt URL*

This jrt URL, which doesn't specify anything else except the jrt:/, specifies all the files that exist in the current runtime image. But we can have more representations of the jrt URL. Figure 5-3 shows another form of the jrt URL that specifies only the module name.

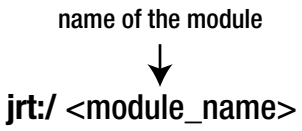

Figure 5-3. *Structure of the jrt URL with module*

In this case, the jrt URL specifies all the files from the module that was specified.

Figure 5-4 represents another form of the jrt URL that contains only the path, but no module name.

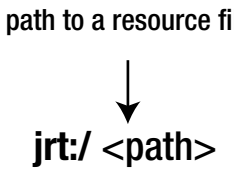

Figure 5-4. *Structure of the jrt URL with path*

In this case, the path represents a resource file or a class from the current runtime image. The path is not bound to a specific module.

Figure 5-5 shows the full structure of the jrt URL, specifying both a module and a path.

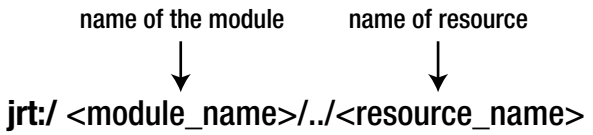

Figure 5-5. *Full structure of the jrt URL*

The full structure of the jrt URL refers to a specific resource file within the given module. The jrt scheme also allows retrieving the contents of platform modules.

Now that we've learned about the new URI scheme, let's move on and talk about the compatibility problems that can arise due to all these modifications.

Compatibility

As mentioned, important compatibility problems can eventually occur in Java 9 for some existing applications due to the new structure of the JDK. Compatibility issues will appear in Java 9 for applications that strictly depend on the internal structure of the JDK. Because the JDK doesn't contain the jre subdirectory anymore, every piece of code that makes use of this subdirectory will stop working as expected.

Moreover, in Java 9 the system properties `java.endorsed.dirs` and `java.ext.dirs` have been removed. This means applications that rely on these two system properties won't work correctly in Java 9.

The removal of rt.jar, tools.jar, and dt.jar files also has a negative impact on applications that rely on them.

Another compatibility problem will arise in Java 9 for source code that expects JAR URLs for naming class and resource files inside the runtime image. As mentioned, the jar URLs have been replaced with jrt URLs.

Another topic that needs some attention is related to class loaders. Table 5-1 shows the class loaders that were changed in Java 9, together with the new class loaders and the corresponding packages. Chapter 10 covers the class loaders in more detail .

Table 5-1. *The Class Loaders Changes*

Package	Old Class Loader	New Class Loader
sun.tracing.dtrace	boot	application
sun.tools.jar	boot	application
sun.security.tools.policytool	boot	application
com.sun.tracing	boot	application
com.sun.tools.script	application	boot
com.sun.tools.corba.se.idl	application	boot
com.sun.jndi.url.dns	boot	extension
com.sun.jndi.dns	boot	extension
com.sun.crypto	extension	boot

As a result, source code that that depends on the class loaders of the packages listed here may not work properly in Java 9.

Summary

This chapter covered the new modular runtime images introduced in Java 9 and the reasons behind the decision to introduce them. It compared the runtime image we had before Java 9 with the one in Java 9.

We explained in detail what every folder of the new runtime image contains and showed the content of the release file. We talked about the removal of the rt.jar, tools.jar, and dt.jar files and emphasized the consequences for existing applications. We continued by presenting the new URI scheme, which consists of a jrt URL instead of a jar URL.

The chapter conculded by summarizing the compatibility problems that may occur due to the changes implemented in JEP 220. For more information on the topics discussed in this chapter, consult the documentation of JEP 220 at `http://openjdk.java.net/jeps/220`.

Chapter 6 shows how to decouple modules using services.

CHAPTER 6

Services

A *service* is basically a piece of functionality defined by an interface or a class for which service providers exist. The role of services is to decouple tightly coupled modules and to allow loose coupling between service providers and service consumers. Using services in JDK 9 isn't mandatory, but they offer a nice solution for having decoupled modules.

Suppose we want to test the brake systems for different types of cars. A brake service could define general guidelines, legal rules, and best practices for testing the brake systems. The car manufacturers could implement their own services for testing the brake systems in their cars, because each brake system is different from model to model. These services are called *service providers*, because they provide specific implementations for the brake service. Internal tools used by car manufactureres for vizualizing and analyzing the functions in a car could use the brake service. These are called *service consumers* because they use, or consume, the service.

The basic idea behind services in Jigsaw is that in a module we don't want to expose our implementation class, but only something that's being exposed through an interface. This leads to the following question: how can we implement this in the Java Plaform Module System of Java 9? The answer is straightforward: we can use an interface as a contract.

Before digging into that answer, let's briefly look at what services are and how they work in Java 9. A *service* can consist of both interfaces and classes that specify the functionality of the service. A service provider implements a service. Multiple service providers can implement a service by providing custom implementations of it.

To provide a separation and a decoupling between service providers and service consumers, Java provides the `ServiceLoader<S>` class in package `java.util` of module java.base. This class wasn't introduced in Java 9. It existed in Java since JDK 6 but has been enhanced in JDK 9 in order to support modules. Its role is to search, find, and load all the service providers for a service of type `S`. This is performed at runtime, not at compile-time. Application code invokes only the service and doesn't refer to service providers.

Imagine that each three of the biggest German car manufacturers—Volkswagen, Daimler, and BMW—has its own service providers that implement a service called `brake system`. The service `brake system` provides the legal rules that a car brake system must fulfill. Because the brake systems produced by car manufacturers differ, each of them decided to provide its own implementations of the `brake system` rule by adhering to its prescription. Each manufacturer decided to build tools for being able to visualize the output generated by its brake systems. These tools act like service consumers and they're aware only of the service `brake system`. They're not aware of the service providers that implement the braking system service interface. What makes the interaction between service providers and service consumers possible, since they're not aware of each other? Here comes the `ServiceLoader` in play. It makes the instances of services providers available to service consumers. In our case, `ServiceLoader` makes the different implementations of the `brake system` service available to the data visualizing tools.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_6

■ **Note** `ServiceLoader`'s role is to find and load all the service providers and make them accessible to service consumers.

Now let's move to the modular world and explain how the concepts just presented fit in the new Java Platform Module System. In a modular context, we could use a provider module that has the role of registering a service on the service registry. Additionally we could make use of a consumer module that has the role of performing lookups for a service in the service registry.

■ **Note** The service registry returns a service instance to the consumer module.

Even if we use modules, the workflow is the same as we know from the non-modular world. First, a provider registers a service or an interface. Second, a consumer searches a registry for any implementations of the interface. When any implementations are found, they can call upon the service via the interface without having to know about the concrete implementation.

The ServiceLoader API is used to decouple modules. A module should depend on an interface rather than on the implementation of another module. The implementation classes should not be exported. Instead, an interface should be exported. One of the strong features of the ServiceLoader API is that the system doesn't need to know about all the service provider implementations right at compile-time. They're computed only at runtime. So at compile-time this kind of dependency between modules doesn't need to be declared inside module declarations.

Strong Coupling Between Modules

In Chapter 4 you learned what the `requires` and `exports` clauses mean. When a module requires another module, a strong coupling between the two modules occurs. If a module changes, then it may be necessary to adjust all the dependent modules. This not only makes the source code much harder to maintain, it can also considerably increase the time interval needed for implementing change requests in the code.

Let's illustrate the strong coupling between modules with an explanatory example. We know from the module declaration of module java.rmi that it requires module java.logging. Module java.rmi has a dependency on module java.logging and also has access to the public types in the API exported by the java.logging module. As a result, there's a strong and tight coupling between the modules java.rmi and java.logging. This has an important impact at both compile-time and runtime.

At compile-time, if the java.logging module isn't found, the java.rmi module can't be built. A `module not found error` will be thrown at compile-time.

At runtime, if java.logging isn't found, no application that depends on java.rmi can start. This happens because the dependency of java.rmi on java.logging isn't resolved. The following error will be thrown:

```
Error occurred during initialization of VM
java.lang.module.ResolutionException: Module java.logging not found, required by java.rmi
        at java.lang.module.Resolver.fail(java.base@9-ea/Resolver.java:841)
        at java.lang.module.Resolver.resolve(java.base@9-ea/Resolver.java:154)
        at java.lang.module.Resolver.resolveRequires(java.base@9-ea/Resolver.java:116)
        at java.lang.module.Configuration.resolveRequiresAndUses(java.base@9-ea/
        Configuration.java:311)
```

```
at java.lang.module.ModuleDescriptor$1.resolveRequiresAndUses(java.base@9-ea/
ModuleDescriptor.java:2483)
at jdk.internal.module.ModuleBootstrap.boot(java.base@9-ea/ModuleBootstrap.java:272)
at java.lang.System.initPhase2(java.base@9-ea/System.java:1927)
```

Using Services in JDK 9

This section describes what services are in Java 9 and how we can use them to prevent tight coupling between modules. Project Jigsaw can use the service registry as a layer of communication for the interaction between modules.

Modules can register their implementation class as a service in the service registry. These modules are called *service provider modules*. Their main role is to provide implementations of an interface. The *service consumer modules* use services that implement the interface that's defined in the service registry. They don't deal with the implementation classes that implement the interface defined in the service registry. Service consumer modules obtain objects from the service registry that implement the interface. In this way, they're able to successfully call methods on this interface.

■ **Note** A service consumer module and a service provider module don't have a dependency upon each other.

The interface defined in the service registry implemented by the corresponding class represents the interaction between service providers and service consumers. The service registry instantiates the classes and afterwards provides this instance to the service consumer. The service consumer only has to know about the interface. It will return an object that implements the interface. It can also call methods on this object.

■ **Note** A service can be declared as an abstract class or as an interface. However, it's better from a design point of view to use an interface rather than an abstract class. When using an abstract class instead of an interface, a public static provider method has to be defined.

Let's look at the syntax of the `uses` and `provides` clauses, which are mandatory in order to be able to implement the concepts present throughout this chapter.

Providing and Consuming Services

In this section you'll learn how to consume and provide services in JDK 9. We present the clauses used in the module declarations to declare that a module provides a service implementation and that a module uses a service.

Providing a Service

he Java Platform Module System introduced a new construct called `provides` in the module descriptor in order for a module to be able to declare that it provides and exposes a service implementation for a specific service. Figure 6-1 illustrates its syntax.

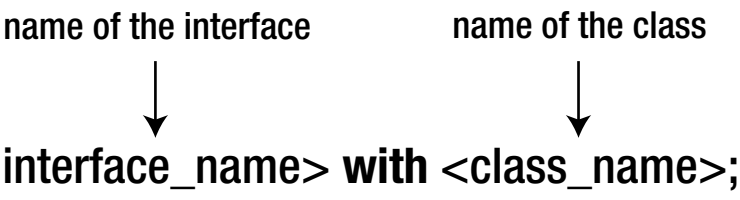

Figure 6-1. *The provides with clause*

The `provides with` clause takes two parameters:

- `<interface_name>` represents the name of the service interface. It specifies the name of the service for which the current module provides an implementation. The service could be either a class or an interface.
- `<class_name>` represents the name of the class. It specifies the name of the class that implements the service interface. This class must be present in the current module. If it is not present in the current module, we get a compilation error.

A module uses the `provides` clause to inform the ServiceLoader that it provides an implementation of a service. Without this knowledge, the ServiceLoader wouldn't have been able to load the service provider because it wouldn't have been aware of it existence.

■ **Note** JDK 9 allows you to have a service implementation as an interface. This wasn't possible in previous versions of Java.

Jigsaw also allows for a single module to provide an implementation for a service and to also consume that service. But it does not allow for more than one provides statement to specify the same interface in a module declaration. A module declaration like this will never compile:

```
module myModule {
        provides myInterface with firstClass;
        provides myInterface with secondClass;
}
```

We mentioned previously the notion of consuming a service. The next subsection explains what this means and how it can be declared in Jigsaw.

Consuming a Service

In JDK 9, a module can explicitly declare that it consumes a service. For this, the service has to be discovered. Therefore, the `uses` clause was introduced in JDK 9 in the module declaration. It takes an interface as a parameter.

Figure 6-2 shows the syntax of the `uses` clause.

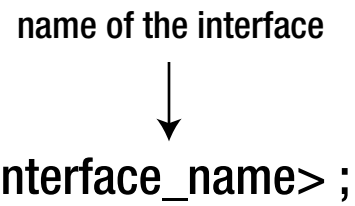

Figure 6-2. *The uses clause*

When should we use this clause? This clause should be used in modules that define a `ServiceLoader<interface_name>`, which loads service providers for the service with the name `<interface_name>`. If our module uses the `ServiceLoader` class to load services, then it's mandatory to declare this in the module's declaration using the `uses` clause, followed by the name of the service interface used.

This means that inside the module that declares the `uses` clause, a `ServiceLoader` is used, as in the following example:

```
Iterable<interface_name> ourInterfaces = ServiceLoader.load(interface_name.class);
```

Here, a `ServiceLoader` for services of type `interface_name` has been used.

■ **Note** The service declared with the `uses` clause doesn't have to reside in the same module. It can also reside in another module provided that there is readability between the two modules.

Retrieving a ServiceLoader

We've already seen how to obtain a service loader. It can be done using the `load()` method, which comes in four flavours, according to the JDK 9 API specification:

- `public static <S> ServiceLoader<S> load(Class<S> service)`

  This method creates a new service loader for the given service type. It uses the context class loader of the current thread.

- `public static <S> ServiceLoader<S> load(Class<S> service, ClassLoader loader)`

  This method creates a new service loader and uses the given class loader to locate service providers for the service. Providers are located first in named modules and then in unnamed modules. Providers are located in all named modules of the class loader or to any class loader reachable via parent delegation.

- `public static <S> ServiceLoader<S> load(ModuleLayer layer, Class<S> service)`

  This method creates a new service loader for the given service type to load service providers from modules in the given module layer and its ancestors. It doesn't locate providers in unnamed modules.

- `public static <S> ServiceLoader<S> loadInstalled(Class<S> service)`

  This method creates a new service loader for the given service type. It uses the platform class loader.

After we retrieve a `ServiceLoader`, we can iterate over all the service providers using the `iterate()` method. Another option would be to call the `stream()` method, which returns a stream to lazily load available providers. The syntax of the `stream()` method is as follows:

```
Stream<ServiceLoader.Provider<S>> stream()
```

Until now, we've seen how to retrieve a ServiceLoader and also how to provide and to consume a service. It's time to see a practical example. We'll use an example using one service consumer and one service provider. Then we'll expand the example and show how to add more service providers.

Using One Consumer and One Provider

This section shows a simple example to illustrate the concepts presented earlier. Suppose we have three modules:

- Module com.apress.moduleA contains a simple interface called ServiceExample.
- Module com.apress.providerA defines a service provider that contains the class `ServiceExampleImplementation1`, which implements the interface from module com.apress.moduleA, `ServiceExample`.
- Module com.apress.consumer defines a service consumer that creates a new service loader for the `ServiceExample` service and uses this service.

Listing 6-1 shows the interface `ServiceExample`, defined inside the module com.apress.moduleA.

Listing 6-1. The Interface ServiceExample from the Module com.apress.moduleA

```
package com.apress.moduleA.interfaces;

public interface ServiceExample {

        String printHelloWorld();
}
```

The module descriptor of the module com.apress.moduleA is depicted in Listing 6-2. The package com.apress.moduleA.interfaces, where the interface is located, is exported.

Listing 6-2. The Module Descriptor of the Module com.apress.moduleA

```
module com.apress.moduleA {

        exports com.apress.moduleA.interfaces;
}
```

Until now we've only defined an interface inside of a module. Next we'll define the provider module. Listing 6-3 shows the implementation class of the interface from the module com.apress.providerA.

Listing 6-3. The Implementation Class of Interface ServiceExample from Module com.apress.providerA

```
package com.apress.providerA;

import com.apress.moduleA.interfaces.ServiceExample;

public class ServiceExampleImplementation1 implements ServiceExample {

    public ServiceExampleImplementation() {
    }

    @Override
    public String printHelloWorld() {

        return "Hello World from ServiceExampleImplementation1";
    }
}
```

In Listing 6-4 you see the module descriptor of module com.apress.providerA.

Listing 6-4. The Module Descriptor of Module com.apress.providerA

```
module com.apress.providerA {
         requires com.apress.moduleA;
         provides com.apress.moduleA.interfaces.ServiceExample with com.apress.providerA.
ServiceExampleImplementation1;
}
```

The module descriptor states that it provides an implementation of the `ServiceExample` interface with the class `ServiceExampleImplementation1`. This means that inside the module we have a class called `ServiceExampleImplementation1` that implements the interface `ServiceExample`. The module descriptor also requires the com.apress.moduleA module because it has to access the interface in order to be able to implement it.

Listing 6-5 shows the content of module com.apress.consumer.

Listing 6-5. The Main Class of Module com.apress.consumer

```
package com.apress.consumer;

import com.apress.moduleA.interfaces.ServiceExample;
import java.util.ServiceLoader;

public class Main {

        public static void main(String[] args) {
                Iterable<ServiceExample> services = ServiceLoader.load(ServiceExample.class);
                for(ServiceExample serviceExample : services) {
                                System.out.println(serviceExample.printHelloWorld());
                }
        }
}
```

The Main class obtains instances of ServiceExample using the ServiceLoader from the java.util package. This is done by creating a new service loader for the ServiceExample type inside the Main class of the com.apress.consumer module. All instances of type ServiceExample are retrieved by calling the load() method with the ServiceExample.class parameter. Finally, we iterate through them and call the printHelloWorld() method on them.

Listing 6-6 shows the module-info.java file of the module com.apress.consumer.

Listing 6-6. The Module Descriptor of Module com.apress.consumer

```
module com.apress.consumer {
         requires com.apress.moduleA;
         uses com.apress.moduleA.interfaces.ServiceExample;
}
```

Module com.apress.consumer requires module com.apress.moduleA because it needs access to the interface to call the corresponding method on it. Additionally, it specifies that it uses the interface ServiceExample. This tells the module system that the module com.apress.consumer wants to consume instances of the com.apress.moduleA.interfaces.ServiceExample interface.

Finally, we compile the modules specified using the following command:

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

Then we run the following command:

```
java --module-path output -m com.apress.consumer/com.apress.consumer.Main
```

The output is printed inside the console:

```
Hello World from ServiceExampleImplementation1
```

We saw in this example how to define a simple service provider, a service consumer, and an interface in a separate module for the communication between the service provider and the service consumer. Both service provider and service consumer require only the interface. This means there's a dependency between the service provider and the interface and respectively a dependency between the service consumer and the interface. It's important to remember that there's no dependency between the service provider and the service consumer. As a consequence, we don't have tight coupling between those two modules.

■ **Note** You can find the source code for this example in the directory /ch06/oneConsumerOneProvider.

Using One Consumer and Two Providers

Until now we've had only one service provider, but we can define many service providers and at the same time keeping the loosely coupled relation between the service providers at the service consumers. We'll illustrate this concept with an example by defining another provider module.

Listing 6-7 shows the implementation class of the interface ServiceExample in the module com.apress.providerB.

Listing 6-7. The Implementation Class of Interface ServiceExample from Module com.apress.providerB

```
package com.apress.providerB;

import com.apress.moduleA.interfaces.ServiceExample;

public class ServiceExampleImplementation2 implements ServiceExample {

    public ServiceExampleImplementation2() {
    }

    @Override
    public String printHelloWorld() {
        return "Hello World from ServiceExampleImplementation2";
    }
}
```

Listing 6-8 shows the module descriptor of module com.apress.providerB.

Listing 6-8. The Module Descriptor of Module com.apress.providerB

```
module com.apress.providerB {
         requires com.apress.moduleA;
         provides com.apress.moduleA.interfaces.ServiceExample with com.apress.providerB.
         ServiceExampleImplementation2;
}
```

The module descriptor of module com.apress.providesB provides an implementation of the `ServiceExample` interface with the class `ServiceExampleImplementation2`.

By compiling and running the modules, the following result is printed in the console:

```
Hello World from ServiceExampleImplementation2
Hello World from ServiceExampleImplementation1
```

In this example, we defined two provider modules and saw how they interact in the context of the module system. None of the provider modules has a dependency upon the consumer module.

In our example, neither the provider modules nor the consumer module exports their packages. In this way, they're encapsulated and can't be accessed from outside. Nonetheless, Jigsaw has the capability to instantiate classes of type `ServiceExampleImplementation1` because it implements the interface `ServiceExample`, which is defined using the `provides` directive inside the module-info.java of module com.apress.providerB.

■ **Note** You can find the source code for this example in the directory /ch06/oneConsumerTwoProviders.

Summary

In this chapter we talked about services. Services are used to decouple modules by specifying a contract in form of an interface. They allow loose coupling between service consumer and service providers. The concept of loose coupling is very important in software development, especially when we're talking about large software applications. We showed how to declare that a module provides a service by using the new construct `provides … with` inside the module descriptor module-info.java. Afterwards, we talked about how to declare that a module consumes a service by using the new construct `uses` inside the module descriptor. Further, we discussed how to retrieve a `ServiceLoader`.

We looked at two examples of how we can define one service consumer and one service provider, respectively one service consumer and two service providers.

In Chapter 7 you'll learn about the Jlink tool, which lets us create custom runtime images that contain only the modules we need.

CHAPTER 7

Jlink: The Java Linker

Throughout the software development process, we may encounter situations in which we need targeted Java Runtime Environments (JREs) for the operating system we're using. The reasons for this are various: we may want to achieve a better degree of performance or we may have some customized libraries that only work on a specific operating system.

For instance, when using microservices, we may not want to use the entire JDK, but rather just a part of it. Microservices are small and typically don't use libraries from the whole JDK.

Jlink helps solve these problems by creating a customized, targeted version of the JRE that is specific to an operating system and contains only the modules we need.

The Java Linker

Java 9 introduces a new tool for dynamically linking modules, called Jlink. Its role is to assemble a group of modules in order to create a runtime image. During the assembly process, different optimizations across module boundaries can be applied. The following sections cover some methods for performing this kind of optimizations.

Jlink starts with the modules we specify and searches recursively for all the `requires` statements inside the module descriptors of the specified modules. In this way, Jlink can find all the modules needed to be assembled inside a new custom runtime image.

A runtime image created by the Jlink tool contains the minimum number of required modules together with their dependencies. We can also specify which modules we want to be added to the runtime image. As a result, a platform-specific binary executable is created.

Jlink can take the following types of files as input:

- Modular JAR files
- JMOD files
- JAR files
- Class files

The following types of files can represent the output of the Jlink tool:

- JMOD files
- JAR files
- JVM images
- Custom JREs

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_7

The Jlink tool can generate these four kinds of files at link time. Jlink can even create custom JVM images. Figure 7-1 illustrates the types of input files that are accepted by the Jlink tool and also which kind of output files it can create.

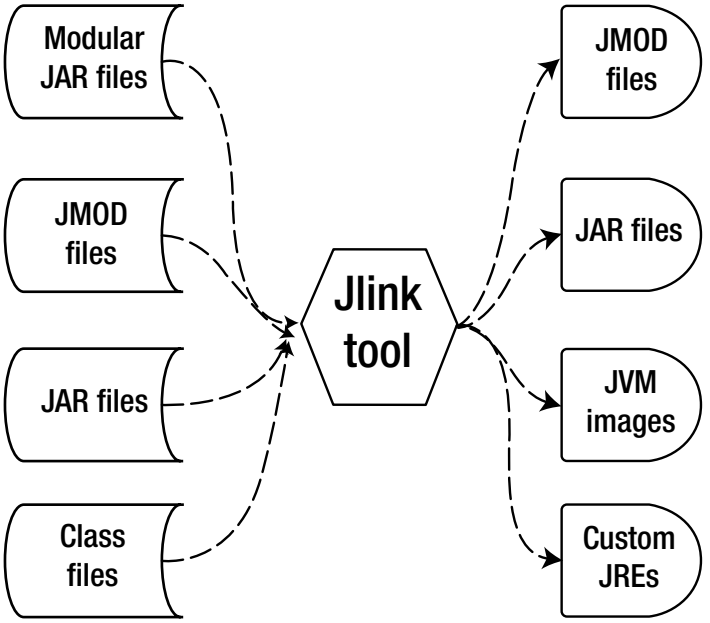

Figure 7-1. *Types of input and output files for the Jlink tool*

The Jlink tool acts like a transformer of code and runs the resolution process in order to compute the minimum possible set of modules needed for creating the runtime image. The minimum runtime image we can hypothetically create is a runtime image that contains only the base module, java.base. Jlink is a tool used at build-time that can cross-compile and use cross-module optimizations, but it can't create cross-platform executable files.

You may ask how Jlink knows what our target platform is. It knows that based on the type of the platform modules we're attempting to link. For example, if we're working on a Linux operating system and are passing a module path to a Windows distribution, Jlink will link the Windows versions of these modules and create a Windows runtime image.

■ **Note** The module path we specify when we use Jlink has to be the module for the target platform.

There is no change needed at code level in order to use Jlink. Jlink assembles the modules into a custom runtime image and doesn't modify the module-info.class files at all.

Jlink Images

Jlink images are specific to each operating system. They represent a customization of the JDK and the JRE. We can consider them as custom runtime images of a JRE that contains only the modules our application needs. Jlink creates a custom directory that can be used to run our Java application.

A Jlink image contains the following directories:

- bin
- conf
- include
- legal
- lib
- release

Jlink also links modules that have the directive `requires static myModule`. At runtime, if the module myModule is located on the module path, the `requires static` clause will be satisfied. If we have an application that uses only the module java.base and nothing else, then our custom runtime image will consist only of our application modules plus the java.base module.

You'll learn more about Jlink images and Jlink command syntax later in this chapter as we create a custom runtime image.

Jlink Command Syntax

To create a custom runtime image using Jlink, the `jlink` command is used together with the necessary options. Figure 7-2 shows the syntax of this command together with its most important options.

- `[jlink_options]` specifies a set of options separated by spaces. You'll learn about these options in the next section.
- The `--module-path` option specifies the location of the modules that should be discovered by the Jlink tool. They can be exploded modules, modular JAR files, or JMOD files.
- The `--add-modules` option specifies the names of the modules to add to the runtime image. The modules specified will be added to the runtime image together with their transitive dependencies.
- The `--output` option specifies the directory where the custom runtime image will be created.

jlink [jlink_options] --module-path <module_path> --add-modules <modules_list> --output <directory>

Figure 7-2. *The syntax of the jlink command*

As mentioned, the Jlink tool receives the module path showing where to find the modules. It finds the modules by starting the module resolution process that searches for all the transitive dependencies of each module until the bottom module, java.base, is reached. Provided that we add a module using the `--add-modules` command-line option, Jlink will search for all its `requires` and `requires static` clauses and add all the corresponding dependent modules into the custom runtime image.

■ **Note** The Jlink tool allows cross-linking if we have the JMOD files for the target platform.

Jlink Command Options

The Jlink tool isn't limited to the options discussed in the preceding section. According to the official JDK 9 API specification, it allows many other options, all of which are shown in Table 7-1.

Table 7-1. *The Options of the Jlink Command*

Option Name	Description
`--help`	Prints a help message.
`--module-path <module_path>`	Defines the module path.
`--limit-modules <list_of_modules_names>`	Limits the group of observable modules to those in the transitive closure of the modules specified. If there are any modules specified using the `--add-modules` option, they'll be also added to the observable modules, even if not present in the list of `--limit-modules`. The main module, if it exists, will also be added to the observable modules.
`--add-modules <module_name>`	Specifies the root modules that need to be resolved.
`--output <directory_name>`	Specifies the name of the directory where the runtime image will be generated.
`--launcher <command_name>=<module_name>`	Specifies the launcher command name for the module.
`--launcher command=<module_name>/main`	Specifies the launcher command name for the module together with the `main` class.
`--endian <little | big>`	Defines the byte order of the runtime image that is being generated.
`--version`	Displays the version information.
`--save-opts <name_of_file>`	Saves the Jlink options in the given file.
`--strip-debug`	Strips the debug information.
`--no-man-pages`	Excludes the `man` pages.
`--no-header-files`	Excludes the header files.
`--disable-plugin <name_of_plugin>`	Disables the plugin.
`--list-plugins`	Lists all the available Jlink plugins that are reachable using the command-line.
`--ignore-signing-information`	Overcomes an error when signed modular JARs are linked to the runtime image.
`@<name_of_file>`	Reads all the options from the file specified as an argument.
`--bind-services`	Executes a full service binding and links in service provider modules and their dependences into the runtime image.
`--suggest-providers <list_of_names_of_service>`	Helps to find providers that implement the service types from the module path.
`--verbose`	Enables verbose tracing.

Link Phase

A new *link phase* was introduced in Java 9. Its role is to produce a runtime image by assembling a collection of modules together with their transitive dependencies. OpenJDK states that the "link time is an opportunity to do whole-world optimizations that are otherwise difficult at compile or costly at runtime."

Linking is a new development phase that has been added throughout the Java development process. It's very important to remember that the linking phase is optional—you don't have to use it if you don't want or need to.

Linking can be used in Java 9 when we have a modular application. The resulting image is platform independent. The linker can link two or more targets—for example, if you're using operating system A, you can successfully target operating system B provided that your module path owns the modules for operating system B rather than the ones for operating system A. But it's not possible to link a WAR file together.

The jdk.jlink Module

Figure 7-3 shows the module graph that contains the jdk.jlink module together with its dependencies.

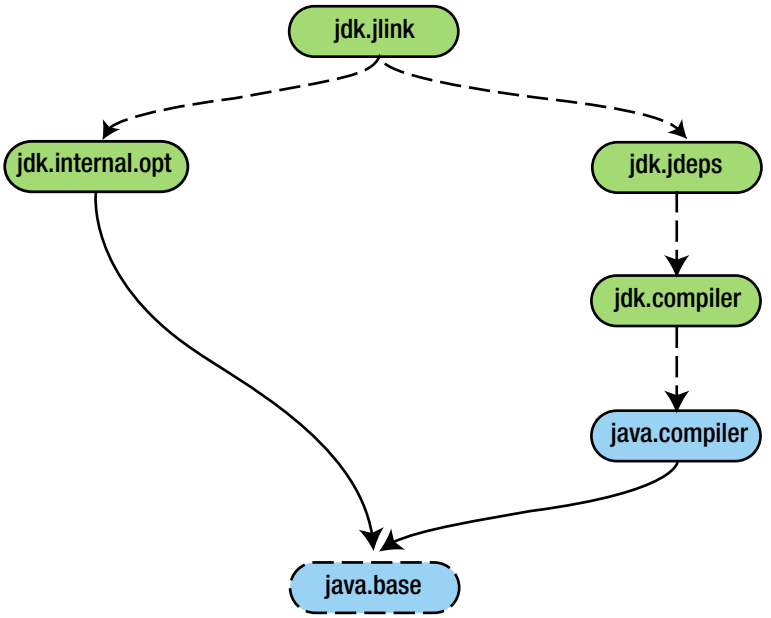

Figure 7-3. *The module graph of the internal module jdk.jlink*

The module jdk.jlink contains different java classes inside the tools directory. There is a jimage directory, jlink directory, and jmod directory inside the tools directory. The module jdk.jlink has dependencies on the jdk.internal.opt and the jdk.jdeps modules, as expressed in its module descriptor, the module-info.java file. The jdk.jdeps module has a dependency on the jdk.compiler module, which has a dependency on the java.compiler module. Finally, the java.compiler module requires the java.logging module.

■ **Note** Module jdk.jlink isn't designed to be used by programmers.

Figure 7-3 shows the non-standard JDK modules (jdk.jlink, jdk.internal.opt, jdk.jdeps, and jdk.compiler) and the standard JDK modules (java.compiler, java.logging and java.base). As in the module graph shown in Chapter 3, the solid lines represent implied readability, and the dashed lines represent simple readability between modules.

Listing 7-1 shows the module descriptor of the jdk.jlink module.

Listing 7-1. The Module Descriptor of the jdk.jlink Module

```
module jdk.jlink {
    requires jdk.internal.opt;
    requires jdk.jdeps;

    uses jdk.tools.jlink.plugin.Plugin;

    provides java.util.spi.ToolProvider with
        jdk.tools.jmod.Main.JmodToolProvider,
        jdk.tools.jlink.internal.Main.JlinkToolProvider;

    provides jdk.tools.jlink.plugin.Plugin with
        jdk.tools.jlink.internal.plugins.StripDebugPlugin,
        jdk.tools.jlink.internal.plugins.ExcludePlugin,
        jdk.tools.jlink.internal.plugins.ExcludeFilesPlugin,
        jdk.tools.jlink.internal.plugins.ExcludeJmodSectionPlugin,
        jdk.tools.jlink.internal.plugins.LegalNoticeFilePlugin,
        jdk.tools.jlink.internal.plugins.SystemModulesPlugin,
        jdk.tools.jlink.internal.plugins.StripNativeCommandsPlugin,
        jdk.tools.jlink.internal.plugins.OrderResourcesPlugin,
        jdk.tools.jlink.internal.plugins.DefaultCompressPlugin,
        jdk.tools.jlink.internal.plugins.ExcludeVMPlugin,
        jdk.tools.jlink.internal.plugins.IncludeLocalesPlugin,
        jdk.tools.jlink.internal.plugins.GenerateJLIClassesPlugin,
        jdk.tools.jlink.internal.plugins.ReleaseInfoPlugin,
        jdk.tools.jlink.internal.plugins.ClassForNamePlugin;
 }
```

You can see in this module descriptor the entire list of plugins the Jlink tool has.

Example: Create a Runtime Image Using Jlink

This section shows an example of creating a custom runtime image. Our small application saves a text message inside a file and then inside a database.

We define four modules and use the ServiceLoader API again, as in Chapter 6. Why the ServiceLoader API again? Because, and we want to stress this, Jlink doesn't provide service binding by default. That means that by default Jlink doesn't add the modules observed using the "use" and "provides" clauses to the runtime image. It only adds the modules specified using the `requires` clauses. However, the JCP team enhanced the Jlink tool with an option called `--bind-services` that does service binding and links in service provider modules and their dependences.

Listing 7-2 shows the interfaces `DatabasePersistenceService` and `FilePersistenceService` of the module com.apress.service.

Listing 7-2. The Interfaces DatabasePersistenceService and FilePersistenceService of the Module com.apress.service

```
// DatabasePersistenceService.java
package com.apress.service.interfaces;

public interface DatabasePersistenceService {

    void saveMessageIntoDatabase(String message);
}

// FilePersistenceService.java
package com.apress.service.interfaces;

public interface FilePersistenceService {

    void saveMessageIntoFile(String message);
}
```

These interfaces contain method definitions for saving a message into the database and saving a message into a file, respectively.

Listing 7-3 shows the module descriptor of module com.apress.service, which simply exports the package com.apress.service.interfaces.

Listing 7-3. The Module Descriptor of Module com.apress.service

```
module com.apress.service {

    exports com.apress.service.interfaces;
}
```

Listing 7-4 shows the class `DatabasePersistenceProvider` of the module com.apress.databasepersistence.

Listing 7-4. The Class DatabasePersistenceProvider of the Module com.apress.databasepersistence

```
package com.apress.databasepersistence;

import com.apress.service.interfaces.*;
import java.sql.*;

public class DatabasePersistenceProvider implements DatabasePersistenceService {

    private Connection connection;
    private static final String JDBC_URL = "jdbc:postgresql://localhost/myDatabase";

    public void saveMessageIntoDatabase(String message) {

        String insertSql = "INSERT INTO MESSAGES(CONTENT) VALUES(" + message + ")";
```

```
        try {
            connection = DriverManager.getConnection(JDBC_URL, "root", "password");
            Statement statement = connection.createStatement();

            int result = statement.executeUpdate(insertSql);
             if (result > 0) {
                System.out.println("Message successfully saved into the database");
            } else {
                System.out.println("Message could not be saved into the database");
            }
            } catch (SQLException sqlException) {
            sqlException.printStackTrace();
        }
    }
}
```

This class is a service provider that implements the method `saveMessageIntoDatabase()` of the interface `DatabasePersistenceService`. It also imports the package java.sql. The role of the class is to simply store a string into a database using JDBC.

Listing 7-5 shows the module descriptor of module com.apress.databasepersistence.

Listing 7-5. The Module Descriptor of Module com.apress.databasepersistence

```
module com.apress.databasepersistence {

    requires com.apress.service;
    requires java.sql;

    provides com.apress.service.interfaces.DatabasePersistenceService with com.apress.
databasepersistence.DatabasePersistenceProvider;
}
```

This module requires the com.apress.service module as it uses its interfaces. It also requires module java.sql because it uses types from this module. Nevertheless, it declares that it provides the implementation of the `DatabasePersistenceService` interface with the `DatabasePersistenceProvider` class.

Listing 7-6 shows the class `FilePersistenceProvider` of the module com.apress.filepersistence.

Listing 7-6. The Class FilePersistenceProvider of the Module com.apress.filepersistence

```
package com.apress.filepersistence;

import com.apress.service.interfaces.*;
import java.io.*;

public class FilePersistenceProvider implements FilePersistenceService {

    private static final String FILENAME = "C:\\Java9\\example.txt";

    // for Linux
    // private static final String FILENAME = "Java9/example.txt";
    private BufferedWriter bufferedWriter = null;
    private FileWriter fileWriter = null;
```

```
    public void saveMessageIntoFile(String message) {

        try {

            fileWriter = new FileWriter(FILENAME);
            bufferedWriter = new BufferedWriter(fileWriter);
            bufferedWriter.write(message);
        }
        catch (IOException e) {

            e.printStackTrace();
        }
        finally {
            try {
                if (bufferedWriter != null)
                    bufferedWriter.close();

                if (fileWriter != null)
                    fileWriter.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

This class is a service provider that implements the method `saveMessageIntoFile()` of the interface `FilePersistenceService`. The role of the class is to save a string into a file using Java I/O.

Listing 7-7 shows the module descriptor of module com.apress.filepersistence.

Listing 7-7. The Module Descriptor of Module com.apress.filepersistence

```
module com.apress.filepersistence {

    requires com.apress.service;
    provides com.apress.service.interfaces.FilePersistenceService with com.apress.
filepersistence.FilePersistenceProvider;
}
```

This module requires the com.apress.service module and declares that it provides the implementation of the `FilePersistenceService` interface with the `FilePersistenceProvider` class.

Listing 7-8 shows the `Main` class located in the module com.apress.application.

Listing 7-8. The Main Class of Module com.apress.application

```
package com.apress.application;

import com.apress.service.interfaces.*;
import java.util.ServiceLoader;

public class Main {

        public static void main(String[] args) {
```

```
                FilePersistenceService filePersistenceService = ServiceLoader.load
                (FilePersistenceService.class).iterator().next();
                filePersistenceService.saveMessageIntoFile("First message saved into the
                file");

                DatabasePersistenceService databasePersistenceService = ServiceLoader.load
                (DatabasePersistenceService.class).iterator().next();
                databasePersistenceService.saveMessageIntoDatabase("Second message saved
                into the database");
        }
}
```

This class uses the ServiceLoader API to load the interfaces `FilePersistenceService` and `DatabasePersistenceService`. It then calls the corresponding methods on the interfaces. As a result, the message is saved in the text file, respectively saved into the database.

Listing 7-9 represents the module descriptor of module com.apress.application.

Listing 7-9. The Module Descriptor of module com.apress.application

```
module com.apress.application {

    requires com.apress.service;

    uses com.apress.service.interfaces.FilePersistenceService;
    uses com.apress.service.interfaces.DatabasePersistenceService;
}
```

We notice that com.apress.application represents our main module. We compile all the Java classes using the `javac` command:

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

As a result, the output directory will contain all the compiled classes of our four modules that we have. Further, we create a custom runtime image using the Jlink tool by running the following command (in our case, on the Windows operating system):

```
jlink --module-path "output;$JAVA_HOME/jmods" --add-modules com.apress.application --output
runtimeImage
```

We specify two directories, separated by semicolon, for the module path. The output directory contains the class files of all our modules. `JAVA_HOME` is an environment variable that points to the current installation of Java 9. Inside `JAVA_HOME` is the JMODS directory, which contains the modules of the JDK. Remember, we have to explicitly point the module path at the JDK we intend to produce a runtime image for.

Using the `--add-modules` option, we specify the name of the main module. Finally, the output option indicates the directory where the new custom runtime image will be created.

The Jlink tool first analyzes the module descriptor file of the module com.apress.application and then recursively adds any modules that are required inside the runtime image. The runtimeImage directory now contains the runtime that we've just created. We can see what modules the runtime image contains by running the `java --list-modules` command inside the runtimeImage directory:

```
./bin/java --list-modules
```

The result is:

```
com.apress.application
com.apress.service
java.base
```

Our runtime image contains those three modules. The module java.base is the single platform module that's always being automatically added to our runtime image. Further, we have two more application modules in our runtime image: com.apress.application, which is the root module that we specified using the `--add-options` option, and com.apress.service module. The com.apress.application module requires the module com.apress.service in its module descriptor. As a result, we have the module com.apress.service in our runtime image.

■ **Note** The modules com.apress.databasepersistence and com.apress.filepersistence aren't present in our newly created runtime image, even if they've been previously successfully compiled inside the output directory. Further, the java.sql module should also be present in the runtime image, because it's required by the module com.apress.databasepersistence. The reason why these modules aren't in the runtime image is that the Jlink tool doesn't do service binding by default. The tool identifies the necessary modules by searching for the `requires` clauses inside the module descriptors. It doesn't search for the uses or provides clauses either. In our module com.apress.application, we don't require the modules com.apress.databasepersistence and com.apress.filepersistence. As a result, they're not added to the runtime image.

To solve this, there are two solutions. The first is also the simplest: use the `new --bind-services` option added to the `jlink` command. This option makes a full service binding. Therefore, it searches for all the `uses` clauses and then, for all the services specified by the `uses` clauses, it adds all the service provider modules in the runtime image. We could use it like this:

```
jlink --module-path "output;$JAVA_HOME/jmods" --bind-services
        --add-modules com.apress.application
        --output runtimeImage
```

A second solution to add them to the runtime image would be to specify them inside the `--add-modules` command-line option, but this is a workaround that was mostly used before the introduction of the `--bind-services` option. Using the workaround with the `--add-modules` option is definitely more costly than using the `--bind-services` option. Hence, we talk about this second solution here. We delete our runtime image and supplementary add the explicit `--add-modules` flags for each of the service provider modules that we want to be added into our runtime image:

```
jlink --module-path "output;$JAVA_HOME/jmods"
        --add-modules com.apress.application
        --add-modules com.apress.databasepersistence
        --add-modules com.apress.filepersistence
        --output runtimeImage
```

If we omit to mention the JMODS directory in our module path, an error will be thrown:

```
Error: module java.sql not found, required by com.apress.databasepersistence
```

This error states that the module java.sql, which is required by the module com.apress.databasepersistence, wasn't found on the module path because we didn't include its location, the JMODS directory, on the module path.

We explicitly specified that we want the modules com.apress.databaspersistence and com.apress.filepersistence to be added to the runtime image, next to the main module com.apress.application. If we run the `--list-modules` command inside the runtimeImage directory, we get the following result:

```
./bin/java --list-modules
com.apress.application
com.apress.databasepersistence
com.apress.filepersistence
com.apress.service
java.base
java.logging
java.sql
java.xml
```

Note that the new runtime image not only contains the two service provider modules that we specified using the `--add-modules` flags, but also contains three new modules: java.logging, java.sql, and java.xml. The reason is that the module com.apress.databasepersistence requires java.sql, so this module is also added to the runtime image. But module java.sql also requires the modules java.logging and module java.xml. Therefore, these two modules are also added to the runtime image in order for all the dependencies to be fulfilled.

Our runtime image has a structure similar to that of a JRE. The newly created runtimeImage directory has the following structure:

```
-bin (directory)
-conf (directory)
-include (directory)
-legal (directory)
-lib (directory)
-release (file)
```

The bin directory has the following content:

```
-server (directory) => contains a jvm.dll file
-java.dll (file)
-java.exe (file)
-javaw.exe (file)
-jimage.dll (file)
-jli.dll (file)
-keytool.exe (file)
-msvcp120.dll
-msvcr120.dll
-net.dll
-nio.dll
-verify.dll
-zip.dll
```

The conf directory has the following structure:

```
-net.properties (file)
-security (folder)    => -policy (folder)
                                -java.policy (file)
                               -java.security (file)
-net.properties (file)
```

The include directory has the following structure:

```
-win32 (folder) => -jni_md.h
-classfile_constants.h
-jni.h
-jvtmi.h
-jvmticmlr.h
```

The legal directory has the following content:

```
-java.base (folder) => aes.md, asm.md, cldr.md, icu.md, zlib.md
-java.logging => COPYRIGHT
-java.sql => COPYRIGHT
-java.xml => bcel.md, COPYRIGHT, dom.md, jcup.md, xalan.md, xerces.md, xmlresolver.md
```

The lib directory has the following content:

```
-security (folder) => blacklist, blacklisted.certs, cacerts, default.policy,
trusted.libraries
-server (folder) => Xusage.txt
-classlist (file)
-jrt-js.jar
-jvm.cfg
-jvm.lib
-modules (file)
-tzdb.dat
-tzmappings (file)
```

■ **Note** We used Windows 7 Professional Edition as the operating system while creating this runtime image using Jlink. That's why the custom runtime image is specific to the Windows operating system.

Jlink doesn't perform service binding by default. That means we have two distinct options:

- Use the `--bind-services` option in order for the service provider modules to be explicitly discovered and added to the runtime image
- Define each service provider module that we want to add in the `--add-modules` option

Next we check the size of the newly generated runtime image:

```
$ du -hs
48M
```

Our entire runtime image has a size of 48 MB, which is much smaller than the size of the entire JDK.

We delete our runtime image and generate a new one using compression in order to reduce its size:

```
jlink --module-path "output;$JAVA_HOME/jmods"
        --add-modules com.apress.application
        --add-modules com.apress.databasepersistence
        --add-modules com.apress.filepersistence
        --compress=2
        --output runtimeImage
```

We used level 2 compression. You'll learn more about this later in the chapter during the discussion of the compress plugin. The new runtime image has a total size of 29 MB. Thanks to compression, we managed to reduce the size of the runtime image by more than 40 percent. It's recommended to use compression in order to create more compact runtime images, especially when we need to install them on small devices.

We can further reduce the size of the runtime image by using the `--strip-debug` flag, which strips the debug information from the image. In our example, the new size of the runtime image will be 26 MB instead of 29 MB after taking advantage of the `--strip-debug` option.

■ **Note** If we use the Windows operating system, we have to use the separator : instead of ; for the module path.

Running the Runtime Image

To run the image, we make use of the Java launcher inside the runtime image:

```
$ ./bin/java -m com.apress.application/com.apress.application.Main
```

We pointed to the bin folder located inside the runtime image and called the Java launcher with the `-m` option, which contains the module name together with the name of the `Main` class. We specified that we want to run the `Main` class from module com.apress.application.

Note that inside the runtime image we have an executable that's targeted at a particular platform (in our case, Windows).

Modular JAR Files as Input for the Jlink Tool

We stated at the beginning of this chapter that the Jlink tool can take as input the following file formats: modular JAR files, JAR files, class files, and JMOD files.

Until now, we've put class files and JMOD files on the module path. In this example, we'll use modular JAR files instead of expanded class files. First, we delete our runtime image because we'll create a new one using modular JAR files and JMOD files. For each module, we create modular JAR files in the output directory using the following pattern:

```
$ jar --create --file output/com.apress.application.jar --main-class com.apress.application.
Main -C output/com.apress.application.
```

After all four modular JAR files are created inside the output directory, we delete all class files from this directory. Further, we run exactly the same `jlink` command as previously in order to create a custom runtime image.

The newly created runtime image created from modular JAR files is the same as the runtime image created previously from expanded class files.

Structure of the Generated Runtime Image

The runtime image we've previously created is a smaller implementation of Java made exactly for our code to be able to successfully run. It doesn't have anything else in it except the minimum libraries necessary to be able to run. The content of the generated runtime image was discussed in the previous section. This section focuses on details regarding each folder of the runtime image.

In the bin directory there are three Java launchers. The keytool is used for managing certificates and other security-related stuff. Java is the launcher we already know.

- In the lib directory are all the classes and resources, and there is no rt.jar file.
- The conf directory contains the user configuration. All the files in this directory can be edited by the user.
- The legal directory contains the copyright stuff from Oracle. There are per module license files. When using Jlink to create our own runtime images, the legal stuff comes from the packaged modules.

In order to find all the modules contained inside a runtime image, we can run `java --list-modules` inside the image, as repeatedly shown in the previous examples.

No Support for Linking Automatic Modules

Project Jigsaw doesn't offer support for linking automatic modules. Trying to add a module that doesn't contain a module-info.class file in the runtime image will result in an error.

In our next example, we download and add the guava.jar inside the output directory. Then we attempt to create the runtime image by additionally specifying the Guava JAR file:

```
--add-modules guava
```

Because the guava.jar is a simple JAR and not a modular one, and therefore doesn't contain a module-info.class inside it, the following error will be raised:

```
Error: module-info.class not found for guava module
```

There's a justified reason for not adding automatic modules support in Jlink. As we know, the automatic modules can access the class path. As a result, no one can assume that a custom runtime image created by Jlink will work correctly if it also contains automatic modules in it. Because automatic modules are employed for performing migration, they may have references to types on the class path. If, for example, an automatic module is linked into a custom runtime image, it could eventually result in faulty references unless it's used with the class path.

To make automatic modules work in Jlink, we should add module-info.class files to each of the existing JAR files. For this, we could use the JDeps tool with the option `--generate-module-info`.

■ **Note** You'll find the source code for this example in the directory /ch07/jlink.

So far, we've seen how to create a runtime image and looked at the structure of it. The Jlink tool has a couple of useful plugins you can use. The next section talks about three of them.

Jlink Plugins

The Jlink tool is plugin-based. Open JDK states that "Jlink gathers the classes, native libraries, and configuration files into a set of resources. These resources are fed through a pipeline of transformers, which are defined by plugins."

Jlink contains a couple of important plugins that can be extended by developers. We can develop our own Jlink plugins for optimizing the runtime images, for example. Table 7-2 lists the existing JLink plugins, according to the official JDK 9 API specification.

Table 7-2. *The Jlink Plugins*

Plugin	Description
`--class-for-name`	Optimizes classes by converting `Class.forName()` calls to constant loads.
`--compress=<0 | 1 | 2> [: filter=pattern-list]`	Enables the compression of resources.
`--strip-debug`	Strips the debug information from the output image.
`--strip-native-commands`	Excludes native commands from the image.
`--vm={client | server | minimal | all}`	Selects the HotSpot VM in the output image. Default is `all`.
`--generate-jli-classes=@filename`	Takes a file hinting to Jlink what java.lang.invoke classes to pre-generate. If you don't specify this flag, Jlink generates a default set of classes.
`--include-locales=langtag[,langtag]*`	Includes the list of locales.
`--dedup-legal-notices=[error_if_not_same_content]`	De-duplicates all legal notices.
`--exclude-files=[pattern_list]`	Specifies the files to exclude.
`--exclude-jmod-section`	Specifies a JMOD section to exclude.
`--exclude-resources`	Specifies the resources to exclude.
`--order-resources`	Specifies a file listing the `java.lang.invoke` classes to pre-generate.
`--release-info=<file> | add:<key1>=<value1> | del:<key list>`	Loads release properties from the supplied file. `add` is used to add properties to the release file. `del` is used to delete the list of keys in the release file.
`--system-modules`	Represents a fast loading of module descriptors. It's always enabled.

The next subsections cover three Jlink plugins: the compress, release-info, and exclude-files plugins.

The compress Plugin

The compress plugin has the role of compressing all the resources in the output image. The syntax of the plugin is simple:

```
--compress=<level of compression>
```

There are a total of three levels of compression:

- *Level 0*: Constant string sharing
- *Level 1*: ZIP compression
- *Level 2*: Both constant string sharing and ZIP compression of image classes

Level 0 scans the image classes' constant pool. Level 1 performs a ZIP compression of the image classes. Level 2 comprises both levels 0 and 1.

Earlier in this chapter we explained how to create a runtime image. In Table 7-3 you'll find a comparison of the size of our runtime image by using different levels of compression.

Table 7-3. *Size of Our Runtime Image Using Different Levels of Compression*

Compression Level	Size of the Runtime Image
No compression	48 MB
Level 0	48 MB
Level 1	38 MB
Level 2	29 MB

As you can see, Level 2 compression is the most performant. If we specify a level of compression other than 0, 1, or 2, an error will be thrown during the creation of the runtime image.

The release-info Plugin

The release-info plugin prints useful information regarding the image. In this example, we want to see what the "release" file of our runtime image contains:

```
runtimeImage>cat release
OS_NAME="Windows"
MODULES="java.base com.apress.service com.apress.application java.logging comapress.
filepersistence java.xml java.sql com.apress.databasepersistence"
OS_VERSION="5.2"
OS_ARCH="amd64"
JAVA_VERSION="9"
JAVA_FULL_VERSION="9-ea"
```

The release file contains different properties. The `MODULES` property specifies all the modules existing in the image. We can add new properties inside the release file using the following command:

```
--release-info add:<key>=<value>
```

It's also possible to delete properties files from the release file by specifying the list of keys to be deleted:

```
--release-info del:<list of keys>
```

In the following example, we build our runtime image by adding a key called `date` inside the release file:

```
jlink --release-info add:date=17.07.2017 ............
```

The release file now contains our newly added key:

```
$ cat release
date=19.03.2017
...
```

The excludes-files plugin

The excludes-files plugin lets us exclude files from the runtime image. It gets as parameter a pattern.

Now we want to exclude all the *.diz files from our runtime image. The *.diz files are compressed debug information files. Because our image is inflated by all these debug files, getting rid of them is worth it:

```
--exclude-files *.diz
```

As a result, our runtime image doesn't contain any *.diz files anymore, because we excluded them.

Summary

Jlink started as a command-line utility to generate runtime images, but after a while became a standard. Jlink is useful and suitable especially when we intend to create a targeted executable.

This chapter explained what Jlink is, described the newly introduced link phase, and presented the syntax of the Jlink command together with its options. It described the jdk.jlink module and looked inside its module descriptor. After that it showed a clear example of creating a custom runtime image that contains four application modules and four JDK modules. For this, we used the ServiceLoader API. We also saw how to run the runtime image and how to create a runtime image using modular JAR files as an input instead of class files.

Next we talked about the structure of the generated runtime image, and we explained the reasons behind Jlink's lack of support for automatic modules. The next section covered the existing Jlink plugins and showed practical examples using the compress, release-info, and excludes-file plugins. We saw how we can compress and strip debug a runtime image in order to reduce its size.

We managed to create a smaller, more compact, and tailored runtime image that we can distribute or run. We learned that if we use the advantages that Jlink gives us, we don't have to install the entire JDK, as the targeted binary Jlink creates is smaller than the JDK.

The next chapter talks about a very complex and important topic: migration.

CHAPTER 8

Migration

This chapter covers key concepts and tools used to ease migration to JDK 9. It covers common issues that can occur when we migrate existing Java applications to JDK 9 and suggests solutions and tips for solving migration problems.

First, why do we need migration? The answer is obvious: without migrating to JDK 9, we can't use the powerful features introduced by Jigsaw, nor can we use the other features introduced in Java 9 by the other JEPs, like the following:

- The Java Shell
- The updates in the Process API
- The HTTP 2 client
- The Stack-Walking API
- The Platform Logging API
- The multi-release JAR files

By looking back in the history of Java, every time a new version of Java SE has been released, there were some changes that caused incompatibilities with the previous versions of Java. The supreme scope of Oracle has always been to provide backward compatibility as much as possible. Hence, the modularization of the JDK is such a disruptive change that backward compatibility can't be 100 percent assured. Oracle struggled to offer the highest possible degree of backward compatibility, but there are some breaking changes that can affect the backward compatibility. This all depends on how our code is structured. Before starting talking about any compatibility issues, we must outline two very important things:

- Code that uses internal JDK APIs might not work in JDK 9. Some changes may be necessary.
- Code that uses only official Java SE platform APIs and supported JDK-specific APIs works in JDK 9 without any necessary change.

When deciding to migrate to Java 9, it's important to know the outcome we want to achieve:

- We want our existing Java application to simply run on JDK 9, but we don't want to have any modules defined inside our code.
- We want to modularize only a part of our application and keep the other part not modularized.
- We want to modularize the entire application.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_8

We'll explain each case in detail. For each case, suppose we have a Java application written in a version lower or equal to Java 8 and that we want to compile and run it using Java 9.

The first case involves only assuring that our application works on Java 9, without creating any modules. This means we stay on the class path and don't use the newly introduced module path at all. Start by setting the `JAVA_HOME` environment variable to point to a JDK 9 installation and then compile and run our application without performing any changes inside the code. Our application most likely works on Java 9 if it's not using JDK internal APIs. Most of the JDK internal APIs have been encapsulated in Java 9 and therefore can't be accessed. A small number of the JDK internal APIs, the ones from the module jdk.unsupported, are still accessible, but all the other are inaccessible. When we talk about JDK internal APIs, we refer to both our application code and library code. It doesn't matter if our application doesn't make use of internal JDK APIs because if one of our libraries that we're using inside our application makes use of JDK internal APIs, then our application will break anyway. Nevertheless, there are a couple of changes performed in JDK 9 that can eventually break our application, like the new versioning scheme or the new structure of the JDK and the JRE. These changes will be covered in detail throughout this chapter. The biggest disadvantage on relying just on the class path is that we can't use two of the most important features brought by JDK 9: reliable configuration and strong encapsulation. That means we can't, for instance, declare dependencies on other modules, we can't hide the internals of parts of our application, and we also can't create custom runtime images.

The second case involves modularizing only a part of our application and keeping the other part not modularized. This means we combine the module path with the class path: the part of the code that contains modules is on the module path, and the non-modularized part of the code is on the class path. A big problem is that by default, code from the module path can't access types from the class path! Fortunately, there are at least two solutions for this. One solution is to take the code from the class path and turn it into automatic modules. This is useful especially for third-party JAR files that may not yet have been modularized by their maintainers. Another solution would be to use a new command-line option called `--add-exports` that exports our packages so they can be accessed from other modules or from the class path.

The third case involves completely modularizing our application. As a result, the class path won't be used anymore. The entire code lies only on the module path. Each piece of code is included in a module defined by a module descriptor. No piece of code is residing outside of a module. This approach brings many advantages because we can use all the features that the Java Platform Module System offers, including strong encapsulation, reliable configuration, improved security, maintainability, reusability, scalability, and so on. We recommend following this approach and modularizing the entire application.

■ **Note** The class path wasn't removed in Java 9. It can still be used standalone or in combination with the module path.

There are three situations that we can have in Java 9 in correlation to the three use cases discussed earlier:

- *Only the class path is used*: The module path isn't used. Corresponds to the first use case mentioned earlier.
- *Both the class path and the module path are used*: Corresponds to the second use case mentioned earlier.
- *Only the module path is used*: The class path isn't used. Corresponds to the third use case mentioned earlier.

We'll start to learn some key concepts in order to be able later to migrate an application to Java 9. Many topics are covered in this chapter, because the migration topic is quite comprehensive. We introduce the key concepts that you need to know. Let's start by presenting the new concept of automatic modules, which are a very important component in the migration to Java 9 ecosystem.

Automatic Modules

Automatic modules are a special type of modules used to ease migration to Java 9 and to accomplish backward-compatibility. An automatic module is a named module created after placing a JAR file onto the module path. An automatic module isn't directly declared by the Java Platform Module System or by us—it's generated automatically for a JAR file that we place on the module path.

Automatic modules bring a great benefit in the modularization landscape. They permit us to start modularizing our own code without needing to wait for all the needed libraries and frameworks to be modularized. It would have been extremely bad to have to wait until each maintainer of each third-party library or framework modularizes their work.

An automatic module is created by deriving a JAR file, modularizing it without modifying its contents. In this way, each JAR file can be treated like a module. Automatic modules help us work with modules instead of working with non-modularized JAR files. They represent a bridge for each JAR file to the modular world.

An automatic module has at least five important characteristics:

- It requires transitive all the existing modules from the system, which comprise all our own modules plus all modules from the JDK image plus all the other automatic modules.
- It exports and opens all of its packages.
- It doesn't consist of a module-info.class file in its top-level directory.
- It can access every type from the unnamed module (from the class path).
- It can't declare that it has any dependencies to any other modules.

We stated previously that an automatic module exports and opens all of its packages. This means the following things:

- All the packages from an automatic module are exported for being accessible at both compile-time and at runtime.
- All the packages from an automatic module are open for being accessible using deep reflection.

■ **Note** An automatic module isn't explicitly declared by us because it's automatically created when a JAR file is placed onto the module path.

An automatic module can access types on the class path and is useful especially for third-party code. Automatic modules are used for migrating existing applications to Java 9. Let's suppose our application uses the Log4j library. If we put the Log4j JAR file on the module path, we can use it in our module by requiring it inside the module descriptor of our application:

```
module com.apress.myModule {
        requires log4j;
}
```

In this way, the Log4j JAR is turned into an automatic module and can be used in our modular application. We can access all the packages from the Log4j module because as an automatic module it exports all its packages by default.

We don't have to wait until the maintainers of the Log4j library have modularized their library because we can turn the Log4j library in an automatic module and use it on the module path (even if the Apache committers were hard-working and have already modularized Log4j at the time of writing this book).

When using an automatic module, the only thing we have to know is the name of the automatic module that will be generated. For this, Jigsaw makes use of a filename-based algorithm we cover shortly.

Don't worry if you run the preceding piece of code and see some warnings. The warnings have been deliberately added at runtime by the JDK team to make the users aware of the fact that they're using automatic modules.

An automatic module `requires transitive` all the existing modules. If we require an automatic module, then we acquire readability to all the modules, because the automatic module requires transitive all modules. Publishing a module that requires an automatic module on public repositories such as Maven Central is discouraged. That's because some of the properties of an automatic module, such as its exported packages, could change when it's later converted into an explicit module. This makes the automatic module unstable and increases the level of risk considerably.

The names of the automatic modules are generated automatically by the Java Platform Module System unless we set them explicitly in the MANIFEST.MF file. The name of the automatic module can be defined directly inside the MANIFEST.MF file from the META-INF directory of the JAR file. Inside MANIFEST.MF we need to set a value to the attribute `Automatic-Module-Name` in order to define the name of the automatic module that will be generated:

```
Automatic-Module-Name: myModule
```

This solution gives us the advantage and flexibility of being able to choose the name of the automatic module. Alternatively, if we don't set the automatic module name, Jigsaw will use an algorithm for deriving the name of the automatic module out of the name of the JAR, covered next.

Computing the Name of the Automatic Module

If the attribute `Automatic-Module-Name` isn't set, the name of the automatic module is automatically derived from the name of the JAR file. If the attribute `Automatic-Module-Name` is set, but the JAR also contains a module-info.class file, then the information stored in the attribute `Automatic-Module-Name` is simply ignored. The name of the automatic module will be the same as the one defined inside the module-info.class file.

Next let's talk about the filename-based algorithm used by Jigsaw for computing the name of the automatic module from the name of the JAR. Two strings are derived from the JAR file: the name of the automatic module and its version:

1. The .jar suffix is removed from the name of the JAR file. The resulting string is further used for determining and extracting the name and the version of the automatic module.
2. The module name is extracted. According to the JDK 9 API documentation, "if the name matches the regular expression -(\\d+(\\.|$)), then the module name will be derived from the subsequence preceding the hyphen of the first occurrence. The subsequence after the hyphen is parsed as a version and it is ignored if it can't be parsed as a version."
3. Some replacements on the name of the module are performed. The JDK 9 API documentation states that "all non-alphanumeric 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."

Table 8-1 shows some examples of deriving the name and the version from a couple of JAR files. The first column represents the name of the JAR file, and the second and third columns represent the automatically extracted name of the module and version, respectively. The fourth column tells us if an error occurred or not.

Table 8-1. *Examples of Deriving Module Names and Versions from JAR Files*

Name of JAR	Name of Module	Version of Module	Error
guava-19.0.jar	guava	19.0	no
hadoop-common-2.8.0.jar	hadoop.common	2.8.0	no
mockito-all-2.0.2-beta.jar	mockito.all	2.0.2-beta	no
spark-core_2.10-2.1.0.jar	-	-	yes
spring-core-4.3.7.RELEASE.jar	spring.core	4.3.7.RELEASE	no
com.apress.myModule0.0.1.jar	-	-	yes
log4j-1.2.17.redhat-2.jar	log4j	1.2.17.redhat-2	no
jackson-core-2.9.0.pr3.jar	jackson.core	2.9.0.pr3	no
jaxrs-api-3.0.12.Final.jar	jaxrs.api	3.0.12.Final	no
maven-plugin-api-3.5.0-beta-1.jar	maven.plugin.api	3.5.0-beta-1	no
123.jar	-	-	yes
1my-module.jar	-	-	yes

The name and version of the guava-19.0.jar JAR file could be successfully derived. According to the filename-based algorithm, first the jar suffix is deleted. The result string is `"guava-19.0"`. Afterward, the name of the module is extracted by searching for the first occurrence of the hyphen. The new string is extracted from the beginning of the resulting string until the last position before the hyphen. In our case, the string found is `"guava"`, which corresponds to the name of the module. The string after the hyphen represents the version of the module: `"19.0"`.

The name and version of the JAR hadoop-common-2.8.0.jar can be successfully extracted. The name of the module is hadoop.common because the hyphen from hadoop-common is replaced with a dot.

There are situations when we're not able to extract the name of the automatic module. An example is the JAR file called spark-core_2.10-2.1.0.jar. When attempting to extract its name, we get the following error:

```
Unable to derive module descriptor for: spark-core_2.10-2.1.0.jar
spark.core.2.10: Invalid module name: '2' isn't a Java identifier
```

The filename-based algorithm searches for the last hyphen in the string `"spark_core-2.10-2.1.0"` and splits the string into name and version. The resulting string for the name is `"spark_core-2.10"`. The hyphens on this string are replaced with dots. Therefore, the string `"spark.core.2.10"` is computed as the name of the module. However, this string is invalid because it contains the identifiers 2 and 10, which aren't valid as Java identifiers. As a result, an error is thrown, and the name of the automatic module can't be extracted. If we place this JAR on the module path, we get the following exception:

```
java.lang.module.ResolutionException: Unable to derive module descriptor for: spark-
core_2.10-2.10.jar
```

We get the same kind of error when we attempt to extract the module name from our com.apress.myModule0.0.1.jar, from our 123.jar, or from our 1my-module.jar:

```
Unable to derive module descriptor for: com.apress.myModule0.0.1.jar
com.apress.myModule0.0.1: Invalid module name: '0' isn't a Java identifier

Unable to derive module descriptor for: 123.jar
123: Invalid module name: '123' isn't a Java identifier

Unable to derive module descriptor for: 1my-module.jar
1my.module: Invalid module name: '1my' isn't a Java identifier
```

Attempting to place any of these three JARs on the module path will result in a `ResolutionException` being thrown.

■ **Note** A fatal error is thrown while placing a JAR on the module path for which the module name can't be extracted.

For the JAR commons-lang3-3.0.jar, the automatic module's name is commons.lang3. As we can observe, the digits are preserved at the end of the module name.

The JDK 9 specification recommends that the modules names follow the reverse Internet domain-name convention. According to the specification, "a module's name should correspond to the name of its principal exported API package, which should also follow that convention. If a module doesn't have such a package, or if for legacy reasons it must have a name that doesn't correspond to one of its exported packages, then its name should at least start with the reversed form of an Internet domain with which the author is associated."

Describing a JAR File

If we have a JAR that we want to use as an automatic module and want to find out what kind of name the JPMS system derives out of it, we can use the `--describe-module` option of the jar tool:

```
jar --describe-module --file <our_JAR_name>
```

The `--describe-module` option prints the following:

- The name of the module and the version
- The module descriptor
- The entire list of packages that the JAR consists of

Listing 8-1 displays the results of running the `jar` command with the `--describe-module` option on the guava.jar file.

Listing 8-1. Running jar --describe-module on the guava-19.0.jar File

```
$ jar --describe-module --file guava-19.0.jar
No module descriptor found. Derived automatic module.

guava@19.0 automatic
requires java.base mandated
```

```
contains com.google.common.annotations
contains com.google.common.base
contains com.google.common.base.internal
contains com.google.common.cache
contains com.google.common.collect
contains com.google.common.escape
contains com.google.common.eventbus
contains com.google.common.hash
contains com.google.common.html
contains com.google.common.io
contains com.google.common.math
contains com.google.common.net
contains com.google.common.primitives
contains com.google.common.reflect
contains com.google.common.util.concurrent
contains com.google.common.xml
contains com.google.thirdparty.publicsuffix
```

The module system finds no module descriptor inside the guava-19.0.jar, so it derives an automatic module out of the JAR file. The new automatic module has the name "guava" and the version "19.0". It requires java.base and consists of the packages listed above.

No Support for Automatic Modules at Link-time

There's a limitation regarding the use of automatic modules at link-time using Jlink. There's no support for automatic modules at link-time, which means that linking automatic modules into a runtime image is deliberately not supported. Automatic modules can't be used with Jlink because they have access to the class path. That means that if automatic modules were hypothetically supported by Jlink, errors of type `NoClassDefFoundError` would have been thrown at runtime.

■ **Note** It isn't possible to create a runtime with Jlink unless all the components are standard modules (not automatic modules).

The `ModuleDescriptor` class of the new module API contains a method called `isAutomatic()`. This method returns true if the module is an automatic one and false otherwise. We talk about the new module API and the API support for automatic modules in the next chapter.

■ **Note** Automatic modules open all their packages by default so we don't need to use the option `--add-opens` when working with automatic modules.

Now that we've covered almost everything we need to know about automatic modules, it's time to look at the JDeps tool, which is an extremely important tool used to find dependencies of a library.

The JDeps Tool

The Java Dependency Analysis Tool (JDeps) is a command-line tool used for different purposes: to discover all the static dependencies of a library, to discover the usages of internal JDK APIs, or to automatically generate a module descriptor for a JAR file. The tool was introduced in Java 8 but enhanced in Java 9 with some useful new options and features. It can be found in the bin directory of the JDK. JDeps is a very useful tool for migration to Java 9. We'll explain why.

Find Dependencies of Unsupported JDK Internal APIs

JDeps has an option called `--jdk-internals` that finds dependencies of any unsupported JDK internal APIs that are private to the JDK implementation. Its syntax is as follows:

```
jdeps --jdk-internals --class-path <input_file>
```

As an input, we can specify a JAR file or a .class file that will be analyzed.

Listing 8-2 shows an example of using JDeps with the option `--jdk-internals` by taking the Guava library and checking whether it has any unsupported APIs.

Listing 8-2. Running jdeps --jdk-internals on the JAR File guava-19.0.jar

```
$ jdeps --jdk-internals guava-19.0.jar
guava-19.0.jar -> jdk.unsupported
   com.google.common.cache.Striped64 -> sun.misc.Unsafe JDK internal API (jdk.unsupported)
   com.google.common.cache.Striped64$1 -> sun.misc.Unsafe JDK internal API (jdk.unsupported)
   com.google.common.cache.Striped64$Cell -> sun.misc.Unsafe JDK internal API (jdk.unsupported)
   com.google.common.primitives.UnsignedBytes$LexicographicalComparatorHolder$Unsafe
   Comparator -> sun.misc.Unsafe
JDK internal API (jdk.unsupported)
   com.google.common.primitives.UnsignedBytes$LexicographicalComparatorHolder$Unsafe
   Comparator$1 -> sun.misc.Unsafe
JDK internal API (jdk.unsupported)
   com.google.common.util.concurrent.AbstractFuture$UnsafeAtomicHelper -> sun.misc.Unsafe
JDK internal API (jdk.unsupported)
   com.google.common.util.concurrent.AbstractFuture$UnsafeAtomicHelper$1 -> sun.misc.Unsafe
JDK internal API (jdk.unsupported)

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

JDK Internal API                        Suggested Replacement
----------------                        ---------------------
sun.misc.Unsafe                         See http://openjdk.java.net/jeps/260
```

In the output we can see that JDeps finds all the JDK internal libraries that the Guava library is using. In our case, it finds the JDK internal class `sun.misc.Unsafe` in five different locations, listed in the preceding code.

JDeps also suggests replacements for the internal APIs found. For `sun.misc.Unsafe` it suggests taking a look on the Open JDK website at JEP 260. In general, JDeps is capable of giving clear information about possible replacements by suggesting the class name that could be eventually used instead.

In order to prepare for using Java 9, JDeps is very useful because we can check whether our JAR files from the class path are making use of JDK internal APIs. It isn't mandatory to replace the JDK internal APIs with the ones suggested by JDeps. We can replace them with whatever library we want. But we should replace them so that we can compile and run our application with JDK 9.

■ **Note** JDeps can be applied to modules as well.

Generate Module Descriptors with JDeps

JDeps can be used to generate a module descriptor for one or more JAR files using the command-line option `--generate-module-info`:

```
jdeps --generate-module-info <output_directory> <list_of_jar_files>
```

This command gets two parameters:

- `<output_directory>` represents the directory where the module-info.java files will be created.
- `<list_of_jar_files>` represents one or more JAR files for which a module-info will be generated. The list is separated by a blank space. For each JAR, each dependency must be listed here.

This command creates a module descriptor module-info.java for each JAR file that we pass. To demonstrate this, next we generate a module-info.java file for the junit-4.12.jar file. We also have to pass the hamcrest-core-1.3.jar file because this JAR is a dependency of JUnit:

```
jdeps --generate-module-info output hamcrest-core-1.3.jar junit-4.12.jar
```

As a result, two module-info files are created inside the output directory, one for JUnit and one for Hamcrest Core:

```
output\junit\module-info.java
output\hamcrest.core\module-info.java
```

Listing 8-3 shows an excerpt of the module-info.java files created.

Listing 8-3. Module Descriptors for JUnit and Hamcrest Core That Were Generated by JDeps

```
module junit {
   requires transitive hamcrest.core;
   requires java.management;
   exports junit.extensions;
   exports junit.framework;
   exports junit.runner;
```

```
    exports junit.textui;
    exports org.junit;
    exports org.junit.experimental;
    ...
    exports org.junit.runners;
    exports org.junit.runners.model;
    exports org.junit.runners.parameterized;
    exports org.junit.validator;
}

module hamcrest.core {
    exports org.hamcrest;
    exports org.hamcrest.core;
    exports org.hamcrest.internal;
}
```

■ **Note** The module descriptor generated by JDeps exports by default all the existing packages of its corresponding JAR.

JDeps can also generate a module-info.java file for an open module. The command is `generate-open-module` and it takes the same type of parameters:

```
jdeps --generate-open-module <output_director> <name_of_jar_file>
```

The only difference is that this command creates a module descriptor that defines an open module instead of a simple module. As a result, no packages are exported. This is suitable for frameworks that access the JDK using reflection.

JDeps also provides other useful features. Table 8-2 shows a list of the most useful options offered by JDeps, as defined in the JDK 9 API specification.

Table 8-2. *JDeps Options*

JDeps Option	Description
`--check <module_name> [,<module_name>...`	Prints the module descriptor and the resulting module dependences after analyzing the dependence of the specified modules
`--list-deps`	Lists the dependences of JDK internal APIs
`--class-path <path>`	Specifies the path where to find class files
`--module-path <module_path>`	Specifies the module path
`--upgrade-module-path <module_path>`	Specifies the upgrade module path
`--module <name_module>`	Specifies the root module that will get analyzed
`--multi-release <version>`	Specifies the version for processing multi-release JAR files
`-filter:module`	Filters dependences within the same module
`--regex <regex>`	Finds dependences matching the given pattern

We've looked at automatic modules and JDeps. It's time to focus on the Java 9 encapsulation topic. We'll learn how to break the encapsulation in Java 9, how to open packages and modules, and how to use the `--add-opens`, `--add-reads`, and `--add-modules` command-line options.

Encapsulation in Java 9

Java has two categories of APIs in the JDK: supported APIs and unsupported APIs. The supported APIs comprise JCP standard APIs like java.* and javax.*, JDK-specific APIs like com.sun.* and jdk.*. These APIs are intended to be used outside the JDK.

The unsupported APIs comprise the sun.* packages. These APIs were never intended for external use outside the JDK. Typically all the packages that contain the name `"internal"` are JDK internal APIs. The problem is that in the past many developers used the sun.* packages, even if they were told they weren't allowed to use these packages outside the JDK.

Java 9 encapsulated almost all the JDK internal APIs, which means that by default, without any hacks, these APIs aren't accessible either at compile-time nor at runtime.

■ **Note** Oracle made a study that revealed the most-used JDK internal classes: `sun.misc.BASE64Encoder`, `sun.misc.BASE64Decoder`, and `sun.misc.Unsafe`.

The JCP team put the JDK-internal APIs into two categories: non-critical JDK-internal APIs and critical JDK-internal APIs.

The non-critical JDK-internal APIs category comprises the APIs that are used outside the JDK to an extremely low degree. Therefore, the risk of breaking applications by encapsulating these APIs is also low. This category of APIs also contains the `sun.misc.BASE64Encoder` and `sun.misc.BASE64Decoder` classes.

The critical JDK-internal APIs category comprises the APIs whose functionality would be extremely difficult to implement outside the JDK. It's demanding, if not almost impossible, to develop replacements for these APIs outside the JDK. This category contains, for instance, the `sun.misc.Unsafe` class, which was marked as critical because it's very demanding to build a similar class outside the JDK.

Therefore, the JCP team decided to do the following:

- Encapsulate all non-critical internal APIs
- Encapsulate all critical internal APIs for which supported replacements exist in JDK 8
- Not encapsulate critical internal APIs, but just deprecate them

The critical internal APIs that weren't encapsulated are `sun.misc.Unsafe, sun.misc.Signal, sun.misc.SignalHandler, sun.misc.Cleaner, sun.reflect.Reflection, sun.reflect.ReflectionFactory`. These are still accessible in JDK 9.

The following example from Listing 8-4 demonstrates the encapsulation of JDK internal APIs. Therefore, we use an instance of the class `URLCanonicalizer` from the package sun.net. All the classes from package sun.net were encapsulated in JDK 9.

Listing 8-4. Use of a Class from an Internal JDK API

```
package com.apress.jdkinternal;

import sun.net.URLCanonicalizer;

public class Main {
```

```
        public static void main(String[] args) {
                URLCanonicalizer urlCanonicalizer = new URLCanonicalizer();
                String apressUrl =  urlCanonicalizer.canonicalize("www.apress.com");
                System.out.println(apressUrl);
        }
}
```

The compilation fails because we try to access a JDK internal API that is encapsulated:

```
error: package sun.net isn't visible
import sun.net.URLCanonicalizer;
  (package sun.net is declared in module java.base, which doesn't export it to module
com.apress.jdkinternal)
```

The error states that the package sun.net, located in module java.base, isn't visible from our module com.apress.jdkinternal. We know that the sun.net package has been encapsulated, so we need a way to make our module com.apress.jdkinternal able to access the sun.net package at compile-time.

Fortunately, there's a solution to gain access to the sun.net package—by exporting this package to our module during compilation using the `--add-exports` command-line option, described next.

Exporting a Package at Compile-time and Runtime

The `--add-exports` option added to the Java compiler (javac) exports a package to a specific named module or to the unnamed module. It corresponds to the qualified export "`exports … to`" statement from the module declaration. It can be used to break the encapsulation of JDK internal APIs and to make them accessible in a named module or in the unnamed module.

The following shows the syntax of the `--add-exports` command-line option:

```
--add-exports <source_module>/<name_of_package_to_be_exported>=<list_of_target_modules>
```

- `<source_module>` represents the module where the package to be exported is located.
- `<name_of_package_to_be_exported>` represents the name of the package that will be exported to the `<list_of_target_modules>`.
- `<list_of_target_modules>` represents a comma-separated list of modules that will gain access to the exported package.

In Figure 8-1 the package sun.net, located in module java.base, is exported to our module com.apress.jdkinternal.

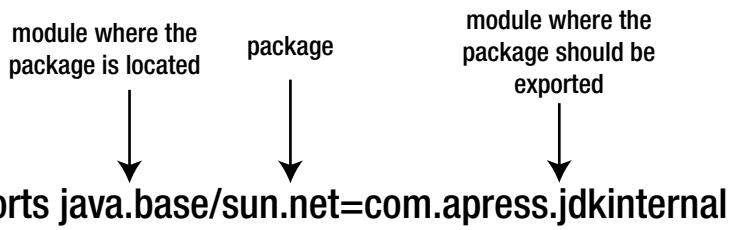

Figure 8-1. *Exporting the sun.net package to a named module with the --add-exports option*

In this way, the package sun.net will be accessible from our the module com.apress.jdkinternal. By compiling our application again using the option `--add-exports` mentioned earlier, the package sun.net will be exported to our module. We pass the module where the package is located (java.base) and the module where the package should be exported (com.apress.jdkinternal).

To compile we have to use the `--add-exports` option, as mentioned:

```
$  javac –d outputDir --add-exports java.base/sun.net=com.apress.jdkinternal
--module-source-path src $(find . –name "*.java")
```

The compilation succeeds, and the .class files are created. However, a warning is displayed, informing us that URLCanonicalizer is an internal proprietary API that may be removed in a future release:

```
warning: URLCanonicalizer is internal proprietary API and may be removed in a future release
import sun.net.URLCanonicalizer;
```

We run the application by using exactly the same `--add-exports` command:

```
java --module-path outputDir --add-exports java.base/sun.net=com.javausergroup.jdkinternal
-m com.apress.jdkinternal/com.apress.jdkinternal.Main
```

Because we need readability at runtime, not only at compile-time, it's mandatory to use the same `--add-exports` option with the same arguments when running the application. If we had run our application without the `--add-exports` flag, the following error would have been thrown:

```
Exception in thread "main" java.lang.IllegalAccessError: class com.apress.jdkinternal.Main
(in module com.apress.jdkinternal) can't access class sun.net.URLCanonicalizer (in module
java.base) because module java.base doesn't export sun.net to module com.apress.jdkinternal
        at com.apress.jdkinternal/com.apress.jdkinternal.Main.main
```

The `IllegalAccessError` occurs at runtime because the package sun.net isn't exported.

Export to the Unnamed Module

We exported an unsupported package to our module to make it accessible. But what if our code was on the class path? Fortunately, there's a solution for that. The constant `ALL-UNNAMED` stands for the entire class path. In our case, the following command exports the sun.net package to the class path so it can be accessed from the entire code on the class path:

```
--add-exports java.base/sun.net=ALL-UNNAMED
```

■ **Note** The constant `ALL-UNNAMED` stands for all the code in the unnamed module, which represents the entire class path.

There are some general aspects that want to mention:

- The `--add-exports` command-line option can be used more than once when running it, meaning it allows duplicates.
- The `--add-exports` command-line option is used by both Java compiler and Java launcher.

- If our code uses only the JDK-critical APIs that remained accessible in Java 9, we don't have to use the `--add-exports` option because these APIs are already accessible.
- If the `--add-exports` command-line option encounters bad values, a warning is raised, but no fatal error is thrown, so the program doesn't stop working.

Throughout this book we've looked at two ways of exporting a package: by specifying the `exports` clause in the module declaration or by using the command-line option `--add-exports`. But there's still another option: specifying the attribute `Add-Exports` in the MANIFEST.MF file of a JAR file. This attribute has the format `module/package`. It exports the specified module from the specified package to the unnamed module. For instance, in order to export the package sun.net from module java.base to the unnamed module, we could write the following:

```
Add-Exports: java.base/sun.net
```

In this section, we've learned to make code that uses encapsulated JDK APIs compile and run in JDK 9. This workaround is very useful because if our code uses JDK internal APIs, we know that we have one solution to make the code run in Java 9 without needing to redesign the code or replace the encapsulated JDK internal APIs. But relying on this flag forever isn't recommended because the JDK internal APIs were deprecated in JDK 9 and may be removed in JDK 10. That means you have time only during a release cycle to refactor your code in order to get rid of these unsupported APIs. If your third-party library is using JDK internal APIs, you should check on a regularly basis if a new version of the library, one that replaces the unsupported APIs with supported ones, has been published.

Opening Packages for Deep Reflection

Chapter 4 talked about the `opens` clause in the module descriptors. There we mentioned that deep reflection is by default allowed by code in a named module to code on the class path, but by default it's not allowed to code in another named module. In this second case, in order to allow reflective access from code in a named module to code in another named module, we could use the new `--add-opens` command-line option. It's used to provide deep reflective access from one module to another module or to the code on the class path. It's equivalent to a qualified `opens` from a module declaration:

```
opens <package_name> to <list_of_target_modules>
```

The syntax of the add-opens command-line option goes like this:

```
--add-opens <source_module>/<name_of_package_to_be_opened>=<list_of_target_modules>
```

The package defined and located in the `<source_module>` is opened for deep reflective access to the modules listed in the `<list_of_target_modules>`. These modules will be able to access the package at runtime only using deep reflection but won't be able to access the package during compilation. If we put the constant `ALL-UNNAMED` instead of the `<list_of_target_modules>`, then the entire code on the class path will be able to access the package at runtime using deep reflection. However, this last case happens already by default, so we should use it only if someone programmatically disabled reflective access by code in a named module to code from the class path.

■ **Note** Deep reflection can take place only at runtime. It can't take place at compile-time. As a result, the `--add-opens` command-line option can be used only at runtime using the `java` command. It can't be used at compile-time using the `javac` command.

Because automatic modules open all their packages by default, there's no need to open them using the `--add-opens` option. Now that we know how to open packages at runtime for deep reflection, let's learn how to add readability at runtime using the option `--add-reads`.

Providing Readability Between Modules

The `--add-reads` command-line option is used at both compile-time and runtime to add readability from a module to another module. Its syntax is as follows:

```
--add-reads <source_module>=<list_of_target_modules>
```

By using the `--add-reads` command-line option, the `<source_module>` gets readability to all the modules represented by the `<list_of_target_modules>`, which means that the `<source_module>` will require all those modules. This is equivalent to providing a `requires` clause in a module descriptor:

```
module <source_module> {
        requires target_module_A;
        requires target_module_B;
}
```

We can make a module `<source_module>` read the entire class path by providing the constant `ALL-UNNAMED` to the `--add-reads` option:

```
--add-reads <source_module>=ALL-UNNAMED
```

This option is used merely during testing—for instance, when a module is patched at compile-time and at runtime in order to add tests in the same modules as the module under test. During testing, we might need a module to read another module, although the first module doesn't depend on the other module because it doesn't define a `requires` directive to the other module. By using the `--add-reads` option, we get readability between the two modules. The first module will be able access all the exported types from the other module.

Suppose we have a Junit test class inside our module com.apress.testing. This class extends a class from the Junit library. So we need a readability relation from our module com.apress.testing to the automatic module junit. This can be achieved very simply using the `--add-reads` option:

```
--add-reads com.apress.testing=junit
```

If we have the Junit library on the class path and don't want to move to the module path, then we use the `ALL-UNNAMED` constant to provide readability between our module and the entire code on the class path:

```
--add-reads com.apress.testing=ALL-UNNAMED
```

Chapter 11 provides an explanatory example of using the `--add-reads` command-line option for running JUnit tests.

■ **Note** The `--add-reads` command allows duplicates and if it encounters bad values, a warning is raised, but no fatal error is thrown. If duplicates are found, only the first class will be taken into consideration.

Before we discuss the `--add-modules` command-line option, there's one more thing worth mentioning: when we use reflection on a member in a module, readability is automatically granted.

Adding Modules to the Root Set

The `--add-modules` command-line option is used to add modules directly to the set of root modules. Hence, the modules will be resolved. This option is used to resolve modules that aren't resolved by default.

Its syntax is simple. It takes one or more modules separated by comma:

```
--add-modules <module_name>(,<module_name)*
```

`<module_name>` represents the name of a module that will be added to the default set of root modules

There are three values that can be used with the `--add-modules` option instead of specifying a list of modules:

- `ALL-DEFAULT`: The official specification released for JDK 9 states that by using the `ALL-DEFAULT` option "the default set of root modules for the unnamed module, as defined above, is added to the root set. This is useful when the application is a container that hosts other applications which can, in turn, depend upon modules not required by the container itself."
- `ALL-SYSTEM`: This option adds all the system modules to the root set.
- `ALL-MODULE-PATH`: This option adds all the observable modules found on the module path to the root set. It's helpful to be able to add each module from the module path at once to the root set. For a large list of automatic modules, it's more practical and easier to add all of them at once and without the need to enumerate them one by one. Maven uses this option considerably because it needs all the modules from the module path.

The `--add-modules` option is also used by Jlink to set the root module inside the runtime image. We saw in Chapter 7 how to create a runtime image and how to add modules to the runtime image using the `--add-modules` command-line option.

■ **Note** Both `javac` and `java` support the `--add-modules` command-line option.

The `--add-modules` option can be repeated. The following usages of `--add-modules` options have the same effect and cause no errors:

```
--add-modules com.apress.moduleA --add-modules com.apress.moduleB
--add-modules com.apress.moduleA,com.apress.moduleB
```

Next we'll look at an explanatory example to better understand when we should use the `--add-modules` option. We download the junit-4-12.jar and the hamcrest-core-1.3.jar and put them into a folder. We run `jdeps -s` on the entire folder to find all the dependencies of the two JAR files:

```
 $ jdeps -s *.jar
hamcrest-core-1.3.jar -> java.base
junit-4.12.jar -> hamcrest-core-1.3.jar
junit-4.12.jar -> java.base
junit-4.12.jar -> java.management
```

Hamcrest-Core depends only on java.base, meaning it use types only from java.base. Hence, Junit is depending on Hamcrest-Core because it uses types from it. If we look inside the content of Junit, we can see lots of imports from Hamcrest-Core packages.

Suppose we have a module com.apress.myModule and we put the junit-4.1.2.jar and the hamcrest-core-1.3.jar files on the module path in order to use them as automatic modules. You already learned at the beginning of this chapter, in the "Automatic Modules" section, that an automatic module can't declare dependencies on other modules. So we can't use the directive `requires hamcrest-core` because we don't have a module descriptor available where to place it, because an automatic module doesn't have a module descriptor module-info.java. The situation is depicted in Figure 8-2.

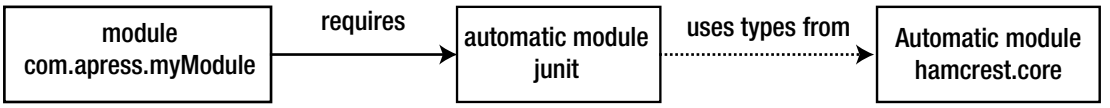

Figure 8-2. *Relation between named modules and automatic modules that have dependencies*

Module com.apress.myModule contains in its module descriptor a `requires junit` clause. The automatic module junit uses types from the automatic module hamcrest.core. The module graph contains the modules com.apress.myModule and junit. The module hamcrest.core isn't added to the module graph because it can't be identified during the resolution process. There's no `requires` clause inside the junit automatic module so that the module system could discover the hamcrest.core automatic module and add it to the module graph. This means we have to manually add the hamcrest.core automatic module to the module graph using the `--add-modules` option at both compile-time and runtime:

```
--add-modules hamcrest.core
```

If we don't add the hamcrest.core module to the module graph, the classes from the hamcrest.core won't be found and an exception of type `ClassNotFoundException` will be thrown at runtime.

Another option provided by default in JDK is the `--illegal-access` one, covered in the next section.

The --illegal-access Option

The `--illegal-access` option was added into JDK 9 to ease migration. This option states that code on the class path can perform illegal reflective access by default.

■ **Note** *Illegal reflective access* means access using reflection to types in named modules only for code in the class path.

With the `--illegal-access` option, the code on the class path gets reflective access to types in any named modules. The reflective access is done using standard reflection-related APIs like java.lang.reflect and java.lang.invoke. `--illegal-access` is very useful for third-party frameworks like Spring, Hibernate, or Guava, which were so designed that they need to perform reflective access inside the internals of the JDK in order to be able to work properly.

■ **Note** The `--illegal-access` option allows reflective access only for code on the class path to types in any named modules. It doesn't allow reflective access for code in named modules to types in other named modules.

The syntax of the `--illegal-access` option is as follows:

```
java --illegal-access <options>
```

The `--illegal-access` option can take one of four possible parameters: permit, warn, debug, and deny:

- `--illegal-access=permit`: The `permit` mode represents the default behavior in Java 9. It states that every package from every module is opened for deep reflection to code in all the unnamed modules. The unnamed modules represent the class path. This means that at runtime the code from the class path can access the entire information stored in modules using deep reflection. A warning will be displayed during the first access.
- `--illegal-access=warn`: The `warn` mode is very similar to the `permit` mode previously discussed. The only difference is that the `warn` mode returns a warning each time an illegal access is performed using reflection.
- `--illegal-access=debug`: The `debug` mode shows a warning in the stack trace for every illegal access performed using reflection.
- `--illegal-access=deny`: The `deny` mode disables all the illegal access operations using reflection. When this mode is set, no illegal access can be done using reflection. Hence, this mode can be overwritten by the command-line option `--add-opens`. With `--add-opens` we're able to open specific packages for reflection.

■ **Note** The JDK internal APIs aren't encapsulated at runtime.

However, the JCP team announced that the `--illegal-access` option will be removed in JDK 10. It will be available only in JDK 9 in order to ease migration of third-party libraries that were constructed by making use of deep reflective access into the internals of the JDK. The `--illegal-access` option has not been planned right from the beginning. It was added later in order to ease migration to Java 9, because an important number of external libraries and frameworks use reflection to access the internal APIs of the JDK.

This option emits warning messages when used:

```
WARNING: Illegal access by A to B (permitted by C)
```

- A is the name of the type that contains the code that invoked the reflective operation in question.
- B is the name of the member being accessed.
- C is the name of the command-line option that enabled this access.

■ **Note** The `--illegal-access` option is set by default in JDK 9.

The next example shows what kind of warning messages are displayed when a library performs illegal reflective access to types in the JDK. We run the `java -jar` command on the the JRuby Complete-9.1 JAR file:

```
$ java -jar jruby-complete-9.1.12.0.jar
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.jruby.util.io.FilenoUtil (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to method sun.nio.ch.SelChImpl.getFD()
WARNING: Please consider reporting this to the maintainers of org.jruby.util.io.FilenoUtil
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access
operations
WARNING: All illegal access operations will be denied in a future release
```

A warning message lets us know that the jruby-complete-9.1.12.0.jar performs an illegal reflective access operation on method `sun.nio.ch.SelChImpl.getFD()` of the JDK. This illegal reflective access is allowed, because the `--illegal-access` flag is set by default.

If we want to see a warning for every illegal access performed, we can use the `mode=warn` of the `--illegal-access` command-line option:

```
$ java --illegal-access=warn -jar jruby-complete-9.1.12.0.jar

WARNING: Illegal reflective access by org.jruby.util.io.FilenoUtil (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to method sun.nio.ch.SelChImpl.getFD()
WARNING: Illegal reflective access by org.jruby.util.io.FilenoUtil (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to field sun.nio.ch.FileChannelImpl.fd
WARNING: Illegal reflective access by org.jruby.util.io.FilenoUtil (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to field java.io.FileDescriptor.fd
WARNING: Illegal reflective access by jnr.posix.JavaLibCHelper (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to method sun.nio.ch.SelChImpl.getFD()
WARNING: Illegal reflective access by jnr.posix.JavaLibCHelper (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to field sun.nio.ch.FileChannelImpl.fd
WARNING: Illegal reflective access by jnr.posix.JavaLibCHelper (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to field java.io.FileDescriptor.fd
WARNING: Illegal reflective access by jnr.posix.JavaLibCHelper (file:/C:/Users/Alex/
Downloads/jruby-complete-9.1.12.0.jar) to field java.io.FileDescriptor.handle
WARNING: Illegal reflective access by org.jruby.java.invokers.RubyToJavaInvoker (file:/C:/
Users/Alex/Downloads/jruby-complete-9.1.12.0.jar) to method java.lang.Object.clone()
WARNING: Illegal reflective access by org.jruby.java.invokers.RubyToJavaInvoker (file:/C:/
Users/Alex/Downloads/jruby-complete-9.1.12.0.jar) to method java.lang.Object.finalize()
...
```

The output is very long, so we decided not to include it all. You can see that the type that performs the illegal reflective access is displayed together with the name of the method or the field from the JDK that's accessed reflectively.

The JDK 9 API introduced a new useful method in the `AccessibleObject` class of package java.lang.reflect called `boolean canAccess(Object object)`. This method lets us test whether the caller can access this reflected object. The method returns true if access is allowed and false otherwise. According to the JDK 9 API documentation, an `IllegalArgumentException` will be thrown if "this reflected object is a static member or constructor or if it is an instance method or field and the given object is null." To avoid any exceptions, we could also use the new `boolean trySetAccessible()` method from the same class. This method doesn't throw any exceptions except a `SecurityException` if the request is denied by the Security Manager.

■ **Note** The system property `sun.reflect.debugModuleAccessChecks=access` allows us to get a stack trace on each warning. It also can help to debug exceptions raised by the use of `--illegal-access`.

We talked about the command-line flags. Now it's time to present some migration issues that can commonly occur.

Migration Issues

This section explains the concepts and gives practical solutions to some of the most common issues that usually occur during the migration process to Java 9:

- Encapsulated JDK internal APIs
- Not resolved modules
- Cyclic dependencies
- New versioning scheme
- Split packages
- Removed methods in Java 9
- Removal of rt.jar, tools.jar, and dt.jar

Encapsulated JDK Internal APIs

Throughout the book we've talked about the encapsulation of the JDK internals APIs. This can cause critical problems when we move to JDK 9. However, this probably isn't the most encountered problem during migration to JDK 9. In our opinion, the split packages and cyclic dependencies problems can occur more often.

Two independent solutions can help solve the problem of JDK internal APIs:

- Replace each of your JDK internal APIs with supported APIs.
- Keep the existing JDK internal APIs and use the `--add-exports` command-line option to break the encapsulation—to make the JDK internal APIs accessible to code in other modules or to code on the class path.

The first solution is by far the better one because we completely get rid of the unsupported JDK APIs in our code. Because the JDK internal APIs are marked as deprecated, it's wise to provide replacements for them as soon as possible.

The second solution is reasonable if you don't have enough time to replace the unsupported JDK internal APIs with supported ones. If all you want is to make your code compile and run again using JDK 9 this time, using the `--add-exports` option is a way to move forward. However, Oracle has stated that the JDK unsupported APIs will be removed in the next major JDK release. This could be JDK 10 or later. Sooner or later, you'll have to replace them with supported JDK APIs in order to ensure that your code won't break. As a conclusion, adding the `--add-exports` option is just a temporary solution to make code work. There's no guarantee how long this workaround will work. It all depends on how long the unsupported JDK APIs that you're using will remain in JDK and not be removed.

We already saw in this chapter how to identify the presence of JDK internal APIs in a JAR file or in a module: by using the JDeps tool with the option `--jdk-internals`. Let's move on to discuss another possible problem we may encounter during compilation: not resolved modules.

Not Resolved Modules

Remember the module graph that resulted after the modularization of the JDK ? Figure 8-3 shows a small part of it.

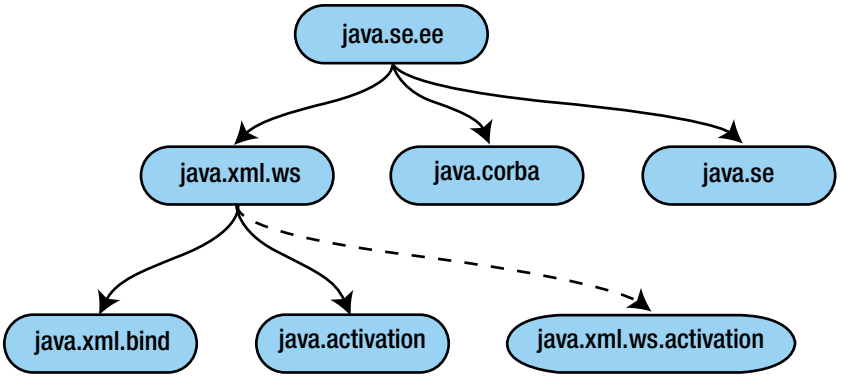

Figure 8-3. *Small part of the module graph of the Java SE modules with module java.se.ee at the top*

Module java.se.ee is located right at the top of the module graph, and module java.se is only a level below. As described in Chapter 3, the differences between the module java.se.ee and the module java.se are as follows:

- The module java.se.ee collects all modules that comprise the Java SE platform, including the modules that overlap with the Java EE platform.
- The module java.se collects all the modules that comprise the Java SE platform that don't overlap with the Java EE platform.
- The module java.se.ee contains a total number of five modules that aren't present in module java.se: java.xml.ws, java.xml.bind, java.corba, java.activation, and java.xml.ws.annotation.

We deliberately use the term *collects* instead of *contains* here because both module java.se.ee and module java.se are aggregator modules, which means that according to the JDK 9 specification, they "collect and re-export the content of other modules but add no content of their own."

During the compilation process in JDK 9, the java.se module is considered the root module and not the java.se.ee module. This means that in the compilation step, the visible modules are the ones that are under the java.se module. It also means that the five modules from the module java.se.ee that aren't in module java.se aren't visible at compilation.

■ **Note** The reason why the five modules aren't resolved by default is related to backward-compatibility problems.

If our code makes use of any of the following five modules, the compilation will fail:

- java.xml.ws
- java.xml.ws.annotation
- java.xml.bind
- java.corba
- java.activation

We can compare the modules contained by the java.se module with the modules contained by the java.se.ee module by limiting the observable modules with the `--limit-modules` option. The following two commands return the name of all the modules having module java.se and java.se.ee, respectively, in the root of the transitive closure:

```
java --limit-modules java.se --list-modules
java --limit-modules java.se.ee --list-modules
```

The solution for getting rid of the non-resolved modules problem is simple. We have to add the modules to the default root set of modules at both compile-time and runtime using the `--add-modules` command-line option, so they can be resolved:

```
--add-modules <module_name>
```

For instance, if we're using types from module java.xml.ws, then it's absolutely necessary to always add the module java.xml.ws in the root set of modules at both compile-time and runtime:

```
javac --add-modules java.xml.ws
java --add-modules java.xml.ws
```

As a result, the java.xml.ws module is resolved and can be used.

■ **Note** Even if we use only libraries that have dependencies on these five modules, we still have to add the non-resolved modules to the root set of modules.

We now know that by adding the modules to the root set of module we can solve compilation errors like `"package java.activation doesn't exist"` or `"package java.xml.bind doesn't exist"`.

It's time to explore another issue that can occur during migration: split packages.

Split Packages

The split packages problem is one of the most serious problems that can take place in the Java 9 modular world. Split packages occur when two or more members of a package reside in more than one module. In order to support reliable configuration, the Java Platform Module System doesn't allow split packages at compile-time. The reason is that the system loads all the modules from the module path with a single class loader, which can't have more than one single type of a package. Two modules loaded by the same class loader can't split a package.

Figure 8-4 illustrates two modules having a split package.

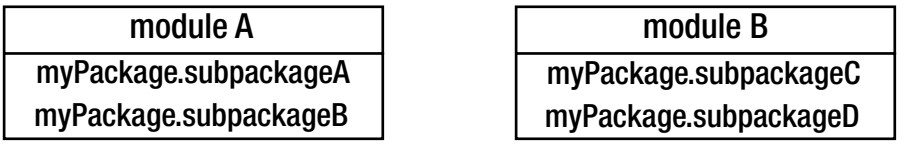

Figure 8-4. *Two modules having a split package*

Module A and module B contain both the package myPackage. Even if the modules contain different subpackages, the split package problem is present, because they share a package with the same name. The split package arises even if the packages aren't exported.

In this case, the compilation will fail with the following error because we have split packages at compile-time:

```
error: module A reads package myPackage from both A and B
```

This error clearly states that the package B is in both module A and module C.

A split package problem can occur for every type of package, even for packages that aren't exported, the so-called *concealed* packages. If two modules contain a package with the same name, an error will occur when we put the modules on the module path. It doesn't matter if the packages are exported, open, or concealed. The split package problem will occur anyway.

■ **Note** If a package is neither exported nor open, we can say that the module conceals the package.

It's also important to mention another aspect. If we develop our own module that uses a package name that already exists in one of the platform modules, we have the split packages problem too.

■ **Note** The packages from the platform modules also count in the split package problem. This means we can't use in our own module a package that has the same name as a package that resides in the existing platform modules.

There's no universal solution to repair the split packages problem. You can choose whatever solution you want in order to reach the desired goal—to not have a package or members of a package with the same name in more than one module.

Suppose we have two third-party JAR files that share a package with the same name. Some of the most used solutions to fix the split packages problems include the following:

- Create a single JAR file out of the two JAR files. Combine them into a single JAR file. If we have two third-party JAR files that share a package with the same name, then we could make a single JAR file out of the two by unzipping them in the same directory and then zipping the entire directory into a single ZIP file. Don't forget to change the suffix of the new ZIP file into a JAR file. In this way, we have just one single JAR that can be put into the module path and have just one automatic module, not two. The package is now in a single module, and the split packages issue is gone.
- Check to see whether one of the JARs can be eventually replaced by a different one. If there's a chance of replacing the JAR with another one, we should at least try it.
- Rename one of the packages. Renaming one of the packages is also a solution to consider. The probability of success depends on the structure of the classes and especially if the classes live in a single namespace or not.

Until now we've talked about different use cases for JAR files. Let's move on to modules. There are three possible solutions that can help getting rid of split packages in the case of modules:

- *Create a single module out of two or more modules*: Combine them into a single module. If we have two modules that share a package with the same name, then we could eventually redesign our code and have just one module out of the two.
- *Create a third module*: Another option would be to take the entire packages that cause the split package problem from both modules and move them into a third new module, which exports the packages we need inside our module. This solution is much easier to implement.
- *Attempt to remove the package dependencies*: This is questionable and can be implemented only if you really don't need the dependency anymore.

You've seen some suggestions of how to solve the split packages problem. You can choose these approaches or implement your own solutions in order to reach the goal of not having a package with the same name in more than one module.

■ **Note** Split package problems can also occur in JAR files that are making use of the Service Provider API.

The JEP 200 states that "a non-standard module must not export any standard API packages." This makes sense because if we have our own module com.apress.myModule, we shouldn't, for example, export the java.sql package, because the java.sql package is already exported by the java.sql platform module. This will result in a split package.

■ **Note** Don't export any standard API from a non-standard module. Otherwise, you will have a split package.

One requirement of the JPMS states that "the Java compiler, virtual machine, and runtime system must ensure that modules that contain packages of the same name don't interfere with each other. If two distinct modules contain packages of the same name then, from the perspective of each module, all of the types and members in that package are defined only by that module. Code in that package in one module must not be able to access package-private types or members in that package in the other module."

■ **Note** When we develop unit test cases in JDK 9, we must be careful not to introduce split packages. If we have a specific test module where we put the test cases, then when we import types from the module under test in the test module, we introduce the split package issue because we'll have the same package in two different modules.

The next section talks about another problem that can arise: cyclic dependencies.

Cyclic Dependencies

A *cyclic dependency* is a relation between two or more modules expressed by the fact that the modules depend on each other, either directly or indirectly. Cyclic dependencies are considered *anti-patterns*. They aren't allowed at compile-time in Java 9. If two modules contain a cyclic dependency, the compilation will fail. Jigsaw deliberately imposes a cyclic dependency check during compilation. The requirement imposed by the Java Platform Module System is severe: no cycle dependencies are allowed in the module graph.

However, cyclic dependencies are allowed at runtime, but only after the module graph is already resolved. We refer here to the `reads` relations of runtime modules, which are allowed at runtime. Cyclic dependencies aren't allowed at compile-time, link-time, and runtime when the module graph is resolved for the first time. But at runtime, you can add readability edges using the command-line option `--add-reads`. At runtime, you can introduce a cyclic dependency using the `--add-reads` option because the module graph has already been resolved before and because we're at runtime, not at compile-time.

The reasons for interdicting cyclic dependencies are justified: to simplify the module system or to make the module graph more understandable. Two modules that require each other would be better represented as a single module.

Cyclic dependencies can occur often for automatic modules, for instance. Because automatic modules imply readability to all other modules, the likelihood of getting two modules that depend on each other isn't low.

■ **Note** Cyclic dependencies between modules are forbidden during compilation. Cyclic dependencies between classes are allowed just inside of a single class, not between distinct modules.

Chapter 4 had an example of cyclic dependency in our module declaration. Cyclic dependencies can be solved by using interfaces to decouple the coupling between modules. A module should depend on an interface, not on another module. This can be implemented using the Service Provider API described in Chapter 6. What we need to do is to implement Service consumers and Service providers to decouple the coupling between modules.

■ **Note** There's an official proposal to allow cyclic relationships amongst modules at runtime, but not at compile-time. It's unclear when this proposal will be implemented—possible in JDK 10. Allowing cyclic relationships at runtime will help solve some problems that can arise especially for very large applications, where the probability of having cycles is much higher.

We covered the cyclic dependencies issue in this section. The next section covers the new versioning scheme introduced in JDK 9.

New Versioning Scheme

Java 9 introduces a new format to define the version. This matters for a migration point of view because code that relies on the old string format will break. The maintainers of the Hadoop library had to fix the Hadoop library because it was broken in JDK 9 due to the introduction of the new version format:

```
System.getProperty("java.version").substring(0, 3).compareTo("1.7") >= 0
```

This piece of code isn't working on JDK 9 anymore, because the version is no longer represented as 1.7. Instead of 1.7, the new version could have a format similar to 7 (containing only the major version) or 7.1.1 (containing the major version, minor version, and security version).

The format of the new version string is as follows:

```
$MAJOR.$MINOR.$SECURITY.$PATCH
```

- `$MAJOR` serves as the major version of a JDK release.
- `$MINOR` serves as the minor version of a JDK release.
- `$SECURITY` serves as a security-related release of the JDK.

The changes affect not only `java -version`, but also the following system properties: `java.runtime.version`, `java.vm.version`, `java.specification.version`, and `java.vm.specification.version`.

We know how the new version looks in JDK 9, so let's move on and see what methods were removed in JDK 9.

Removed Methods in JDK 9

The following methods have been completely removed in Java 9:

- `java.util.logging.LogManager.addPropertyChangeListener`
- `java.util.logging.LogManager.removePropertyChangeListener`
- `java.util.jar.Pack200.Packer.addPropertyChangeListener`
- `java.util.jar.Pack200.Packer.removePropertyChangeListener`
- `java.util.jar.Pack200.Unpacker.addPropertyChangeListener`
- `java.util.jar.Pack200.Unpacker.removePropertyChangeListener`

We should make sure not to use these six methods in Java 9—otherwise our code will break at compile-time. The probability of having one at least one of these six methods in our code is low.

Another change in JDK 9, with a definitely greater impact, is the removal of the runtime rt.jar and of tools.jar and dt.jar.

Removal of rt.jar, tools.jar, and dt.jar

Chapter 2 talked about the removal of the rt.jar, tools.jar, and dt.jar in JDK 9. This can have a consequence on our code if we make assumptions throughout our code based on one of these three JAR files. But the impact is greater on tools rather than on our own code.

Calling the `ClassLoader::getSystemResource()` method in JDK 9 won't return an URL to a JAR file. Instead, it will return a valid URL.

If we call the method `getSystemResource()` with the parameter `java/lang/Class.class`,

```
ClassLoader.getSystemResource("java/lang/Class.class")
```

The following URL will be returned:

```
jrt:/java.base/java/lang/Class.class
```

We have to be aware of these new changes and check if our code expects to receive this URL in a specific format, which may now not be the same.

Let's move to the next section, where we show migration strategies for migrating a Java application to Java 9.

Migrating an Application to Java 9

This chapter describes the process of migrating an application to Java 9 using the top-down approach. There are basically two types of migration we can perform when we decide to migrate our existing Java application, together with its dependencies, to Java 9: top-down migration and bottom-up migration.

The main difference between the two approaches is that application migration migrates the application first. By contrast, library migration starts migrating the libraries first, rather than the application.

■ **Note** In Chapter 4 you learned what an unnamed module is. It is important to remember the following rule: code that exists in a named module can't access anything on the class path!

Top-down Migration

We have a small application that reads some news from a JSON file using Google Gson. It logs the output using SLF4J and formats it using Google Guava. Therefore, we have four JAR libraries on the class path:

- slf4j-simple-1.7.25.jar
- slf4j-api-1.7.25.jar
- guava-21.0.jar
- gson-2.8.0.jar

Our application consists of a POJO class called `News` that has four attributes: `id`, `title`, `category`, and link. It also consists of a `Main` class that reads the entire information from the news.json file as a list of `News` objects. A for loop iterates over the entire list of `News`, formats the results so that everything is uppercase, and then logs the results.

Listing 8-5 shows the `News` class.

Listing 8-5. The News Class

```
package org.news;

public class News {

    private String id;

    private String title;

    private String category;

    private String link;
```

```
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public String getLink() {
        return link;
    }

    public void setLink(String link) {
        this.link = link;
    }

    @Override
    public String toString() {
        return "Id: " + id + " - " + "Title: " + title + " - " + "Category: " + category +
" - " + "Link: " + link;
    }
}
```

Listing 8-6 represents the `Main` class, which imports packages from Gson, Guava, and SLF4J, reads the news.json, logs its content, and also formats it.

Listing 8-6. The Main Class of our Application

```
package org.news;

import java.io.*;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.List;
import com.google.gson.reflect.TypeToken;
import com.google.gson.Gson;
```

```
import com.google.gson.GsonBuilder;
import com.google.common.base.CaseFormat;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Main {

    public static void main(String[] args) throws FileNotFoundException {

        Logger logger = LoggerFactory.getLogger(Main.class);

        BufferedReader bufferedReader = new BufferedReader(new FileReader("news.json"));

        Type listType = new TypeToken<ArrayList<News>>(){}.getType();
        List<News> yourClassList = new Gson().fromJson(bufferedReader, listType);

        for(News news : yourClassList) {
            logger.info("Id: " + CaseFormat.LOWER_UNDERSCORE.to(CaseFormat.UPPER_UNDERSCORE,
            news.getId()));
            logger.info("Title: " + CaseFormat.LOWER_UNDERSCORE.to(CaseFormat.UPPER_
            UNDERSCORE, news.getTitle()));
            logger.info("Category: " + CaseFormat.LOWER_UNDERSCORE.to(CaseFormat.UPPER_
            UNDERSCORE, news.getCategory()));
            logger.info("Link: " + CaseFormat.LOWER_UNDERSCORE.to(CaseFormat.UPPER_
            UNDERSCORE, news.getLink()));
        }
    }
}
```

First, we compile and run our application in JDK 9 using only the class path, so we're sure that our application works in JDK 9 without any changes.

```
javac -d out -cp "lib/gson-2.8.0.jar;lib/guava-21.0.jar;lib/slf4j-api-1.7.25.jar;lib/slf4j-
simple-1.7.25.jar" $(find src -name '*.java')
```

We create a JAR file named news.jar:

```
jar --create --file lib/news.jar -C out.
```

Finally, we run our application:

```
java -cp "lib/gson-2.8.0.jar;lib/guava-21.0.jar;lib/slf4j-api-1.7.25.jar;lib/slf4j-
simple-1.7.25.jar;lib/news.jar" org.news.Main
```

Our application is running successfully. We now have the confirmation that our not modularized application is running with JDK 9 without any changes.

Let's start the modularization process. In this part we'll modularize our News application only as part of the top-down migration strategy. We won't modularize and won't even change the four JAR files that represent our dependencies.

The first thing we do is to create a module-info.java file in the root directory. We have to figure out what kind of `requires` and `exports` clauses we need to put inside the module descriptor. We have to require our dependency JAR files inside the module descriptor and we do this by putting them on the module path so they become automatic modules. We covered the automatic modules in detail at the beginning of this chapter, where we also talked about how to find out the name of the generated automatic modules:

```
jar --describe-module --file gson-2.8.0.jar
```

By running the command `jar --describe-module` on the Gson JAR file, we find out that the generated name of the automatic module is gson. We add this name into our module descriptor and do the same for all the other JAR files, because our application depends on these. If we're not sure about the dependencies used by our application, we can run the Jdeps on our previously created news.jar file:

```
$ jdeps -cp "lib/gson-2.8.0.jar;lib/guava-21.0.jar;lib/slf4j-api-1.7.25.jar;lib/slf4j-
simple-1.7.25.jar;lib/news.jar" -s lib/news.jar

news.jar -> lib\gson-2.8.0.jar
news.jar -> lib\guava-21.0.jar
news.jar -> java.base
news.jar -> lib\slf4j-api-1.7.25.jar
```

The JDeps tool informs us that our news.jar file has dependencies on three JAR files and on module java.base.

Our module-info.java looks like this:

```
module news {
    requires slf4j.simple;
    requires slf4j.api;
    requires guava;
    requires gson;
}
```

Because our News application is standalone and isn't an API, we have no `exports` clauses. We don't need to give our application to somebody else to include it in their own application, so `exports` clauses are for the moment not necessary.

We compile our application:

```
javac -d modules --module-path lib --module-source-path src -m news
```

Now, if we take a look in the modules directory, we see that we have .class files not only for the corresponding Java classes, but also for the module-info.java we have a compiled module-info.class file.

We create a modular JAR for our application:

```
jar --create --file lib/news.jar -C modules/news.
```

Next we run our `Main` class:

```
java --module-path lib -m news/org.news.Main
```

Unfortunately, we got a `ClassNotFoundException`, which informs us that the class `java.sql.Time` can't be found:

```
Exception in thread "main" java.lang.NoClassDefFoundError: java/sql/Time
        at gson@2.8.0/com.google.gson.Gson.<init>(Gson.java:240)
        at gson@2.8.0/com.google.gson.Gson.<init>(Gson.java:174)
        at news/org.news.Main.main(Main.java:23)
Caused by: java.lang.ClassNotFoundException: java.sql.Time
        at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(Unknown Source)
        at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass
        (Unknown Source)
        at java.base/java.lang.ClassLoader.loadClass(Unknown Source)
        ... 3 more
```

The `java.sql.Time` class is located in the java.sql module. We need to add this module to the root set of modules using the `--add-modules` option so it can be resolved:

```
java --add-modules java.sql --module-path lib -m news/org.news.Main
```

The previous error doesn't appear anymore, because it was solved. Unfortunately, we get now a different type of exception:

```
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to make
field private java.lang.String org.news.News.id accessible: module news doesn't "opens org.
news" to module gson
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible
        (Unknown Source)
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible
        (Unknown Source)
        at java.base/java.lang.reflect.Field.checkCanSetAccessible(Unknown Source)
        at java.base/java.lang.reflect.Field.setAccessible(Unknown Source)
```

We get an `InaccessibleObjectException` because Gson is performing deep reflective access into the JDK and it doesn't succeed because our package org.news is by default not opened for deep reflection. We have to open our package org.news to module Gson using a qualified `opens` clause. Therefore, we need to add the following statement in our module descriptor:

```
opens org.news to gson;
```

We compile and run our application again, and, because we opened the org.news package so that the Gson library can perform deep reflection on it, it works.

In this section, we managed to migrate an application that uses JAR files to run on JDK 9. We created a module for our own code and put the JAR files on the module path by transforming them into automatic modules. We don't have any code on the class path, because the entire code is now on the module path.

■ **Note** The source code we had before starting the modularization process can be found in the directory /ch08/topDownMigrationStart. The source code after the top-down modularization process can be found in the directory /ch08/topDownMigration.

This was top-down migration, where we modularize our application and use the JAR libraries as automatic modules on the module path.

Summary

This chapter presented useful information on topics related to migration. Migrating an application to Java 9 is a multi-step process, depending on the size and the libraries the application is using.

We started this chapter by presenting the automatic modules, which help us make significant steps forward in the process of migrating to modules because they reuse existing JARs. The automatic modules can be used as a replacement for JAR files. If you don't plan to migrate your codebase to modules, you can use automatic modules instead. It's understandable to use automatic modules when the JAR file has not yet been modularized by its authors, but you should replace the automatic modules by their corresponding named modules as soon as the corresponding named modules are available.

Further we presented the JDeps tool. This is a very useful tool used to find static dependencies of a library, but it can't find reflective uses of JDK internal APIs. JDeps performs a static investigation on class level and outputs any use of JDK internal APIs. If we're using Maven, we can make use of the Maven JDeps plugin, because JDeps is very well integrated with Maven through it.

Next we talked about the encapsulation introduced in Java 9. We learned which packages are encapsulated in Java 9 and which aren't. We showed an example of breaking encapsulation of JDK internal APIs using the `--add-exports` command-line option. We also saw how to open packages, provide readability between modules, and add modules to the root set of modules. The `--illegal-access` option introduced to allow deep reflection for code in the class path was covered in detail.

When talking about changes in the area of the JDK internal APIs in Java 9, we must distinguish between accessing the internal APIs and accessing the internal APIs using reflection. The first isn't so surprising due to the fact that for a long time the JDK internal APIs were marked as deprecated. The latter is different because no deprecation warning can be thrown as the code is executed at runtime. As a conclusion, normal access to JDK internal APIs isn't possible in Java 9 anymore, but reflective access to JDK internal APIs is still possible, though limited.

We also discussed and gave solutions to some of the most common issues that can occur during migration to Java 9: encapsulated JDK internal APIs, not resolved modules, cyclic dependencies, new versioning scheme, split packages, removed methods in Java 9, and removal of rt.jar, tools.jar, and dt.jar. An important problem that can occur when moving to Java 9 is the split packages problem. A split package is a single package located inside two or more modules.

We finished this chapter by showing an example of migrating a small application that uses some third-party libraries to Java 9. We migrated the application step-by-step using the top-down approach. During migration, the messages displayed in the exceptions and errors give valuable hints for solving the root cause of the problem and for moving forward.

In Chapter 9, we'll learn about the new API introduced in JDK 9 for handling modules, module descriptors, module references, and layers.

CHAPTER 9

The New Module API

Project Jigsaw introduced a new layer in JDK 9, modules, and added many new features to the Java platform. Among the new features, a new module API was introduced for handling the work with modules. The new module API was added in the module java.base in the packages java.lang and java.lang.module. It contains a set of classes, interfaces, enumerations (enums), and exceptions that can be used to work with modules.

The module API allows us to perform different operations on modules, such as extracting the information from the module descriptors, accessing the resources of a module, searching for modules on the module path or for all the system modules, creating layers, and more. The module API can also be used to dynamically add `reads`, `opens`, or exports `directives` to other modules at runtime. We'll start this chapter by briefly presenting the structure of the new module API: its interfaces, classes, enums, and exceptions.

According to the JDK 9 API specification, two interfaces were added for working with modules. Table 9-1 shows the new interfaces of the module API.

Table 9-1. *The Interfaces of the Module API*

Name	Type	Description
`ModuleFinder`	Interface	Finds modules during resolution or service binding
`ModuleReader`	Interface	Accesses the module's content

Table 9-2 lists the classes of the new module API.

Table 9-2. *The Classes of the Module API*

Name	Type	Description
`Configuration`	Class	A configuration that contains the readability graph
`Module`	Class	A module at runtime
`ModuleDescriptor`	Class	A module descriptor
`ModuleDescriptor.Builder`	Class	A builder for creating module descriptors
`ModuleDescriptor.Exports`	Class	A package exported by a module
`ModuleDescriptor.Opens`	Class	A package opened by a module
`ModuleDescriptor.Provides`	Class	A service with implementations provided by the module
`ModuleDescriptor.Requires`	Class	A dependence upon a module
`ModuleDescriptor.Version`	Class	A module version
`ModuleReference`	Class	A reference to a module
`ResolvedModule`	Class	A module in a graph of resolved modules

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_9

Table 9-3 shows the enumerations of the module API.

Table 9-3. *The Enumerations of the Module API*

Name	Type	Description
`ModuleDescriptor.Exports.Modifier`	Enum	A modifier on an exported package
`ModuleDescriptor.Modifier`	Enum	A modifier on an module
`ModuleDescriptor.Opens. Modifier`	Enum	A modifier on an open package
`ModuleDescriptor.Requires.Modifier`	Enum	A modifier on a module dependence

Table 9-4 shows the exceptions of the module API.

Table 9-4. *The Exceptions of the Module API*

Name	Type	Description
`FindException`	Exception	Thrown when an error occurs while finding a module
`InvalidModuleDescriptorException`	Exception	Thrown when a module descriptor has an invalid format
`ResolutionException`	Exception	Thrown when the process of resolving a set of modules fails
`LayerInstantiationException`	Exception	Thrown when an error occurs while creating a module layer

We'll next look at some of the most important classes and interfaces of the new module API in detail, starting with the fundamental one, the `Module` class.

The Module Class

The `Module` class represents a module at runtime, which can be either a named module or an unnamed module. The `Module` class was introduced in Java 9 and is located in module java.base in the package java.lang in the following location:

```
\java.base\share\clasess\java\lang\Module.java
```

Let's explore the attributes, constructors, and methods defined by the `Module` class.

Attributes

Following are some of the most important attributes defined by the `Module` class:

- `ModuleLayer layer`: Represents the layer that contains this module. The layer can also be null.
- `String name`: Represents the name of this module.
- `ClassLoader loader`: Represents the `ClassLoader` of this module.

- `ModuleDescriptor descriptor`: Represents the `ModuleDescriptor` of this module.
- `Map<String, Set<Module>> exportedPackages`: Represents the packages that are exported by this module.
- `Map<String, Set<Module>> openPackages`: Represents the packages that are open to other modules.
- `Set<Module> reads`: Represents the modules that this module reads.
- `static final Module ALL_UNNAMED_MODULE`: Represents a special module that defines the entire set of unnamed modules. An unnamed module has no ClassLoader object, no ModuleLayer object and no ModuleDescriptor object defined.

Constructors

The `Module` class has three types of constructors, as stated in the JDK 9 API specification:

- `Module(ClassLoader loader)`: This constructor creates the unnamed module for the given `ClassLoader`. The `ClassLoader` object can also be null. An unnamed module contains no `ModuleDescriptor` and no `ModuleLayer`.
- `Module(ClassLoader loader, ModuleDescriptor descriptor)`: This constructor creates a named module that's not inside a `ModuleLayer`.
- `Module(ModuleLayer layer, ClassLoader loader, ModuleDescriptor descriptor, URI uri)`: This constructor creates a named module inside a `ModuleLayer`, which means that the module is represented inside the virtual machine.

Methods

The most important methods defined by the Module class are, according to the JDK 9 API specification, the following:

- `boolean isNamed()`: Returns true if the module is a named module and false otherwise.
- `String getName()`: Returns the name of the module if the module is a named module. If the module is in an unnamed module, it returns null.
- `ClassLoader getClassLoader()`: Returns the `ClassLoader` for this module.
- `ModuleDescriptor getDescriptor()`: Returns the `ModuleDescriptor` for this module if the module is a named module. If the module is an unnamed module, it returns null.
- `ModuleLayer getLayer()`: Returns the `ModuleLayer` that contains this module. If this module is not in a module layer, it returns null. If the module is in an unnamed module, null is returned.
- `boolean canRead(Module other)`: Returns true if this module reads the module given as parameter and false otherwise. If our module is an unnamed module, then true will be always returned, since an unnamed module reads all the modules.
- `Module addReads(Module other)`: Updates this module to read the given module.

- `boolean isExported(String packageName)`: Returns true if this module exports the given package. If our module is in an unnamed module, then true is by default returned. If the package is opened, then this method returns true because an opened package is also exported at runtime.
- `boolean isExported(String packageName, Module other)`: Returns true if this module exports the given package to at least the given module.
- `boolean isOpen(String packageName)`: Returns true if this module has opened the given package. If the module is in an unnamed module, then true is always returned.
- `boolean isOpen(String packageName, Module other)`: Returns true if this module has opened the given package to at least the given module specified as the second parameter.
- `Module addExports(String packageName, Module other)`: Updates this module to export the given package to the given module.
- `Module addOpens(String packageName, Module other)`: Updates this module to open the given package to the given module.
- `Module addUses(Class<?> service)`: Updates this module to add a service dependence on the given service type.
- `boolean canUse(Class<?> service)`: Returns true if this module has a service dependence on the given service type. If our module is an automatic module or is in an unnamed module, the method returns true by default.
- `Set<String> getPackages()`: Returns a set of package names for all the packages in this module.
- `InputStream getResourceAsStream(String name)`: Returns an `InputStream` object for reading a resource in this module. The resource is identified by the given name.

Changes in java.lang.Class

The class `java.lang.Class` has been enhanced with three methods in order to better fit into the newly added module system. Here are the methods added in Java 9 inside the `Class` class, according to the JDK 9 API specification:

- `Class<?> forName(Module module, String className)`: Returns the `Class` object of the given class from the given module.
- `Module getModule()`: Returns the module that this class or interface is a member of.
- `String getPackageName()`: Returns a `String` representing the name of the package of this class.

The ModuleDescriptor class

The `ModuleDescriptor` class is located inside the java.lang.module package of the java.base module. This class represents a module descriptor of a named module. An instance of this class expresses a module descriptor that's obtained from a module-info.class file.

Because a ModuleDescriptor object is immutable, obtaining a ModuleDescriptor object is straightforward. It's obtained by calling the getDescriptor() method on the corresponding Module object:

```
java.lang.Module myModule = MyClass.class.getModule();
java.lang.ModuleDescriptor myModuleDescriptor = myModule.getDescriptor();
```

But this is not the only way we can get a ModuleDescriptor object. We can create a ModuleDescriptor using the build() method of the Builder class, which is an inner class of the ModuleDescriptor class. Creating a ModuleDescriptor using the Builder class is not the common approach, so we don't cover it in this book.

■ **Note** A module descriptor can't describe the unnamed module. It describes all the existing types of modules except the unnamed module: normal modules, open modules, and automatic modules.

The ModuleDescriptor class consists of a set of nested classes, discussed later in this section:

- ModuleDescriptor.Builder class
- ModuleDescriptor.Exports class
- ModuleDescriptor.Modifier class
- ModuleDescriptor.Opens class
- ModuleDescriptor.Provides class
- ModuleDescriptor.Requires class
- ModuleDescriptor.Version class

The instances of four of these nested classes (ModuleDescriptor.Exports, ModuleDescriptor.Opens, ModuleDescriptor.Provides, and ModuleDescriptor.Requires) represent statements that can reside inside a module declaration module-info.java file: exports, opens, provides, and requires.

We'll continue by looking at some of the attributes defined inside the ModuleDescriptor class.

ModuleDescriptor Attributes

The most important attributes defined by the ModuleDescriptor class are the following, and they're described more fully in the next subsection:

- String name
- Version version
- Set<Modifier> modifiers
- boolean open
- boolean automatic
- Set<Requires> requires
- Set<Exports> exports
- Set<Opens> opens
- Set<String> uses
- Set<Provides> provides

- `Set<String> packages`
- `String mainClass`
- `static enum Modifier {OPEN, AUTOMATIC, SYNTHETIC, MANDATED}`

Next up: the most important methods of the `ModuleDescriptor` class.

ModuleDescriptor Methods

The JDK 9 API specification defines a couple of methods for the ModuleDescriptor class. The most important methods of the ModuleDescriptor class are as follows:

- `String name()`: Returns the name of the module.
- `Set<Modifier> modifiers()`: Returns a set of the `Modifier` enum, which represents the module modifiers. A `Modifier` enum contains the following values: `OPEN`, `AUTOMATIC`, `SYNTHETHIC`, and `MANDATE. Modifier.OPEN` denotes an open module. `Modifier.AUTOMATIC` represents an automatic module. `Modifier.SYNTHETIC` specifies that the module wasn't declared, either explicitly nor implicitly. `Modifier.MANDATED` states that the module was implicitly declared.
- `boolean isOpen()`: Returns true if the module is open and false otherwise.
- `boolean isAutomatic()`: Returns true if this is an automatic module and false otherwise.
- `Set<Requires> requires()`: Returns a set of `Requires` objects that denote the dependencies of the module.
- `Set<Exports> exports()`: Returns a set of `Exports` objects that stand for the exported packages of the module.
- `Set<Opens> opens()`:Returns a set of `Opens` objects that represent the open packages.
- `Set<String> uses()`: Returns a set of `String`s that represent the service dependencies of the module.
- `Set<Provides> provides()`: Returns a set of `Provides` objects that expresses the services provided by the module.
- `Optional<Version> version()`: Returns the version of the module.
- `String toNameAndVersion()`: Returns the module name and the version formatted as `<module_name>@<version>`.
- `Optional<String> mainClass()`: Returns the `main` class of the module.
- `Set<String> packages()`: Returns a set of `String`s that expresses the packages from the module.

The following sections focus on the nested classes of the `ModuleDescriptor` class.

The ModuleDescriptor.Requires Class

The `ModuleDescriptor.Requires` class, whose instance expresses a `requires` clause in a module descriptor, contains some attributes and methods, described next.

It contains an enum called `Modifier` with the following values:

- `TRANSITIVE`: As stated in the official JDK 9 documentation, "this dependence causes any module which depends on current module to have an implicitly declared dependence on the module named by the `requires`."
- `STATIC`: This dependence is mandatory at compile-time but is optional at runtime.
- `SYNTHETIC`: This dependence wasn't declared in the module declaration.
- `MANDATED`: This dependence was declared in the module declaration.

The methods defined by the `ModuleDescriptor.Requires` class are as follows:

- `Set<Modifier> modifiers()`: Returns a set of `Modifier` objects
- `String name()`: Returns the name of the module
- `Optional<Version> compiledVersion()`: Returns a `Version` object representing the version of the module
- `Optional<String> rawCompiledVersion()`: Returns a `String` representing the unparseable version of the module

The ModuleDescriptor.Exports Class

The `ModuleDescriptor.Exports` class can be instantiated to express an `exports` clause inside the module declaration. The class contains an enum called `Modifier` with the following values: `SYNTHETIC` and `MANDATED`.

Here are the methods defined by this class:

- `Set<Modifier> modifiers()`: Returns a set of `Modifier` objects
- `boolean isQualified()`: Returns true if the export is qualified or false otherwise
- `String source()`: Returns a `String` representing the name of the package
- `Set<String> targets()`: Returns a set representing the names of the modules to which the package is exported. If the export is unqualified, it returns an empty set.

The ModuleDescriptor.Opens Class

The `ModuleDescriptor.Opens` class can be instantiated to express an `opens` clause inside the module declaration. The class contains an enum called `Modifier` with the following values: `Modifier.SYNTHETIC` and `Modifier.MANDATED`. In this case, `MANDATED` means that the `opens` statement was declared in the module declaration, and `SYNTHETIC` means that the `opens` statement was not declared in the module declaration.

According to the JDK 9 API specification, the methods defined by this class are as follows:

- `Set<Modifier> modifiers()`: Returns a set of `Modifier` objects.
- `boolean isQualified()`: Returns true if it is a qualified `opens` operation and false otherwise. A qualified `opens` operation is characterized by the fact that the modules to which the package is opened are specified in the module declaration. An *unqualified* `opens` operation doesn't specify any modules in the module declaration, which means that the package is opened to all the modules.

- `String source()`: Returns the name of the package as `String`.
- `Set<String> targets()`: Returns a set of `String` that represents the names of the modules to which the package is open, but for an unqualified `opens` it returns an empty set.

The ModuleDescriptor.Provides Class

The `ModuleDescriptor.Provides` class is, in a manner of speaking, the correspondent of the `provides` statement from the module-info.java file. To recap, the `provides` statement's role is to define a service type. The syntax of the `provides` statement is like this: `provides <interface_name> with <class_name>`, where `<class_name>` represents the implementation class for the service type defined by the interface with the name `<interface_name>`.

The `ModuleDescriptor.Provides` class defines the following variables: `String` service and `List<String>` providers. This means we can create one instance of the `ModuleDescriptor.Provides` class to define a single service and one or more providers.

There are two methods defined by the `ModuleDescriptor.Provides` class:

- `String service()`: This method returns the fully qualified class name of the service type.
- `List<String> providers()`: This method returns a list of `String`s representing the fully qualified class names of the providers or provider factories.

Further, we can define three `provides` statements inside our module descriptor:

```
module com.apress.myModule {
        provides ServiceType1 with package1.Class1;
        provides ServiceType1 with package1.Class2;
        provides ServiceType1 with package2.Class3;
}
```

Because we have only a single instance of the service type, we have only one instance of the class `ModuleDescriptor.Provides`. We can get all the names of the providers classes by calling the method `providers()` and the name of the service type by calling the method `service()`.

The ModuleDescriptor.Version Class

The nested class `ModuleDescriptor.Version` represents the version of a module. The version of a module is used only for documentation, because the JPMS does not support versioning. The most used methods provided by the `ModuleDescriptor.Version` class are as follows:

- `Version parse(String)`: Parses the given `String` as a version `String`
- `int compareTo(Version version)`: Compares this module version to the given module version

The ModuleFinder Interface

According to the JDK 9 API specification, this interface "represents a finder of modules and is used to find methods during resolution or service binding." Here are the methods contained in this interface:

- `Optional<ModuleReference> find(String moduleName)`: Finds and returns a `ModuleReference` object to a module whose name is passed as parameter
- `Set<ModuleReference> findAll()`: Returns a set containing all the `ModuleReference` objects that can be located in the system
- `static ModuleFinder ofSystem()`: Returns a `ModuleFinder` object that locates all the system modules from the Java runtime
- `static ModuleFinder of(Path… entries)`: Returns a `ModuleFinder` object that locates modules on the file system by searching a sequence of directories or packaged modules

The role of the `ModuleFinder` interface is to find modules. A `ModuleFinder` finds only a single module. It can't find more than one module. If we search for modules inside directories, the `ModuleFinder` will retrieve only the first module found.

For instance, if we have two directories and want to find a module named myModule, we could first get the `MethodFinder` by passing the sequence of directories as an argument to the method `of()` of the `ModuleFinder` interface:

```
ModuleFinder moduleFinder = ModuleFinder.of(directoryA, directoryB);
```

We can call the `find()` method on the the `ModuleFinder` object by passing the name of the module we're searching for:

```
Optional<ModuleReference> moduleReference = moduleFinder.find("myModule");
```

The `find()` method returns a `ModuleReference` object to the module with the name myModule. From a `ModuleReference` object, we can derive a `ModuleDescriptor` object:

```
if(moduleReference.isPresent()) {
        ModuleDescriptor moduleDescriptor = moduleReference.get().descriptor();
}
```

The `findAll()` method returns a set of all the module references that can be located. Hence, we could then find all the modules located inside the two directories.

The ModuleReader Interface

The role of the `ModuleReader` interface is to help access the content of a module. As stated in the official JDK 9 API documentation, "a module reader is intended for cases where access to the resources in a module is required, regardless of whether the module has been loaded. A framework that scans a collection of packaged modules on the file system, for example, may use a module reader to access a specific resource in each module."

The JDK 9 API specification defines a couple of methods for the ModuleReader interface. The most important methods defined by the `ModuleReader` interface are as follows:

- `Optional<URI> find(String resourceName)`: This method finds the resource identified by the name `resourceName`. It returns a URI object to the resource in the module. It can throw an I/O Exception if the module reader is closed.
- `Optional<InputStream> open(String resourceName)`: This method opens a resource with name `resourceName`. It returns an `InputStream` object to read the resource in the module.
- `Optional<ByteBuffer> read(String resourceName)`: This method reads the given resource and returns a `ByteBuffer` object that contains the contents of the resource.
- `Stream<String> list()`: This method lists the contents of the module. It returns a `Stream` of `String` objects that represents the names of all the resources in the module. Like the `find(resourceName)` and `open(resourceName)` methods, it can throw an I/O Exception if the module reader is closed.

The following example uses what we've learned so far to read some information from a module. We will search inside the java.base module for all the implementation classes. Once we find them, we load them and then print their name and the name of their package.

Listing 9-1 shows the class `ModuleReaderExample`, which loads all the implementation classes from module java.base and then prints their name and their package name:

Listing 9-1. The Class ModuleReaderExample

```
package com.apress.apimodule;

import java.io.IOException;
import java.io.UncheckedIOException;
import java.lang.module.ModuleFinder;
import java.lang.module.ModuleReader;
import java.lang.module.ModuleReference;
import java.util.*;

public class ModuleReaderExample {

    public static void main(String[] args) {

        List<Class<?>> listClasses = getClassesByModuleName("java.base");

        for(Class<?> myClass : listClasses) {
            System.out.println("Name of the class is: " + myClass.getName());
            System.out.println("Name of the package is: " + myClass.getPackageName());
        }
    }

    private static List<Class<?>> getClassesByModuleName(String moduleName) {

        ModuleFinder finder = ModuleFinder.ofSystem();
        Optional<ModuleReference> optionalModuleReference = finder.find(moduleName);
        ModuleReference moduleReference = optionalModuleReference.get();
```

```
        try (ModuleReader moduleReader = moduleReference.open()) {
            return moduleReader.list()
                    .filter(name -> name.endsWith("Impl.class"))
                    .map(ModuleReaderExample::classLoadByFileName)
                    .collect(Collectors.toList());
        } catch(IOException ioException) {
            throw new UncheckedIOException(ioException);
        }
    }

    private static Class<?> classLoadByFileName(String classFileName) {

        ClassLoader classLoader = ModuleReaderExample.class.getClassLoader();
        String nameOfClass = classFileName.substring(0, classFileName.length() - ".class".length());

        try {
            nameOfClass = nameOfClass.replace('/','.');
            return classLoader.loadClass(nameOfClass);
        }
        catch (ClassNotFoundException classNotFoundException) {
            throw new UncheckedIOException(new IOException(classNotFoundException));
        }
    }

}
```

The method `static Class<?> classLoadByFileName(String classFileName)` is simple. It uses the `ClassLoader` object to load the given class and returns an instance of object `Class`. The method `List<Class<?>> getClassesByModuleName(String moduleName)` is interesting. It returns a list of `Class` objects that are located in the given module. First, we get an instance of the `ModuleFinder` by calling the method `ofSystem()`. This method returns a module finder that locates the system modules. Then we find a reference to a module with the name moduleName by calling the `find(moduleName)` method on the `finder` object:

```
Optional<ModuleReference> optionalModuleReference = finder.find(moduleName);
ModuleReference moduleReference = optionalModuleReference.get();
```

A `ModuleReference` object is a reference to the module's content—in our case, module java.base. To open the module for reading, we call the method `open()` on the `ModuleReference` object. Thus we obtain an instance of a `ModuleReader` object:

```
ModuleReader moduleReader = moduleReference.open()
```

Next we call the `list()` method on the `ModuleReader` object and filter the results by searching only for the classes with names ending in `Impl.class`. In the end, we call the method `classLoadByFileName()` to get a `Class` object of the corresponding class. In the `main` method, we load all the implementation classes of the module java.base and print their names together with their package names.

The output is huge, so we'll show only a small snippet of it here:

```
Name of the class is: sun.util.locale.provider.BreakIteratorProviderImpl
Name of the package is: sun.util.locale.provider
Name of the class is: java.lang.ProcessImpl
Name of the package is: java.lang
```

```
Name of the class is: java.util.jar.JavaUtilJarAccessImpl
Name of the package is: java.util.jar
...
```

Performing Operations on Modules

This section covers some operations we can perform on modules programmatically, such as getting the module of a class, accessing the resources of a module, searching for all the modules in the module path, or getting the module information,

Getting the Module of a Class

As we already learned in this chapter, a module at runtime is expressed by the `Module` class defined in package java.lang of the module java.base. The `Module` class can represent either a named or unnamed module. In order to return a `Module` object for our class called `ModuleCore`, we call the method `getModule()` of the class `Class`:

```
Class<ModuleCore> myClass = ModuleCore.class;
Module module = myClass.getModule();
```

This method returns a module that the class `ModuleCore` is a member of. If the class is in the unnamed module, the method `getUnnamedModule()` from `ClassLoader.java` is called.

Accessing Resources of a Module

Listing 9-2 shows how we can access resources of the module using the `getResourceAsStream()` method, which returns an `InputStream` object.

Listing 9-2. Accessing Resources of a Module Using Method getResourceAsStream()

```
this.class.getModule().getResourceAsStream("file.properties");
```

We can also access resources of a module using the `getResource()` method, which returns a URL object, as in Listing 9-3.

Listing 9-3. Accessing Resources of a Module Using Method getResource()

```
this.getClass().getResource("file.properties")
ClassLoader.getPlatformClassLoader().getResource("file.properties")
```

Searching for all Modules in the Module Path

Using the new module API, we can even find all the modules in the module path. Listing 9-4 shows how to search for all the modules in the module path in the system environment variable `jdk.module.path`, get their module descriptors, and print their module names.

Listing 9-4. Searching for All the Modules in the Module Path

```
ModuleFinder.of(Paths.get(System.getProperty("jdk.module.path"))).
        .findAll()
        .stream()
        .forEach(ref -> {
                System.out.println(moduleReference.descriptor().name());
});
```

Getting Module Information

Using the methods and classes described throughout this chapter, we can get complete information about a module. Listing 9-5 shows an example where we make use of the new classes and interfaces in order to get all the available information from the module java.base.

Listing 9-5. Print Extensive Information from Module java.base

```
import java.lang.module.ModuleDescriptor;
import java.lang.module.ModuleFinder;
import java.lang.module.ModuleReference;
import java.util.NoSuchElementException;
import java.util.Optional;
import java.util.Set;

public class BaseModule {

    public static void main(String[] args) {

        String moduleName;
        Optional<String> mainClass;
        Set<ModuleDescriptor.Exports> exports;
        boolean isAutomatic;
        boolean isOpen;
        Set<String> allPackagesNames;
        Set<ModuleDescriptor.Provides> provides;
        Set<ModuleDescriptor.Requires> dependencies;
        String moduleNameVersion;
        Set<String> serviceDependencies;
        ModuleDescriptor.Version version;

        ModuleFinder finder = ModuleFinder.ofSystem();
        Optional<ModuleReference> moduleReference = finder.find("java.base");

        if(moduleReference.isPresent()) {
            ModuleDescriptor moduleDescriptor = moduleReference.get().descriptor();

            // get the name of the module
            moduleName = moduleDescriptor.name();

            // get the module's main class
            mainClass = moduleDescriptor.mainClass();
```

```
exports = moduleDescriptor.exports();
isAutomatic = moduleDescriptor.isAutomatic();
isOpen = moduleDescriptor.isOpen();
allPackagesNames = moduleDescriptor.packages();
provides = moduleDescriptor.provides();
dependencies = moduleDescriptor.requires();
moduleNameVersion = moduleDescriptor.toNameAndVersion();
serviceDependencies = moduleDescriptor.uses();

try {
    Optional<ModuleDescriptor.Version> versionOptional = moduleDescriptor.
    version();
    version = versionOptional.get();
}
catch (NoSuchElementException exception) {
    version = null;
}

System.out.println("Module name is: " + moduleName);
System.out.println();

System.out.println("Main class is: ");
if(mainClass.isPresent())  {
    System.out.println(mainClass);
}
else {
    System.out.println("Not exists");
}
System.out.println();

System.out.println("The module exports the packages with the following name: ");
for(ModuleDescriptor.Exports moduleExport : exports) {
    System.out.print(moduleExport.source());
    System.out.print(", ");
}

System.out.println();
System.out.println();
System.out.println("Is an automatic module: " + isAutomatic);

System.out.println();
System.out.println("Is an open module: " + isOpen);

System.out.println();
System.out.println("All packages names: ");
for(String packageName : allPackagesNames) {
    System.out.print(packageName);
    System.out.print(", ");
}
```

```
            System.out.println();
            System.out.println();
            System.out.println("The services provided by the module: ");
            for(ModuleDescriptor.Provides provide : provides) {
                System.out.print("Service " + provide.service());
                for(String p : provide.providers()) {
                    System.out.print(" with providers: " + p);
                    System.out.print(", ");
                }
            }
            System.out.println();
            System.out.println("The name of the dependencies of the module: ");
            for(ModuleDescriptor.Requires dependency : dependencies) {
                System.out.print(dependency.name());
                System.out.print(", ");
            }

            System.out.println();
            System.out.println("Module name and version: " + moduleNameVersion);

            System.out.println();
            System.out.println("The service dependencies of the module: ");
            for(String serviceDependency : serviceDependencies) {
                System.out.print(serviceDependency);
                System.out.print(", ");
            }

            System.out.println();
            System.out.println("The version of the module: " + version);
        }
    }
}
```

We use the interface `ModuleFinder` to locate all the system modules. On the resulting object, we call the `find()` method and pass the string `"java.base"` representing the module's name. This will return a `ModuleReference` to the java.base module. We verify whether the `ModuleReference` was found using the `isPresent()` method. Further, we call the `descriptor()` method on the `ModuleReference` object in order to get the module descriptor. The `ModuleDescriptor` object contains comprehensive information about a module, such as its name, its `main` class, its packages, its dependencies, its service dependencies, its provided services, its version name, and so on. We retrieve this information and print it. Listing 9-6 shows only the most important parts of the output.

Listing 9-6. Output After Running the Preceding Code that Prints the Information Regarding Module java.base

```
Module name is: java.base

Main class is:
Not exists

The module exports the packages with the following name:
jdk.internal.module, javax.net.ssl, java.time.format, java.nio.charset.spi, sun.security.
ssl, sun.security.pkcs, sun.security.internal.interfaces, jdk.internal.util.jar, java.
security.interfaces, sun.util.logging, jdk.internal.perf, java.util.function, sun.net.util,
jdk.internal.misc, javax.security.auth.login, sun.security.x509, sun.security.rsa, jdk.
internal.util.xml, jdk.internal, java.util.jar, java.util.regex, sun.security.action, jdk.
internal.jmod, java.util.stream,
.....................................................

Is an automatic module: false

Is an open module: false

All packages names:
jdk.internal.org.objectweb.asm.signature, sun.text.bidi, sun.text.normalizer, sun.
security.action, sun.util.logging, sun.security.internal.interfaces, jdk.internal.jimage.
decompressor, jdk.internal.util.jar, java.net.spi, sun.reflect.generics.factory, sun.util.
resources.cldr, sun.security.tools, com.sun.java.util.jar.pack, java.text.spi, java.nio,
jdk.internal.ref, sun.security.tools.keytool, java.security.spec, sun.security.util, java.
nio.channels.spi, sun.net.www.protocol.ftp, java.util, sun.util.cldr, sun.reflect.generics.
reflectiveObjects, java.util.spi, java.lang.ref,
.............................

The services provided by the module:
Service java.nio.file.spi.FileSystemProvider with providers: jdk.internal.jrtfs.
JrtFileSystemProvider,
The name of the dependencies of the module:

Module name and version: java.base@9

The service dependencies of the module:
java.util.spi.LocaleNameProvider, jdk.internal.logger.DefaultLoggerFinder, java.
lang.System$LoggerFinder, sun.util.resources.LocaleData$SupplementaryResourceBundl
eProvider, java.text.spi.NumberFormatProvider, java.time.chrono.Chronology, java.
util.spi.CalendarNameProvider, java.text.spi.DateFormatSymbolsProvider, java.time.
zone.ZoneRulesProvider, sun.text.spi.JavaTimeDateTimePatternProvider, java.text.spi.
DecimalFormatSymbolsProvider
.........................

The version of the module: 9
```

■ **Note** The source code for the previous example can be found in the folder /ch09/moduleDescriptorJavaBase.

Summary

This chapter discussed the new module API introduced in Java 9, which gives us the means to access modules and the information inside modules.

You learned what kind of classes, interfaces, enums, and exceptions are contained inside the new module API. We talked about the `java.lang.Module` class together with its attributes, constructors, and methods. Next we showed how the `java.lang.Class` class has been enhanced with three useful methods. The new `ModuleDescriptor` class was also covered in detail. We explained its attributes and methods, but also its nested classes like `ModuleDescriptor.Requires`, `ModuleDescriptor.Exports`, `ModuleDescriptor.Opens`, `ModuleDescriptor.Provides`, and `ModuleDescriptor.Version`. We talked about the new `ModuleReader` and `ModuleFinder` interfaces and showed an example of how we can read the contents of a module. With the help of the `ModuleFinder` interface, we searched for all the implementation classes of the java.base module. Once we found them, we loaded them using the `loadClass()` method of the `ClassLoader` class. Then we displayed the names of the classes loaded together with the names of their packages.

Afterwards, we saw some examples of performing operations on modules, such as getting the module of a class, accessing the resources of a module, or searching for all the modules in the module path. The chapter concluded by discussing a Java class that reads all the properties from the module java.base and prints them at the system console.

Chapter 10 will cover some advanced topics related to Jigsaw, including layers, class loaders, multi-release JAR files, JMOD files and upgradeable modules. The layers are also part of the Module API.

CHAPTER 10

Advanced Topics

Applications with plugin and container architectures need to be able to use two important features: dynamic configuration and runtime augmentation of platform modules. This means that such applications must be able to load new additional modules at runtime, bind them into the existing application's configuration, and use them without the need to stop the application and compile it again. This type of application also needs to be able to load and configure other platform modules after a runtime image has been invoked. Jigsaw introduces such support in form of layers. *Layers* are one of the main subjects of this chapter.

This chapter presents some advanced topics on Java 9 modularity. They didn't fit in the other chapters, so we put them here instead of creating separate chapters for each of them.

This chapter covers class loaders, layers, the JMOD format, multi-release JAR files, and upgradeable modules. We'll also touch on some new features that will come in the next JDK releases and a couple of issues that were fixed.

JMOD Files

According to the JDK 9 documentation, "for the purpose of modularizing the JDK, a new artifact format called JMOD goes beyond JAR files to accommodate native code, configuration files, and other kinds of data that do not fit naturally, if at all, into JAR files."

A JMOD file is a new module artifact that consists of a compiled module definition in the form of a ZIP file. It enhances a JAR file by also including native code and configuration files. JMOD is a new format used for packaging the modules. This new format isn't executable.

A JMOD file is an alternative to the modular JAR file covered in Chapter 4. It's mostly used when a module also contains native code. JMOD files are used to package the modules of the JDK, but they can be used only at compile-time and link-time. They can't be used at runtime. The JMOD files are also used by the Jlink tool to create a modular runtime image. The directory consisting of the JMOD files represents the module path used by the linker.

The JMOD Tool

The JMOD tool can be especially used for the following:

- To create a JMOD file for a standard or JDK-specific module
- To list the content of an existing JMOD file

The JMOD tool has been extended in order to be able to install a module as a JAR file with a module-info in the top level directory. In other words, the JMOD tool is to the new JMOD format similar to what the jar tool is to the JAR format.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_10

Using the `jmod` command we can create a new JMOD archive:

```
jmod create <options> <jmod-file>
```

The `jmod create` command creates a new JMOD archive named <jmod-file>. Table 10-1 lists some of the most important options of the JMOD tool, as specified in the JDK specification.

Table 10-1. *Summary of the Options Provided by the JMOD Tool*

Option	Description
`--class-path <path>`	Specifies the application JAR files that contain classes.
`--config <path>`	Defines directories holding user-editable configuration files that are copied into the JMOD file.
`--exclude <pattern-list>`	The files matching `<pattern-list>` won't be copied into the JMOD file. `<pattern-list>` is comma-separated and can have one of the formats: `<glob-pattern>`, `glob:<glob-pattern>`, or `regex:<regex-pattern>`.
`--libs <path>`	Defines the directories containing native libraries that are copied into the JMOD file.
`--main-class <class-name>`	Defines the `main` class.
`--module-version <module-version>`	Defines the module version.
`--module-path <path> or -p <path>`	Specifies the module path where to find the modules with content that will be copied into the JMOD file.

The `jmod list` command prints the names of all the entries contained in the JMOD file passed as parameter:

```
jmod list <jmod-file>
```

The `jmod describe` command prints the details of the module contained in the JMOD file passed as parameter:

```
jmod describe <jmod-file>
```

The next section discusses the basics of the multi-release JAR files.

Multi-release JAR files

Suppose you want to switch to the latest JDK version but have a third-party library that's incompatible with the latest version of the JDK. As a consequence, you decide not to switch to the latest JDK version at least until the third-party library you're using will be made compatible with the latest version of the JDK. This is a bad scenario that was solved in Java 9 by the introduction of *multi-release JAR files*, which allow packaging code for different versions of the JDK in a single JAR file.

Multi-release JAR files were implemented in JDK 9 in JEP 238. This JEP isn't part of Jigsaw. We mention it here because it's an important feature that helps during migration to Java 9. It doesn't depend on the Java Platform Module System at all. We can use multi-release JAR files in a non-modular world, too.

Java 9 enhances the JAR file format so that multiple major versions of a class can be stored inside a single JAR file, in its META-INF directory. The new JAR file format is called a *multi-release JAR* and can consist of a single library for different JDK versions. The correct versions of the classes are loaded at runtime depending on the JDK version used by the user. A multi-release JAR file changes the structure of a JAR file only to a low degree.

■ **Note** Multi-release JAR files are supported for both normal JARs and modular JARs.

Listing 10-1 shows a multi-release JAR file that contains versioning metadata for JDK 8 and JDK 9.

Listing 10-1. Multi-release JAR file Having Versioning Metadata for JDK 8 and JDK 9

```
Root of JAR
        -   A1.class
        -   B1.class
        -   C1.class
        -   D1.class
        -   E1.class
-META-INF
        - MANIFEST.MF
        - versions
                - 8
                        - A1.class
                        - B1.class
                        - F1.class
                - 9
                        - A1.class
                        - C1.class
                        - D1.class
                        - F1.class
                        - G1.class
```

In this example, we have Java .class files in the root directory of the JAR, but also in the 8 and 9 directories. In the META-INF directory, we have the MANIFEST.MF file and a directory called versions, which contains two directories: a directory called 8 that represents the resources provided for JDK 8 and a directory called 9 that represents the resources provided for JDK 9. There are Java .class files in the 8 directory and in the 9 directory. The classes inside the 8 directory will be considered when we use JDK 8, and the classes inside the 9 directory will be considered when we use JDK 9.

The classes in the root directory of the JAR file will be considered in the following situations:

- If we're using a version of JDK different from JDK 8 or JDK 9
- If we're using JDK 8 or JDK 9 and the corresponding classes aren't present in the versions/8 and versions/9 directories

We'll explain this in detail. For the multi-release JAR file just discussed, we first have to know whether the JDK version we're using supports multi-release JAR files. If it doesn't, then everything inside the versions/8 and versions/9 directories will be ignored, and only the class files from the root directory will be considered: A1.class, B1.class, C1.class, D1.class, and E1.class.

When we use a JDK version different from JDK 8 or JDK 9, only the class files from the root directory will be used. In our example, the classes F1 and G1, present only in the versions/9 directory, won't be used at all.

When we use JDK 8, the following classes will be used:

```
A1.class (from the versions/8 directory)
B1.class (from the versions/8 directory)
C1.class (from the root directory)
D1.class (from the root directory)
E1.class (from the root directory)
F1.class (from the versions/8 directory)
```

Instead, when we use JDK 9, the following classes will be used:

```
A1.class (from the versions/9 directory)
B1.class (from the root directory)
C1.class (from the versions/9 directory)
D1.class (from the versions/9 directory)
E1.class (from the root directory)
F1.class (from the versions/9 directory)
G1.class (from the versions/9 directory)
```

As you can see from this example, the classes inside a specific 8 or 9 directory, if present, override the classes from the root directory. But this happens only if we're using JDK 8 or JDK 9, respectively.

A module-info.class file can also be added inside the versions directory of a multi-release JAR file. This feature is supported by the jar tool. But a module-info.class can't be placed in the root directory.

■ **Note** Java Compiler, Java Class File Disassembler, and JDeps are able to handle multi-release JAR files.

A multi-release JAR file has an attribute called `Multi-Release` set to true, declared in its MANIFEST.MF:

```
Multi-Release: true
```

This attribute distinguishes a multi-release JAR from a non-multi-release one. If the attribute is set to false or is missing, we have a normal JAR.

A multi-release JAR file retains the structure of a JAR file. What a multi-release JAR adds is a directory called versions, under the META-INF directory. This directory can contain subdirectories for specific major JDK versions: 6, 7, 8, 9, and so on. Inside these subdirectories we can put the .class files specific to that JDK version.

■ **Note** A multi-release JAR file supports only major versions of the JDK. A minor or a security version can't be put in a multi-release JAR file.

What happens if a version of the JDK doesn't support multi-release JARs? In this case, only the classes and resources present in the root of the JAR files are visible. Everything inside the versions directory will be invisible and implicitly not be taken into account.

■ **Note** Jlink has been enhanced with support for creating images with modules that are packaged as multi-release JAR files. Jlink adds the classes for the right version into the Jimage.

Next, we'll explain how to build a multi-release JAR file.

Build a Multi-release JAR File

A multi-release JAR file is built with the jar tool using its new `--release` command-line option. The syntax goes like this:

```
jar --create --file --release <version_number> <options>
```

- `<version_number>` represents the major version of the JDK.
- `<options>` represents a set of other options.

Suppose we have a class that's supported only in JDK 9. We want to build a multi-release JAR file and put this specific class in the versions/9 directory—everything else should be put in the root directory. For this, we use the `--release` option and specify the version, which in our case is 9, and the location of the directory that contains the class that will be put in the versions/9 directory—in our case, classesDirectoryJDK9:

```
jar --create --file myMultiReleaseJar.jar -C classesDirectoryJDK8 --release 9 -C
classesDirectoryJDK9 .
```

This command creates a multi-release JAR file by doing the following:

- Taking all the files from the directory classesDirectoryJDK8 and putting them into the root directory of the multi-release JAR file.
- Taking all the files from the directory classesDirectoryJDK9 and putting them into the versions/9 directory of the multi-release JAR file.

We now know to create a multi-release JAR file by specifying different sources for specific major version of the JDK. Next let's find out how to update a multi-release JAR file.

Update Multi-release JAR Files

It's possible to update multi-release JAR files using the jar tool by adding different versions of the module descriptor in the versions directory. Therefore, we use the option `--update` of the jar tool.

We update the previous multi-release JAR file created earlier and add some classes specific to the upcoming JDK 10:

```
jar --update --file myMultiReleaseJAR.jar --release 10 -C classesDirectoryJDK10 .
```

The multi-release JAR is updated, and the content of the classesDirectoryJDK10 directory are placed inside a new directory called versions/10. This new directory will be considered for JDK 10.

Class Loading Mechanism in JDK 9

This section covers the class loading mechanism in JDK 9. As you know, the role of a class loader is to load a class. The JCP team didn't change the class loading process in JDK 9. The Classloader API hasn't been modified in JDK 9. The same class loaders from JDK 8 are present in JDK 9: the bootstrap class loader, the platform class loader, and the application class loader.

The class loading process in JDK 9 is the same as in the previous versions of the JDK: first, the request to load a type is delegated to the parent class loader. The parent class loader delegates further to its parent class loader. This process traverses the application class loader and platform class loader and stops at the bootstrap class loader. If the bootstrap class loader can't load the type, the class loader that started the delegation process will load the type.

By examining the structural layers that compose the JDK 9, we can observe that the Java Platform Module System (JPMS) is located on top of the JVM, under the class loading architecture. Right at the bottom we have the Java Virtual Machine (JVM). Above that we have the JPMS, and above the JPMS we have the three types of class loaders mentioned earlier. This architecture is illustrated clearly in Figure 10-1.

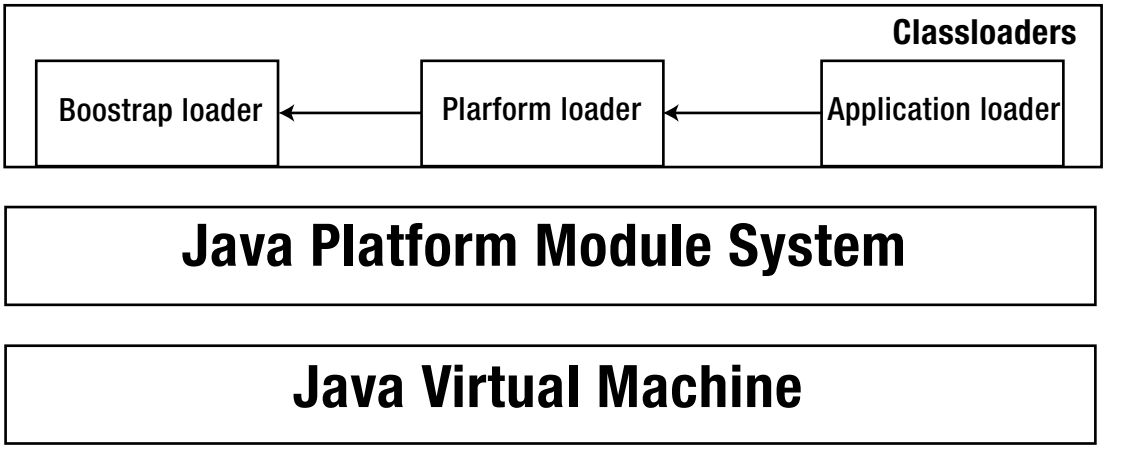

Figure 10-1. *Overview over the structure of JDK 9*

The bootstrap class loader is used to define the classes from most of the modules, like java.base, java.sql, or java.logging. The platform class loader is used to define the classes from only a few modules, like java.corba or java.transaction. Both bootstrap and platform class loader load types from platform modules, whereas the application class loader loads types from the module path. The application class loader is used to define the classes from jdk.compiler, junit, guava, slf4j, and so on. In JDK 9, the application and platform class loaders aren't instances of the class `java.net.URLClassLoader` anymore.

Jigsaw allows loading modules using our own class loaders. This can be done using the method `defineModules()` from the `Module` class, but this feature is pretty advanced, so not many developers will ever use it.

■ **Note** Every module has a class loader at runtime.

Jigsaw also introduced support for class loader names. Class loaders can have optional names. If the name isn't specified when a class loader is created, it will have no name. The name of the class loader is retrieved by the `getName()` method of the `Classloader` class. The name of the module's class loader is always mentioned together with the module name and version in warning messages or stack traces.

■ **Note** Jigsaw uses the existing class loaders and doesn't create its own class loaders.

Because a class loader can have a name attribute in JDK 9, a new constructor has been added for the `Classloader` class that creates a new class loader of the specified name by using the specified parent class loader for delegation:

```
protected ClassLoader(String name, Classloader parent)
```

Jigsaw also allows you to find a class by name in a specific module with the help of the new `findClass()` method, which gets as parameter the name of the module and the binary name of the class:

```
Class<?> findClass(String moduleName, String name)
```

This method returns the `Class` object or null if the class couldn't be found. If we pass a name for the module, then the method will always return null. Otherwise, it will call the `findClass(name)` method by passing the name of the class. This method isn't useful in a modular context unless we have our own class loader implementation that supports the loading from modules. Then we could overwrite this method.

■ **Note** Class loaders can be upgraded to load types in modules.

The Extension class loader was renamed to platform class loader in JDK 9. The name of the built-in platform class loader is platform. The new static *getPlatformClassLoader()* method returns a platform class loader via which all the built-in Java SE and JDK types are visible. This method checks for permissions and can throw a `SecurityException` exception.

A class loader can load types from multiple modules if two conditions are simultaneously met:

- All types from every module are loaded by just one single class loader.
- Modules are independent and don't conflict with each other.

To assure backwards compatibility, it's possible to load types from the class path. Every class loader has a unique unnamed module that's retrieved by the new method `getUnnamedModule()` located in the `java.lang.Classloader` class. If the class loader loads a type that isn't defined in a named module, that type is in the unnamed module. The unnamed module from the application class loader loads types from the class path when these types are in packages that aren't defined by any known module.

New Methods in the ClassLoader Class

Here are the methods added in Java 9 inside the `ClassLoader` class:

- `Class<?> findClass(String moduleName, String className)`
- `URL findResource(String moduleName, String resourceName)`
- `String getName()`

- `ClassLoader getPlatformClassLoader()`
- `Module getUnnamedModule()`

Next we'll look at the key concept of *layers*, which was introduced in JDK 9.

Layers

Suppose we want to add a couple of new modules at runtime into our application. We don't know all the modules that we need straight at compile-time, so we need the possibility of adding new modules later at runtime. Fortunately, the Java Platform Module System provides a solution for this in the form of a new concept called layers. *Layers* group a set of modules and are used to add new modules at runtime into an application. The JDK 9 API specifies that a layer maps each module in the graph to the unique class responsible for loading the types defined in that module. Therefore, a layer is used to find a class loader in order to load classes for a graph of modules.

Not all applications make use of layers. The applications that use layers are those that implement a container architecture, where modules are dynamically added and linked at runtime. On top of the existing layer, a container application can create a new layer. It does that by resolving the initial module of the application against a whole set of observable modules, like non-platform modules from the lower layer, upgradeable platform modules that can have different versions, different service providers, and so on. Nevertheless, a container application might require a different version of a module already present in the runtime environment. This can be implemented in Jigsaw using the powerful features introduced by layers.

■ **Note** Layers allow the use of more than one version of a module.

A layer is built at runtime from a graph of modules. Each layer has the following:

- A configuration, represented by an instance of the `Configuration` class located in package java.lang.module
- A function that maps each module to a class loader, represented by an instance of the `ClassLoader` class located in package java.lang

According to the JDK 9 API specification, a configuration "encapsulates the readability graph that is the output of the resolution. It is the result of resolution or resolution with service binding." The following sections talk about configurations.

■ **Note** A module can read modules from its own layer and from any layer situated lower in the layer hierarchy.

Layers can be built in a hierarchy similar to a stack. They can be created on top of the boot layer, and other layers can be created on top of the previously created layers, and so on. The Java Virtual Machine has the boot layer, which is the basic and the first layer used by the JPMS.

■ **Note** Every Java 9 modular application has at least one layer. Each layer, without the empty layer, has one or more parent layers. Layers don't have names.

The ModuleLayer class from the package java.lang in the module java.base represents an instance of a layer. A ModuleLayer object is obtained by calling the getLayer() method on a Module. A module layer contains only named modules. Hence, if we call the getLayer() method on an unnamed module, a null is returned. Otherwise a ModuleLayer object is returned:

```
ModuleLayer moduleLayer = module.getLayer();
```

Next we'll look at the boot layer, which is the most important layer.

The Boot Layer

The boot layer consists of the bootstrap loader, platform loader, and application loader. The modules in the boot layer are mapped to the bootstrap, to the platform, and to the application class loaders. The boot layer maps modules to loaders. For instance, it maps the module java.base to the bootstrap loader.

■ **Note** Most of the applications don't use any layer except the boot layer.

The JVM creates the boot layer at startup. This is done in a process called resolution. The root modules of the application are resolved together with their dependencies. The boot layer contains the module graph after all the modules have been resolved.

In most cases, the boot layer will be enough. It contains by default the module java.base. Modules in the boot layer are mapped to the bootstrap class loader and other existing class loaders from the JVM.

The boot layer can be retrieved by calling the static method boot() on a ModuleLayer class:

```
ModuleLayer bootLayer = ModuleLayer.boot();
Set<Module> modulesSet = bootLayer.modules();
```

The method boot() returns the boot layer, an object of type ModuleLayer. In order to get the set of modules from the boot layer, we called the method modules() on the resulting bootLayer object.

To get a Module object from the boot layer for our module com.apress.myModule, we can use the findModule() method of the ModuleLayer class:

```
Optional<Module> myModule = bootLayer.findModule("com.apress.myModule");
```

We could further get an InputStream for reading a resource from our module by calling the getResourceAsStream() method on the Module object:

```
InputStream inputStream = myModule.getResourceAsStream(resourceName);
```

The parameter of the method must be a path separated by / (forward slash) that identifies the resource.

Now that we have a grasp of what the boot layer is, let's move on to the new concept of configuration.

Configuration

According to the JDK 9 API specification, "a configuration encapsulates the readability graph that is the output of a resolution." To retrieve a configuration, the class ModuleLayer defines a method called configuration() that returns the configuration for this layer.

In Jigsaw, a configuration is independent and isolated from other configurations. It can also relate to other dynamically created configurations and not only to the initial configuration. Nevertheless, a configuration lets you include and to use multiple versions of non-platform modules and upgradeable platform modules that are different from those already available in the enclosing configuration. This is a very strong feature that the configurations bring in JDK 9.

The `Configuration` class is located in the module java.base in package java.lang.module. Its most important methods are the following, as described in the JDK 9 API specification:

- `Set<ResolvedModule> modules()`: Returns an immutable set of the resolved modules in this configuration.
- `Configuration resolve(ModuleFinder before, ModuleFinder after, Collection<String> roots)`: Creates a new configuration by resolving a collection of root modules with this configuration as its parent. The first parameter represents the main module finder to find modules. If no modules can be found, then the modules are searched using the module finder passed as the second parameter. The third parameter represents a collection of module names of the modules to resolve. The collection can also be empty.
- `Optional<ResolvedModule> findModule(String name)`: Finds a resolved module in this configuration. It gets as parameter the name of the module for which we want to find its `ResolvedModule` object.
- `List<Configuration> parents()`: Returns a list of this configuration's parents.
- `Configuration resolveAndBind(ModuleFinder finder, Collection<String> roots, boolean check, PrintStream output)`: Resolves a collection of root modules with service binding. It's used to create the configuration for the boot layer.

Create a Configuration

To create a new configuration, we usually take the configuration of the boot layer as a parent. To get the configuration of the boot layer, we call the method `configuration()` on the boot layer:

```
Configuration configuration = ModuleLayer.boot().configuration()
```

■ **Note** In the JVM, each layer of modules is created from a configuration.

Next we'll show how to resolve a module with a configuration.

Resolve a Module with a Configuration

To resolve a module with a configuration, we need to use the `resolve()` method described earlier. In the following example, we resolve our module name called com.apress.myModule. For this we use the configuration of the boot layer as the parent, as illustrated in Listing 10-2.

Listing 10-2. Resolving the Module com.apress.myModule with a Configuration

```
Path ourDirectory = ...;
ModuleFinder finder = ModuleFinder.of(ourDirectory);
Configuration parentConfiguration = ModuleLayer.boot().configuration();
Configuration configuration = parentConfiguration.resolve(finder, ModuleFinder.of(),
Set.of("com.apress.myModule");
```

Creating Layers

In this section we'll see how to create a layer. This is not a difficult process. The new module API provides some useful methods for creating a layer inside the `ModuleLayer` class, as specified by the official JDK 9 API specification:

- `ModuleLayer defineModules(Configuration cf, Function<String, Classloader> clf)`: Creates a new module layer with the current layer as its parent by defining the modules in the given `Configuration` to the JVM. The second parameter represents the function that maps a module name to a class loader. It returns the newly created `ModuleLayer`.
- `ModuleLayer defineModulesWithManyLoaders(Configuration cf, ClassLoader parentLoader)`: Creates a new module layer with the current layer as its parent. Each module is defined to its own `Classloader` created by this method. The second parameter represents the parent class loader for each of the class loaders created by this method.
- `ModuleLayer defineModulesWithOneLoader(Configuration cf, ClassLoader parentLoader)`: Similar to the previous method described above. The only difference is that this method creates one class loader and defines all modules to that class loader.

As you can see, in order to create a module layer we need to pass a `Configuration` object and a `ClassLoader` object. In the previous section, we learned how to create a `Configuration` object by resolving a module com.apress.myModule with the configuration for the boot layer as a parent configuration. Now that we know how to get the `Configuration`, we can create a new layer with the modules in our configuration, as demonstrated in Listing 10-3.

Listing 10-3. Create a Module Layer

```
ModuleLayer parentLayer = ModuleLayer.boot();
ClassLoader classLoader = ClassLoader.getSystemClassLoader();
ModuleLayer layer = parent.defineModulesWithOneLoader(configuration, classLoader)
```

We used the `configuration` object created earlier and passed it to the `defineModuleWithOneLoader()` method. We also passed the system class loader which was retrieved from the method `getSystemClassLoader()`.

■ **Note** Most times, the parent of an own created layer is the boot layer.

Jigsaw has to enforce constraints on the module graph because of some class loading constraints. As a result, only module graphs that contain no cycles can be transformed into a layer.

Get the Loaded Modules from a Layer

Getting the loaded modules from a layer is very simple. On the `ModuleLayer` object, the method `modules()` is called, which returns a set of the modules loaded in this layer. Listing 10-4 shows how to get all the modules from the boot layer and print their names.

Listing 10-4. Print the Names of the Modules in the Boot Layer

```
ModuleLayer moduleLayer = ModuleLayer.boot();

    moduleLayer.modules().stream().forEach(module -> {
        String moduleName = module.getName();
        System.out.println("Name of the module is: " + moduleName);
    });
```

Following in an excerpt of the output printed. We won't show it entirely because it's too large:

```
Name of the module is: jdk.javadoc
Name of the module is: jdk.deploy
Name of the module is: javafx.graphics
Name of the module is: java.security.jgss
Name of the module is: jdk.editpad
Name of the module is: java.compiler
Name of the module is: jdk.jdeps
Name of the module is: jdk.packager
Name of the module is: java.management.rmi
Name of the module is: javafx.swing
Name of the module is: jdk.attach
Name of the module is: java.desktop
Name of the module is: jdk.unsupported
Name of the module is: javafx.fxml
...
```

■ **Note** You can find the source code for this example in the directory /ch10/layers.

The next example shows how to print information about all the layers that exist in the system.

Describe Layers at Runtime

Listing 10-5 shows the current module layer and its parent layers.

Listing 10-5. Describe the Current Modules Layer and Its Parent Layers

```
package modulelayer;

import java.lang.ModuleLayer;
import java.util.List;

public class LayerUtil {
```

```
    public static void describeCurrentAndParentLayers() {

        // prints the layer information for the current layer
        ModuleLayer thisModuleLayer = LayerUtil.class.getModule().getLayer();
        printLayerInformation(thisModuleLayer);

        // gets all the parents of the layer
        List<ModuleLayer> parentModuleLayerList = thisModuleLayer.parents();

        if(parentModuleLayerList.isEmpty()) {
            System.out.println("This layer has no parent layers");
        }
        else {
            for(ModuleLayer moduleLayer : parentModuleLayerList) {
                printLayerInformation(moduleLayer);
            }
        }
    }

    private static void printLayerInformation(ModuleLayer moduleLayer) {
        System.out.println("The name of the modules in this layer are: " + moduleLayer.
        toString());
        System.out.println("The configuration for this layer: " + moduleLayer.configuration());
    }

    public static void main(String[] args) {
        describeCurrentAndParentLayers();
    }
}
```

First, we print the information for the current layer. We retrieve it by calling the `getLayer()` method on the current module. Afterward, we retrieve the layer's parents by calling the method `parents()` on the current layer. If the resulting list is empty, then we have no parent for our layer. Otherwise, if the resulting list has only one element and this element is the empty layer, then our layer doesn't have any parent layers, so we print a corresponding message. Finally, we iterate over the list of parent layers and print the names of the modules they contain together with the configuration.

■ **Note** The empty layer doesn't consist of any modules. The Java Platform Module System can't load two modules into the same layer when the two modules have the same package name. This is true even if the package is private.

We've covered the basics of layers. It's time to move forward and present the concept of upgradeable modules.

Upgradeable Modules

A module is *upgradeable* if it can be upgraded by deploying it on the upgrade module path. The JPMS performs a check at link-time and runtime to make sure that only the upgradeable modules are allowed to be upgraded.

From the Java SE modules, the only upgradeable modules are the ones from module java.se.ee. The upgradeable modules are java.activation, java.compiler, java.corba, java.transaction, java.xml.bind, java.xml.ws, java.xml.ws.annotation, and jdk.internal.vm.compiler. A couple of standard modules from JDK are upgradeable.

`javac` and `java` provide a command-line option called `--upgrade-module-path` that takes a list of directories. These directories contain modules that replace the existing modules in the runtime image. For example, to upgrade JAXB, we can execute the following command:

```
java --upgrade-module-path myDirectory --add-modules java.xml.bind
```

Here, myDirectory represents a directory that contains the modules that will replace the existing modules.

■ **Note** Non-upgradeable modules can't be upgraded even when using the command-line option `--patch-module`. Non-upgradeable modules are modules linked into a runtime image.

For upgrading a module in the runtime image, a module can be deployed on the upgrade module path. An automatic module can also be deployed on the upgrade module path. But an automatic module can't be upgraded because an upgradeable module is linked into a runtime image, whereas an automatic module isn't linked into a runtime image.

■ **Note** The upgradeable modules replace the old Endorsed Standard Mechanism in JDK 9.

Features Coming in the Next Releases

Some features will be coming in the next releases of JDK. The JCP team announced that the next releases will resolve two issues that we have now and will also add two new features. The new features that will come in the next releases are the following:

- *multi-module JAR files*: Now a modular JAR file can contain only a single module. It's not allowed to contain more than one module. In practice, we can have large JAR files that contain different pieces of functionality that don't pass into a single module. So it would be better to have multiple modules in a single modular JAR file.
- *Additional module-layer operations*: The ModuleLayer.Controller API will be enhanced with new methods, like `addUses()` and `addPackage()`.

The issues that will be solved in the next releases are the following:

- *Concealed package conflicts*: This issue was covered in Chapter 8. The problem is that two distinct modules can't share the same name of a package. The proposal made by the JCP team is to avoid concealed package conflicts by doing a redesign so that modules that contain conflicting packages will be loaded in their own class loaders.
- *Cyclic module relationships*: This issue was also covered in Chapter 8. The proposal for the next JDK releases is to allow cyclic relationships among modules at runtime.

Summary

We started this chapter by looking at the new JMOD files together with the jmod tool. Then we talked about multi-release JAR files, which represent one single JAR file. Therefore, they're only a unit of release and can be used, for example, for replacing JDK internal APIs. If we're using a JDK internal API in Java 8, we can provide a new class with a replacement for it in JDK 9 and place both class files in a multi-release JAR file. Multi-release JAR files were introduced to be able to use a specific version of the JDK, even if some third-party libraries haven't been yet upgraded to the last version. They help third-party libraries to use API features from newer releases of Java.

Further, we explained the new notion of layer in the context of the Java Platform Module System. I explained what layers are, why they're useful, and how we can create our own layers. If an application uses layers, it will probably use only the boot layer. I covered the class loading mechanism in JDK 9 and explained how it fits together with the new module system. We learned that there is no relation between class loaders and modules enforced by Jigsaw.

The chapter concluded by talking about upgradeable modules and about some features that will come in the next JDK releases.

In Chapter 11, you'll learn how to unit test modular applications.

CHAPTER 11

Testing Modular Applications

By now, you should have a deep overview of Project Jigsaw and should be able to start using use it in your projects. But there's one important topic we haven't mentioned yet: unit testing. This chapter focuses on unit testing modular applications in Java 9 and the different approaches to it that you can take. In this chapter, we'll show you some best practices for performing unit testing in Java 9 in the context of a modular application.

Suppose we have a module with classes that need to be tested. If we put the unit test classes in another module, then we need to make the unit tests be able to access types from the module being tested. This puts the strong encapsulation mechanism introduced in JDK 9 in play. A solution would be to add `--add-exports` flags to make the types from the module under test available to the unit tests. But this isn't sufficient because it's also compulsory that the types from the module under test are public. If they're not, then the export of packages doesn't give us the necessary level of accessibility. This is just one of the many challenges that we have to solve before performing unit testing in Java 9.

Performing unit testing in a modular application is necessary in order to achieve the desired level of software quality, and it's even more critical here than when testing non-modular applications. Unit testing in Java 9 is a little more complicated than unit testing in versions prior to Java 9 because in Java 9 we have to assure readability between the Junit test classes and the objects under test. There can be different combinations for locations of the Junit test classes for the classes being tested. They can reside in the same module, in different modules, on the class path, or partially on the module path and on the class path. The next section goes into these scenarios in more detail.

Scenarios for Unit Testing in Java 9

As mentioned earlier, we can have different scenarios for unit testing in JDK 9, depending on the location of the Junit test classes and the objects under test. The following common scenarios may occur during unit testing in JDK 9:

- The Junit test classes and the test objects reside in different modules.
- The Junit test classes don't reside in a module, but the test objects do.
- Both Junit test classes and the test objects reside in the same module.

Each of the three scenarios has to be treated differently and requires a different approach to assure readability between the Junit test cases and the objects under test. The following subsections look at each of them in detail.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_11

Scenario 1: Junit Test Classes and Types Under Test Are in Different Modules

Suppose that the types under test are in module A and the Junit test classes are in module B. This scenario is one of the simplest. Figure 11-1 illustrates it.

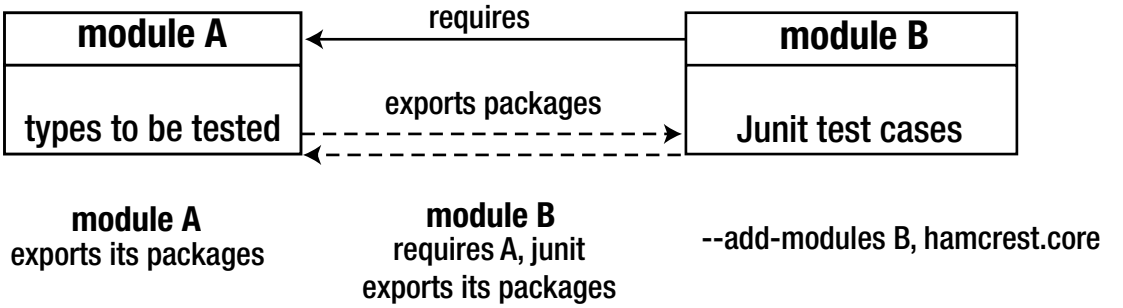

Figure 11-1. *Junit test classes and classes under test are in different modules*

First of all, we need to assure readability between the two modules. Module A should export its packages. Module B should require module A and also export its packages. In this way, module B can access the public types from module A. Module B should further require the junit automatic module in its module declaration because it's making use of it.

To make this scenario work, be aware of one more important thing. When we run the Junit test cases from module B, we need to also add all the existing modules (including the automatic modules) using the `java` launcher `--add-modules` flag. In our case, we need to add the module B and the automatic module hamcrest.core, which is a dependency of Junit. This is basically everything we need to do in order to make this scenario work. Later in this chapter, we'll will show an example using this scenario.

Scenario 2: Only the Types Under Test Reside Inside a Module

This second scenario is the most difficult one. In this scenario, we have the types under test in module A, but the Junit test classes aren't residing in any module—they reside on the class path. For this, we have to use the `javac -Xmodule` and the `java` launcher `--patch-module` command-line options. Both options are described in detail later in this chapter. Figure 11-2 illustrates this scenario.

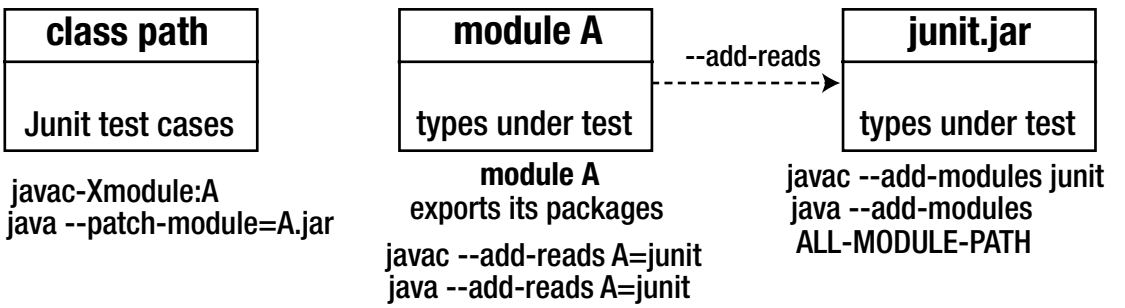

Figure 11-2. *Types under test are inside a module, and Junit test cases are on the class path*

First, we have to compile the Junit test classes as if it were part of module A (using the `javac -Xmodule` option). In this way, we make the Junit test classes part of module A. Second, we have to use the `javac --add-reads` command-line option to add the reading edges to Junit. This is mandatory because now module A has a dependency on Junit. Because module A doesn't read Junit, we have to use the `--add-reads` command-line option to tell it to read Junit. At the same time, we also have to add the junit automatic module using the `javac --add-modules` option.

To make this scenario work, when we're running the Junit test classes, we have to use the `java` launcher `--patch-module` command-line option so that we can patch module A. Therefore, we make the Junit test class part of module A at runtime using the `ALL-MODULE-PATH` constant. Don't worry if it's not clear yet how this scenario works. Later in this chapter, you'll see this scenario at work, and we'll explain the new command-line options used.

Scenario 3: Both Junit Test Classes and Test Under Test Reside in the Same Module

In this scenario, we have implicit readability between the types. The disadvantage in this case is related to the fact that our module needs all the test dependencies. Introducing dependencies on test libraries for every module that requires unit testing is definitely not the best solution. If we have ten modules that need to be tested, then all of them would need to add the test dependency separately.

Now that we know the most common scenarios that can occur when performing unit testing in Java 9, let's move on to the `-Xmodule` command-line option of the Java compiler and the `--patch-module` command-line option, mentioned earlier in the second scenario. They're used to patch modules with classes.

The -Xmodule Option

The Java compiler command-line option `-Xmodule` is used to compile classes for a module. Figure 11-3 describes its syntax.

-Xmodule:<module_name>

Figure 11-3. *Syntax of the javac -Xmodule command-line option*

The `-Xmodule` option specifies that we should compile the classes as if they were part of the module <module_name>. This option is used at compile-time to inject a class into a module. It can't be used at runtime. Using the `-Xmodule` option, we can make classes be part of a specific module. If the module we pass as an argument doesn't exist, an error `module not found` will be thrown:

```
error: module not found: <module_name>
```

■ **Note** It's not possible to list more than one module. You can't specify multiple module names to the `-Xmodule` command-line option.

The --patch-module Option

The JDK 9 specification states: "When testing or debugging it is sometimes useful to replace selected class files or resources of specific modules with alternate or experimental versions, or to provide entirely new class files, resources, and even packages. This can be done via the `--patch-module` option."

The `--patch-module` option is used at both compile-time and runtime to replace the class files of a module with other class-specific class files. It can be used by the Java compiler as well as by the `Java` launcher. The role of the `--patch-module` option is to override classes inside a module. This option replaced the old `-Xbootclasspath/p` option, which has been removed in Java 9.

Figure 11-4 shows the syntax of the `--patch-module` command-line option.

--patch-module <module_name>=<file>(<path_sepatator><>file>)*

Figure 11-4. *Syntax of the --patch-module option*

- `<module_name>` represents the module name.
- `<file>` represents the file system path name of a module definition.
- `<path_separator>` represents the host platform's path-separator character.

The module specified by `<module_name>` is patched with the class files existing inside the directory <file>. We can also specify a normal JAR (not a modular one) instead of the directory containing the class files.

■ **Note** The `--patch-module` can be used to make the test classes part of the module at runtime. The JCP team states that it is "intended only for testing and debugging. Its use in production settings is strongly discouraged."

The `--patch-module` command-line option can also be used to patch automatic modules, but it can't be used to replace module-info.class files, as the JCP team states in the specification: "The `--patch-module` option cannot be used to replace module-info.class files. If a module-info.class file is found in a module definition on a patch path, then a warning will be issued and the file will be ignored." The JCP team also tell us what happens with the packages that aren't exported: "If a package found in a module definition on a patch path is not already exported by that module, then it will, still, not be exported. It can be exported explicitly via either the reflection API or the `--add-exports` option."

Patching a Module

The following example shows how to patch a class inside a module. That means we replace a Java class file inside a module with another one. For doing this, we'll use the `javac` command-line option `-Xmodule` and the `java` command-line option `--patch-module`. We'll patch an existing module using the `--patch-module` option, and you'll see two ways of doing that. Therefore, we create four folders:

- The modules folder
- The modulesLibrary folder
- The patchModules folder
- The patchModulesLibrary folder

We have a module com.apress.moduleA that contains a POJO class called `Employee.java` and another class called `EmployeeImpl.java`, which creates an object of type `Employee` and sets some properties on it. There's also a module com.apress.moduleB that contains the public static void `main(String[] args)` method. This module simply creates an object of type `EmployeeImpl` and then calls some methods on this object. Further, we define another Java class called `EmployeeImpl.java`, having the same package name as the package name of the former `EmployeeImpl.java` class. This new class isn't part of a module. Our intention is to replace the new class with the older one inside the com.apress.moduleA module using the command-line options `-Xmodule` and `--patch-module` described earlier.

Listing 11-1 shows the classes `Employee.java` and `EmployeeImpl.java` of the com.apress.moduleA module. The class `Employee.java` is inside the package com.apress.moduleA.entity.

Listing 11-1. The Classes Employee.java and EmployeeImpl.java from the com.apress.moduleA Module

```
// Employee.java

package com.apress.moduleA.entity;

public class Employee {

    private String firstName;
    private String lastName;
    private String department;

    public Employee() {
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String department) {
        this.department = department;
    }

}
```

```
// EmployeeImpl.java

package com.apress.moduleA;

import com.apress.moduleA.entity.Employee;

public class EmployeeImpl {

    public Employee employee;

    public EmployeeImpl() {
    }

    public Employee createNewEmployee() {
        employee = new Employee();
        return employee;
    }

    public Employee setEmployeeInfo() {
        employee = createNewEmployee();
        employee.setFirstName("John");
        employee.setLastName("Anderson");
        employee.setDepartment("IT");
        return employee;
    }

    public void getEmployeeInfo() {
        System.out.println("Employee first name is: " + employee.getFirstName());
        System.out.println("Employee last name is: " + employee.getLastName());
        System.out.println("Employee department is: " + employee.getDepartment());
    }

}
```

Listing 11-2 shows the module descriptor of module com.apress.moduleA, which exports the package.

Listing 11-2. The module-info.java File of Module com.apress.moduleA

```
module com.apress.moduleA {
  exports com.apress.moduleA;
}
```

Module com.apress.moduleB imports types from module com.apress.moduleA and calls methods on an `EmployeeImpl` object, as shown in Listing 11-3.

Listing 11-3. The MainClass.java File of Module com.apress.moduleB

```
package com.apress.moduleB;

import com.apress.moduleA.*;

public class MainClass {
```

```
    public static void main(String[] args) {
        EmployeeImpl employeeImpl = new EmployeeImpl();
        employeeImpl.createNewEmployee();
        employeeImpl.setEmployeeInfo();
        employeeImpl.getEmployeeInfo();
    }
}
```

Listing 11-4 represents the module descriptor of module com.apress.moduleB.

Listing 11-4. The module-info.java of Module com.apress.moduleB

```
module com.apress.moduleB {
        requires com.apress.moduleA;
}
```

In the new directory com.apress.moduleA2, we define another version of the `EmployeeImpl.java` class, which isn't part of any module. Listing 11-5 shows the new class having the same package name as the old one, com.apress.moduleA.

Listing 11-5. Class EmployeeImpl

```
package com.apress.moduleA;

import com.apress.moduleA.entity.Employee;

public class EmployeeImpl {

    public Employee employee;

    public EmployeeImpl() {
    }

    public Employee createNewEmployee() {
        employee = new Employee();
        return employee;
    }

    public Employee setEmployeeInfo() {
        employee = createNewEmployee();
        employee.setFirstName("Andrew");
        employee.setLastName("Lopez");
        employee.setDepartment("Big Data");
        return employee;
    }

    public void getEmployeeInfo() {
        System.out.println("Employee first name is: " + employee.getFirstName());
        System.out.println("Employee last name is: " + employee.getLastName());
        System.out.println("Employee department is: " + employee.getDepartment());
    }
}
```

We changed only the first name, last name, and department in this last example. Now that we have the code, we begin the process of replacing the `EmployeeImpl.java` from the last listing with the one from module com.apress.moduleA. First, we compile both existing modules but exclude the directory com.apress.moduleA2, because it's not a module. The following command does this, making use of the `grep -v` command in order to exclude the directory com.apress.moduleA2 from the compilation:

```
$ javac -d modules --module-path modulesLibrary --module-source-path src $(find src -name
“*.java” | grep -v com.apress.moduleA2)
```

■ **Note** The role of `grep -v` or `grep -invert-match` is to invert the sense of matching. In our case, it excludes the files from the com.apress.moduleA2 directory and selects the ones that don't match.

As a result, both com.apress.moduleA and com.apress.moduleB modules are compiled, and the class files are now residing in the modules directory. Next, we create modular JAR files for each of the previous compiled modules. For this, we go inside the modules directory and create two modular JAR files for each module. As input, we take the class files from the modules directory:

```
cd modules
$ jar --create --file=../modulesLibrary/com.apress.moduleA.jar -C com.apress.moduleA .
$ jar --create --file=../modulesLibrary/com.apress.moduleB.jar -C com.apress.moduleB .
```

Both modular JAR files were created in the modulesLibrary directory. Further, we attempt to compile the patch as a class. We compile the EmployeeImpl.java file from the package com.apress.moduleA located inside the directory com.apress.moduleA2:

```
cd ..
$ javac -Xmodule:com.apress.moduleA --module-path modules -d patchModules/com.apress.moduleA
src/com.apress.moduleA2/com/apress/moduleA/EmployeeImpl.java
```

Here are the command-line options used:

- `-Xmodule:com.apress.moduleA` specifies that the class `EmployeeImpl.java` should be compiled as if it's actually part of the module com.apress.moduleA.
- `-d patchModules/com.apress.moduleA` specifies the output directory where to compile the class `EmployeeImpl.java`.

We pass the path to the class that will be compiled, `EmployeeImpl.java`. As a result, the EmployeeImpl file from com.apress.moduleA2 directory has been compiled into the patchModules directory. Further, we create a JAR file for the class that we previously patched. In this case, we create a normal JAR, not a modular one. Therefore, we go inside the patchModules directory and type the following:

```
jar --create -file=../patchModulesLibrary/com.apress.moduleA.jar -C com.apress.moduleA .
```

As a result, inside the patchModulesLibrary directory a new JAR called com.apress.moduleA.jar is created.

We can run this example in two different ways: by patching module com.apress.moduleA with classes or by patching it with a JAR.

First, let's run it by patching it with classes. In this example, we make use of the `--patch-module` command-line option to express that we want to patch the module com.apress.moduleA with the class existing inside the patchModules directory. To recap, inside the patchModules directory is our `EmployeeImpl.class`, which corresponds to the new `EmployeeImpl` class that we want to be used to replace the old one:

```
cd ..
$ java --patch-module com.apress.moduleA=patchModules/com.apress.moduleA --module-path
modulesLibrary -m com.apress.moduleB/com.apress.moduleB.MainClass
```

We also passed the `main` class to the `-m` flag. The following output is printed at the console:

```
Employee first name is: Andrew
Employee last name is: Lopez
Employee department is: Big Data
```

As we can see in this output, the new `EmployeeImpl` class replaced the existing one. In the `MainClass` we created an object of type `EmployeeImpl` that represents the new `EmployeeImpl` class.

In the previous example, we specified the location of the .class files to the `--patch-module` option. As an alternative, we can also specify the location of the JAR file we previously created inside the patchModulesLibrary directory. Therefore, we run the `java` launcher and patch the module com.apress.moduleA with the JAR com.apress.moduleA.jar:

```
$ java --patch-module com.apress.moduleA=patchModulesLibrary/com.apress.moduleA.jar
--module-path modulesLibrary -m com.apress.moduleB/com.apress.moduleB.MainClass
```

The result is the same as the one we previously had. In this example, we revealed how we can patch a module using the `--patch-module` command-line option first.

Now that we know how to patch a module, let's see how we can apply this knowledge for running Junit test cases in a modular context.

■ **Note** You can find the source code for this example in the directory /ch11/patchingAModule.

Earlier in this chapter, you saw three scenarios for unit testing in Java 9. Let's look at practical code examples for scenario 1 (Junit test classes and types under test reside in separate modules) and scenario 2 (types under test reside in a module and the Junit test classes are on the class path).

Running a Junit Test Where the Junit Test Class and the Types Under Test Reside in Separate Modules

The following shows a very simple example of running a Junit test in Java 9. In our case, the Junit test and the classes under test reside in different modules. This scenario corresponds to scenario 1 described earlier in this chapter. We modify the previous example (the example that patched a module) in order to make it suitable for testing with Junit.

We add the following method in the `Employee.java` class of module com.apress.moduleA:

```
public String getEmployeeFullData() {
        return getFirstName() + ", " + getLastName() + ", " + getDepartment();
 }
```

We also need to add a Junit test case. It will reside in the module com.apress.moduleB and will simply call the method `getEmployeeFullData()` from the module com.apress.moduleA.

Listing 11-6 shows the class `EmployeeTest`.

Listing 11-6. The EmployeeTest Class

```
package com.apress.moduleB;

import org.junit.Assert;
import org.junit.Test;
import org.junit.Before;

import com.apress.moduleA.entity.Employee;

public class EmployeeTest {

    Employee employee;

    @Before
    public void setEmployeeData() {
        employee = new Employee();
        employee.setFirstName("Alexandru");
        employee.setLastName("Jecan");
        employee.setDepartment("IT");
    }

    @Test
    public void employeeDataTest() {
        Assert.assertEquals("Alexandru, Jecan, IT", employee.getEmployeeFullData());
    }
}
```

This class imports the `Employee` class from module com.apress.moduleA, instantiates an object of type `Employee`, and calls a method on it. Listing 11-7 shows the module descriptor of module com.apress.moduleB.

Listing 11-7. The module-info.java of Module com.apress.moduleB

```
module com.apress.moduleB {
        requires junit;
        requires com.apress.moduleA;
        exports com.apress.moduleB;
}
```

Module com.apress.moduleB requires module com.apress.moduleA, because it uses types from it. It also requires the module junit, because we're using the Junit test framework. In this example, junit is an automatic module.

We define a folder called automaticModules where we put the two necessary JARs needed to run a Junit test: Junit and Hamcrest-core.

■ **Note** The Junit and Hamcrest-core JAR files can be downloaded from the Maven repository at https://mvnrepository.com/. You can download them using your web browser by clicking the Download (JAR) link. You don't need to use Maven for this.

Listing 11-8 shows the content of the automaticModules folder.

Listing 11-8. The automaticModules folder

```
hamcrest-core-1.3jar
junit-4.12.jar
```

We compile both modules com.apress.moduleA and com.apress.moduleB:

```
javac -d modules --module-path "automaticModules;modulesLibrary" --module-source-path src
$(find src -name "*.java")
```

The automaticModules folder, which contains both JAR files described earlier, is passed to the `--module-path` command-line option. In this way, the JAR files become automatic modules. Next, we switch to the modules folder and create two JARs for both modules com.apress.moduleA and com.apress.moduleB:

```
cd modules
jar --create --file=../modulesLibrary/com.apress.moduleA.jar -C com.apress.moduleA .
jar --create --file=../modulesLibrary/com.apress.moduleB.jar -C com.apress.moduleB .
```

We can now run our unit test. For this, we have to pass all the modules that we have on the module path, including the automatic modules. We can use the constant `ALL-MODULE-PATH`. As specified in the JDK documentation, this variable stands for all the modules on the module path. It's much easier to specify this constant instead of specifying each module separately:

```
cd ..
java --module-path "automaticModules;modulesLibrary" --add-modules ALL-MODULE-PATH -m
junit/org.junit.runner.JUnitCore com.apress.moduleB.EmployeeTest
```

The `--module-path` command-line option points to the automaticModules folder, which contains the Hamcrest and Junit JAR files, and to the modulesLibrary folder, which contains the com.apress.moduleA.jar and the com.apress.moduleB.jar files. The `-m` option gets the parameter `junit/org.junit.runner.JUnitCore com.apress.moduleB.EmployeeTest`. In this way, we state that we want to use `JunitCore` from the module junit to run the tests from the class `EmployeeTest`, which is located inside the package com.apress.moduleB.

Instead of the constant `ALL-MODULE-PATH`, it would have also been possible to list the modules we need using comma. In this case, we would have had this:

```
--add-modules com.apress.moduleB,hamcrest.core
```

The result of running this test is OK, which means the test was successful and we've successfully managed to run the test case by accessing the other module and by reading the junit-4.12.jar and the hamcrest-core-1.3.jar files as automatic modules.

■ **Note** You can find the source code for this example in the directory /ch11/junitSeparateModules.

Running a Junit Test Where the Junit Test Class Doesn't Reside Inside a Module

So far, so good. In the previous example, our Junit test was in a module, and the test object was in another module. We managed to connect them and make them work. But what happens when the Junit test isn't part of a module and instead resides on the class path? Things get a little more complicated in this case, which corresponds to scenario 2 from the beginning of the chapter.

In the next example, we have our `EmployeeTest` class, but this isn't part of a module anymore. We also change the package name of the `EmployeeTest` class. Its new name is com.apress.moduleA. We do this because during patching, the package name has to be the same as the one from module com.apress.moduleA.

In this example, we don't have a module com.apress.moduleB anymore, so we delete the module-info.java file from it. Listing 11-9 shows the module descriptor of module com.apress.moduleA.

Listing 11-9. The module-info.java File of Module com.apress.moduleA

```
module com.apress.moduleA {
    exports com.apress.moduleA.entity;
    exports com.apress.moduleA;
}
```

We start by compiling the module com.apress.moduleA, and afterwards we create a modular JAR out of it:

```
javac -d modules --module-path "automaticModules;modulesLibrary" --module-source-path src
$(find src -name "*.java" | grep -v com.apress.moduleB)
cd modules
jar --create --file=../modulesLibrary/com.apress.moduleA.jar -C com.apress.moduleA .
```

Next, we compile the EmployeeTest.java file

```
cd ..
javac -d patchModules/com.apress.moduleA -Xmodule:com.apress.moduleA --add-reads com.apress.
moduleA=junit --add-modules junit --module-path "modulesLibrary;automaticModules" src/com.
apress.moduleB/com/apress/moduleA/EmployeeTest.java
```

During compilation, we used the option `-Xmodule:com.apress.moduleA` in order to compile the Java class `EmployeeTest` as if it were part of module com.apress.moduleA. The command-line option `--add-reads com.apress.moduleA=junit` is mandatory because module com.apress.moduleA uses Junit, so as a result a read dependency to Junit is required by the module com.apress.moduleA. The option `--add-modules junit` is also mandatory. We're making use of Junit and therefore we add the automatic module junit, which is automatically created after placing the junit.jar on the module path.

After compiling the statement just mentioned, the patchModules folder will contain the EmployeeTest.class file. Further, we create a JAR containing this class file:

```
cd patchModules
jar --create --file=../patchModulesLibrary/com.apress.moduleA.jar -C com.apress.moduleA .
```

We run the Junit test by patching the module com.apress.moduleA:

```
cd ..
java --patch-module com.apress.moduleA=patchModules/com.apress.moduleA --module-path
"automaticModules;modulesLibrary" --add-reads com.apress.moduleA=junit --add-modules
ALL-MODULE-PATH -m junit/org.junit.runner.JUnitCore com.apress.moduleA.EmployeeTest
```

Here we patched the module using the EmployeeTest.class file located in the patchModules folder. We also added the reads dependency on Junit and added all the modules from the module path using the constant `ALL-MODULE-PATH`, which of course include the automatic modules. In this way, we were able to successfully run a Junit test located outside of a module that uses test objects from a module.

We could also use the JAR file com.apress.moduleA.jar for patching the module com.apress.moduleA (instead of the class files as in the previous example). This `java` command gives the same result as the previous example:

```
$ java --patch-module com.apress.moduleA=patchModulesLibrary/com.apress.moduleA.jar
--module-path "automaticModules;modulesLibrary" --add-reads com.apress.moduleA=junit --add-
modules ALL-MODULE-PATH -m junit/org.junit.runner.JUnitCore com.apress.moduleA.EmployeeTest
```

■ **Note** You can find the source code for this example in the directory /ch11/junitTestNotInModule.

Testing with Maven

Maven and other build automation tools make our lives much easier because we don't have to write so many command-line flags. We don't have to write `-Xmodule` and `--patch-module` when running tests using Maven because Maven does this for us in the background.

■ **Note** The Maven Compiler plugin starting from version 3.6.0 has Jigsaw support.

It's much easier to compile our application and run our tests with Maven than it is by writing our entire commands with flags on the command-line. The Jigsaw support built into Maven helps us reduce the amount of work considerably.

We take our last example (from scenario 2), which has the Junit test class outside the module. We have to modify the structure of our project a little in order to make it work with Maven. Therefore, the module com.apress.moduleA has to be located under the src/main/java directory. Also, the EmployeeTest.java will be located under the src/test directory.

A pom.xml file is the essential unit of work in Maven. It contains information about the project and the configuration stuff used to build the project. Listing 11-10 shows the pom.xml file that we add in the root of our project.

Listing 11-10. The pom.xml file

```
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://maven.apache.
org/POM/4.0.0"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
         maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.apress.junit</groupId>
    <artifactId>junit-testing</artifactId>
    <version>0.0.1</version>

    <properties>
        <maven.compiler.source>9</maven.compiler.source>
        <maven.compiler.target>9</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>${project.build.directory}/lib</outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Using the `<maven.compiler.source>` and `<maven.compiler.target>` tags, we set the source and target to 9 because we want to use JDK 9 for compiling our project. We use the Maven Compiler plugin version 3.6.1 which has Jigsaw support. We also use the Maven Dependency plugin to copy the dependencies inside the target/lib directory. We can now run `mvn clean package` to build the project. By building the project, our tests are also executed. We get a `BUILD SUCCESS` message, so the tests were successful.

In the newly created target directory, we have a classes folder that contains our class files (including the module-info.class file) that were compiled using the Maven Compiler plugin. Inside the lib folder we can find the test dependencies hamcrest-core-1.3.jar and junit-4.12.jar.

Using Maven, we didn't have to use the `-Xmodule` flag or the `--add-modules` flag because everything was done in the background by the Maven Compiler plugin.

■ **Note** You can find the source code for this example in the directory /ch11/junitTestNotInModuleMaven.

Summary

This chapter discussed the most important aspects of unit testing modular applications in Java 9. We learned about the `-Xmodule` Java compiler command-line option used to compile classes for a module. We also learned about the `--patch-module` command-line option used to override classes inside a module. Using these handy flags, we showed how to test an application where the types under test reside inside a module, but the Junit test classes don't. We also looked at an example where the types under test and the Junit test classes reside in different modules. At the end of the chapter, we demonstrated how to use Maven with its Maven Compiler plugin to compile and run unit tests in JDK 9.

Chapter 12 covers the integration of Jigsaw with integrated development environments (IDEs) like Intellij IDEA and Eclipse. It also talks about how Jigsaw works together with build tools like Maven.

CHAPTER 12

Integration with Tools

In order for Java 9 to be adopted easily and as quickly as possible by the developer community, it's very important for the integrated development environments (IDEs) and build tools to offer extensive support for Java 9 as much as possible—so important that we dedicate this entire chapter to this topic.

This last chapter shows how JDK 9 in general and the Java Platform Module System in particular integrates with the following:

- IDEs like Intellij IDE, Eclipse, and NetBeans
- Build tools like Apache Maven

We'll discover what kinds of support these tools offer for Jigsaw.

Integration with IDEs

Jigsaw is already integrated in IDEs like Intellij IDEA, Eclipse, and Netbeans. The next three subsections cover what kind of support these IDEs offer to make the work with Jigsaw easier for developers. We'll start with Intellij IDEA.

Note Intellij IDEA, Eclipse, and NetBeans are among the most popular IDEs for Java programming according to an article published on July 2017, at www.keycdn.com/blog/best-ide/. So we decided to focus on these three IDEs in this chapter instead of other IDEs that don't especially focus on the Java programming language.

Integration with Intellij IDEA

Intellij IDEA, developed by JetBrains, is a Java IDE that has both a community edition and a commercial edition. It offers support for Project Jigsaw starting from version 2017.1, released in March 2017. Among its many features, IDEA supports code completion inside the module-info.java module descriptor file.

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8_12

Figure 12-1 shows how we can create a module-info.java file in Intellij IDEA by selecting New ➤ module-info.java.

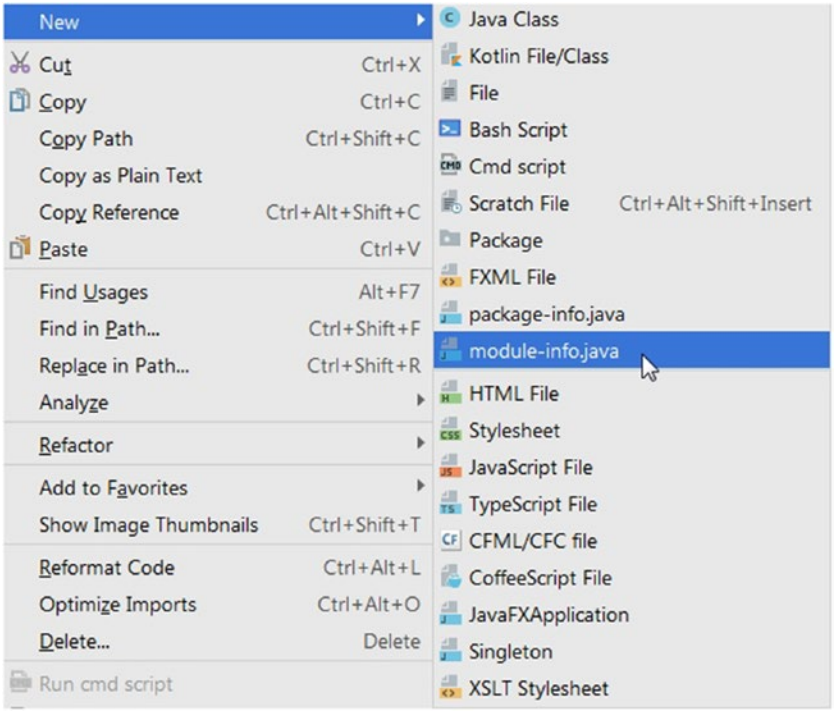

Figure 12-1. *Add a module-info.java in Intellij IDEA*

IDEA creates an empty module-info.java file that contains only the module keyword and a name for the module.

If we add a new import into a java file, Intellij IDEA can automatically add the necessary `requires` clause inside the module-info.java. For example, if we import the `java.sql.DriverManager` class in our code, Intellij IDEA can find out the name of the module where this class resides. As a result, it can indicate to us to add the `requires java.sql` clause inside the module descriptor, as illustrated in Figure 12-2.

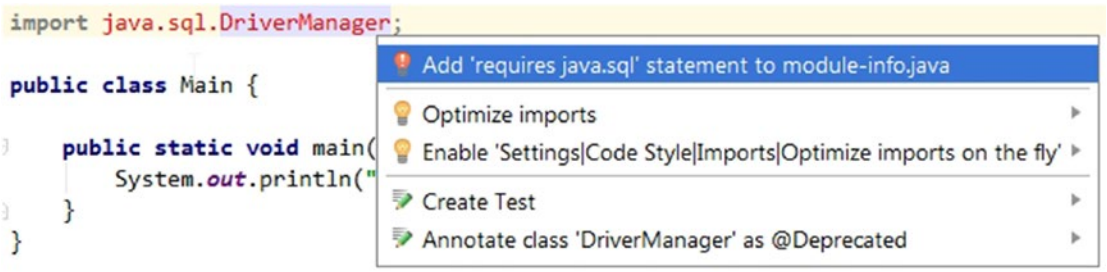

Figure 12-2. *Autocomplete function for adding a requires statement inside the module-info.java*

Intellij IDEA also provides code autocomplete functionality inside the module-info.java file. If we start to type the name of a module, IDEA will be compute and show the available suggestions, as shown in Figure 12-3.

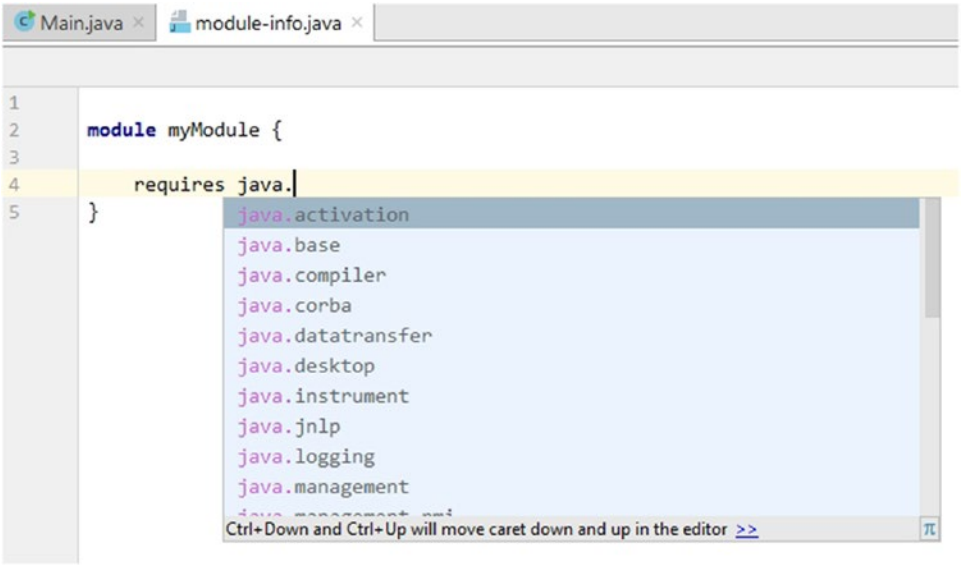

Figure 12-3. *Autocomplete for the module names*

Intellij IDEA can also provide autocomplete functionality for the packages we want to export. Figure 12-4 shows an example of providing the autocomplete feature for the `exports` clause. When we start to type the name of the package we want to export, Intellij IDEA can indicate likely suggestions so we don't have to type the entire name.

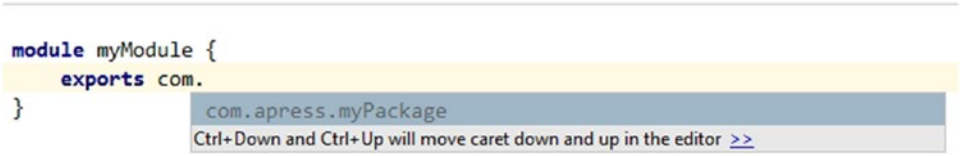

Figure 12-4. *Autocomplete function for the names of the packages inside the module-info.java file*

Among other features related to Jigsaw that Intellij IDEA offers, we want to mention these:

- *Visualizing module diagrams*: Module diagrams allow us to visualize the dependencies between our modules. These can be visualized by selecting Diagrams ➤ Show Diagram ➤ Java Modules Diagram.
- *Visualizing module usages*: Shows where a module is used.

Intellij IDEA provides many other features not covered in this chapter. Covering all the features is beyond the scope of this book. For more on Jigsaw support in Intellij IDEA, check out the documentation on the official JetBrains blog, at https://blog.jetbrains.com/idea/?s=java+9. Search for the keywords *java 9* or *module*.

The next section explores another popular IDE: Eclipse.

Integration with Eclipse

Eclipse is a free IDE. As of JDK 9 build 178 (July 2017), Eclipse offers a useful tool called Java 9 Support for Oxygen that works only with Eclipse Oxygen (4.7).

However, it's possible to start every version of Eclipse with JDK 9, and there are two possibilities for doing that. The first is to have JDK 9 on the system path, and the second is to add the path to JDK 9 in the eclipse.ini file, as in the following example:

```
--launcher.appendVmargs
-vm
C:\Program Files\Java\jdk-9\bin\javaw.exe
```

Eclipse can be started using JDK if you're using a version greater or equal to Eclipse 4.7. If you're using a version prior to Eclipse 4.7, you have to add the flag `--add-modules=ALL-SYSTEM` inside the eclipse.ini file. This flag has been added in eclipse.ini in Eclipse 4.7, so you don't have to add it anymore if you're using Eclipse 4.7 or higher. The `ALL-SYSTEM` flag is used because not all the types that Eclipse uses reside inside the java.base module.

As for the Java 9 Support for Oxygen tool, the Eclipse documentation states: "Eclipse Java 9 Support contains the following: ability to add JRE and JDK 9 as installed JRE, support for JavaSE-9 execution environment, ability to create Java and Plug-in projects that use a JRE or JDK 9, ability to compile modules that are part of a Java project."

The support of Java 9 for Eclipse is still work in progress as of August 2017. For more information on Jigsaw support in Eclipse, check the documentation on the official Eclipse Wiki at https://wiki.eclipse.org/Java_9_Readiness.

Integration with NetBeans

Netbeans is a cross-platform IDE developed by Oracle. It offers support for JDK 9 starting from Netbeans version 9. As of August 2017, NetBeans lets us create only one single module inside a NetBeans project—we can't create more than one module into a single NetBeans project. If we have more than one Jigsaw module, we have to create a separate NetBeans project for each module.

NetBeans 9 is still under development as of August 2017. You can download it from http://bits.netbeans.org/download/trunk/nightly/latest/.

If we have only JDK 9 installed on our system, then it's fine for NetBeans 9, but if we have JDK 9 *and* another JDK version <9 installed on our system, we have to explicitly specify during the installation of NetBeans that we want to use JDK 9.

According to the official NetBeans website, here are the most important areas where NetBeans JDK 9 offers support:

- Maven projects
- module-info.java support
- Compilation
- Run and debug
- Module dependency graph

■ **Note** For more information on JDK 9 integration with NetBeans, visit http://wiki.netbeans.org/JDK9Support.

You just got an overview of the support that three of the most popular Java-related IDEs give for Project Jigsaw. The next section talks about Jigsaw integration with build tools like Apache Maven.

Integration with Build Tools

Apache Maven has provided very good integration for Jigsaw since the first half of 2016. It started integrating Jigsaw very early and collected valuable feedback from the developer community. The Maven team also developed a great Apache Maven JDeps plugin for running JDeps from Maven.

Integration with Apache Maven

One of the primary goals of Maven was to upgrade only its plugins in order to offer support for Java 9. No changes were necessary inside the Maven Core for making Maven run on Java 9. Another primary goal was to offer support for Java 9 starting with Maven 3.0.

In order to use Maven with Java 9, two conditions have to be met simultaneously:

- The `JAVA_HOME` variable for Maven has to be set to point to a JDK 9 installation.
- The `source` and `target` of the Maven Compiler plugin should be greater or equal than 6.

The Maven Compiler plugin defines a parameter for `source` and a parameter for `target` that correspond to the version of the JDK. The minimum supported version for the `source` and `target` for JDK 9 is 6. The version of JDK used to run Maven doesn't necessarily have to be the same as the version of JDK used to run the Maven Compiler plugin.

Maven needed adjustments for some of the JEPs implemented in Java 9. Besides the JEPs related to Jigsaw, Maven also needed adjustments in order to fit the following JEPs: JEP 223 – New Version-String scheme, JEP 226 – UTF-8 Property Files, JEP 238 – Multi-Release JAR files, JEP 247 – Compile for Older Platform Versions, and JEP 285 – Modular Java Application Packaging.

Even if it's not part of Jigsaw, we should say something about the JEP 223 – New Version-String scheme because it has huge impact on Maven. Maven relies heavily on the system properties. Since the version string has changed in Java 9, Maven throws an `ArrayIndexOutOfBoundsException` as it internally tries to compute the version. Fortunately, the issue has been fixed starting with the following versions of the following plugins:

- maven-archiver-3.0.1
- maven-jar-plugin-3.0.0
- maven-war-plugin-3.0.0
- maven-ear-plugin-xxx
- maven-javadoc-plugin-2.10.4

If you're using these plugins in Java 9, make sure to upgrade them to at least one of these versions.

Table 12-1 shows the Maven plugins affected by the introduction of Java 9.

Remember the JDeps tool described in Chapter 8? Maven integrates this tool into a new plugin, the Apache Maven JDeps plugin.

Table 12-1. *Maven Plugins Affected by Java 9*

Plugin Name	Minimum Compatible Version	Affected Goal and Status
Maven Compiler plugin	3.6.1	**compile** => new feature **testCompile** => new feature
Maven Javadoc plugin	2.10.4	**jar** => failure **javadoc** => warning **aggregate** => failure
Maven Plugin plugin	3.5	**descriptor**
Maven War plugin		**war** => failure
Plexus :: Component Metadata	1.7	**generate-metadata** => new feature

Apache Maven JDeps Plugin

This plugin makes use of the JDeps tool to analyze internal API calls inside our classes. It can perform analysis when building a project.

■ **Note** The first version of the Maven JDeps Plugin is 3.0. This version has been chosen deliberately by Maven to reveal that Maven 3.0 or greater should be used.

The plugin consists of two goals:

- A goal called `jdeps:jdkinternals` that verifies whether the main classes depend on internal JDK classes
- A goal called `jdeps:test-jdkinternals` that verifies whether the test classes depend on internal JDK classes

Table 12-2 shows some of the most important options that can be used inside the `<configuration>` tag of the Maven JDeps plugin, as recorded in the Oracle documentation for Java SE.

Table 12-2. *Options for the Maven JDeps Plugin*

Plugin Name	Description	Example

`failOnWarning`	Specifies whether the build continues if there are JDeps specific warnings. Default is true	`<failOnWarning>`
		`  false`
		`</failOnWarning>`

`dependenciesToAnalyzeIncludes`	Specifies additional dependencies to be analyzed. The format is `<include>`
	`  groupId:artifactId`
	`</include>`.
	Patterns are allowed.
		`<dependenciesToAnalyzeIncludes>`
		`  <include>*:*</include>`
		`  <include>`
		`  com.apress.*:*`
		`  </include>`
		`  <include>`
		`  com.apress.book:*`
		`  </include>`
		`  <include>`
		`  com.apress.book:utils`
		`  </include>`
		`</dependenciesToAnalyzeIncludes>`

`dependenciesToAnalyzeExcludes`	Specifies dependencies that shouldn't be analyzed. The format is `<exclude>`
	`  groupId:artifactId`
	`</exclude>`.
	Patterns are allowed.
		`<dependenciesToAnalyzeExcludes>`
		`  <exclude>`
		`  com.apress.book:*`
		`  </exclude>`
		`</dependenciesToAnalyzeExcludes>`

`jdeps.include`	Restricts analysis to classes that are matching the pattern. It filters the list of classes to be analyzed.	
`jdeps.profile`	Shows profile or the file containing a package.	
`jdeps.recursive`	Traverses all dependencies recursively.	
`jdeps.module`	Shows the module containing the package.	

Running with the option `-R` results in warnings being displayed if there are transitive dependencies that are making use of JDK internal APIs.

■ **Note** By setting the `<failOnWarning>` option to true, the build will immediately fail if there are any warnings.

If we have an application that uses third-party libraries, it would make sense first to find the JDK internal APIs inside our application code using the Maven JDeps plugin with option `<failOnWarning>` set to true so that the build fails if any JDK internal APIs are found. In the next step we could run the Maven JDeps plugin only on our third-party libraries, but this time setting `<failOnWarning>` to false so that our build doesn't fail if the third-party libraries are using JDK internal APIs. This is reasonable because we can't dig inside the third-party libraries to fix them, but we can do this inside our own application code.

Listing 12-1 shows an example of using the Maven JDeps plugin to implement this specific use case.

Listing 12-1. Example Using the Maven JDeps Plugin

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jdeps-plugin</artifactId>
    <version>3.0.0</version>
       <executions>
          <execution>
             <id>testOnClasses</id>
             <goals>
                <goal>jdkinternals</goal>
                <goal>test-jdkinternals</goal>
             </goals>
          </execution>
          <execution>
             <id>testOnDependencies</id>
             <goals>
                <goal>jdkinternals</goal>
                <goal>test-jdkinternals</goal>
             </goals>
             <configuration>
               <failOnWarning>false</failOnWarning>
               <recursive>true</recursive>
             </configuration>
          </execution>
       </executions>
</plugin>
```

We search for JDK internal APIs inside our application code in the execution block which we named `testOnClasses`. We specified both goals, `jdkinternals` and `test-jdkinternals`, so that both `main` and `test` classes are verified. We didn't specify the `<failOnWarning>` attribute here, so it will default to true. Afterward, we specify another execution block to search our third-party libraries that are attached to our application. For this, we specify the `<recursive>` tag as true. `failOnWarning` is set to false so the build doesn't fail in case we find a JDK internal API.

The next section explores the support that the Maven Compiler plugin offers for Jigsaw.

Apache Maven Compiler Plugin

The Apache Maven Compiler plugin offers support for the new Java Platform Module System starting with version 3.6.0, released in October 2016.

The version of the Apache Maven Compiler plugin can be specified directly in the plugin's configuration:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.6.0</version>
</plugin>
```

Version 3.6.0 of the Apache Maven Compiler plugin added support for the module path. As we already know, the Maven Compiler plugin has two goals: `compile` and `test-compile`. During the compile phase, when a module-info.java file is found, the plugin will automatically switch to the module path. During the `test-compile` phase, the plugin will switch to the module path for the main sources and to the class path for the test sources.

■ **Note** The Maven team also added support for specifying flags like `--add-modules` or `--add-exports` directly inside the pom.xml configuration.

For instance, if we want to use the `--add-modules` flag to add the module java.xml.bind using the Maven Compiler plugin, we could define this inside the configuration of the Maven Compiler plugin:

```
<compilerArgs>
    <arg>--add-modules</arg>
    <arg>java.xml.bind</arg>
</compilerArgs>
```

We could also use Maven to make the sun.net package from module java.base available to our module com.apress.myModule:

```
<compilerArgs>
    <arg>--add-exports</arg>
    <arg>java.base/sun.net=com.apress.myModule</arg>
</compilerArgs>
```

Listing 12-2 shows the entire configuration of the plugin for the two use cases mentioned earlier:

Listing 12-2. Adding Compiler Arguments to the Maven Compiler Plugin

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.6.0</version>
    <executions>
      <execution>
        <id>example</id>
        <goals>
           <goal>compile</goal>
        </goals>
        <configuration>
           <compilerArgs>
              <arg>--add-exports</arg>
              <arg>java.base/sun.net=com.apress.myModule</arg>

              <arg>--add-modules</arg>
              <arg>java.xml.bind</arg>
           </compilerArgs>
        </configuration>
      </execution>
    </executions>
</plugin>
```

In this example, we defined the version of the maven-compiler-plugin to be 3.6.0. Inside the `<compilerArgs>` XML tag, we specified arguments that we want to be passed to the compiler. Each argument is specified inside the `<arg>` XML tag.

Backward Compatibility

During migration to Java 9, projects written in a Java version <8 will get a module-info.java file. With Maven, it's possible to compile these projects in Java 9 (taking the module-info.java file into account) or compile them in versions prior to Java 9.

For this, two compilations have to be made:

- The first compilation will be executed by the Maven Compiler plugin using the configuration `<release>9</release>`.
- The second compilation will be executed by the Maven Compiler plugin using a configuration lower than 9, for instance: `<source>1.8</source>` and `<target>8</target>`.

This can be easily executed by the Maven Compiler in two different execution blocks. If we want to be compatible with a version prior to JDK 6, we have to use different JDKs. That's because JDK 9 doesn't support compilation for versions prior to JDK 6.

The Maven Compiler plugin also offers support for JEP 247 – Compile for Older Platform Versions. It allows us to add a module-info.java file for Java 9 projects and also be compatible with earlier versions of Java. For this, we need to call `javac` twice. First we need to call `javac` with `release=9` in order to compile the module-info.java file with JDK 9. Then we need to set the `source` and `target` to a lower version of Java in order to compile the rest of the source code with a lower Java version. If we're using at least Maven version 3.3.1, we can use toolchains for achieving this use case.

For instance, if our `JAVA_HOME` environment variable is lower than or equal to JDK 9, we can set the version of the `jdkToolchain` to 9 in order to compile everything, including the module-info.java file:

```
<configuration>
    <jdkToolchain>
        <version>9</version>
    </jdkToolchain>
    <release>9</release>
</configuration>
```

Subsequently, we recompile the files and exclude the module-info.java file:

```
<configuration>
    <excludes>
        <exclude>module-info.java</exclude>
    </excludes>
</configuration>
```

On the other side, if our `JAVA_HOME` environment variable is equal to JDK 9, then we could set the version of the `<jdkToolchain>` to `[1.5,9)` and compile the files with `<source>` and `<target>` = 1.5:

```
<jdkToolchain>
    <version>[1.5,9)</version>
</jdkToolchain>
    <source>1.5</source>
    <target>1.5</target>
```

■ **Note** As a rule, we should compile with the matching JDK version.

To configure the Maven Toolchain plugin, we could edit the toolchains.xml file in the .m2 folder or, if we're using a version of Maven greater of equal to 3.3.1, we could directly edit the toolchains.xml file inside the Maven conf directory.

Maven also defines a new command-line option `--release`, which lets us pass the version of the JDK release that we want to compile with. For example, the option `--release 8` is equivalent to `-source 8 -target 8 -bootclasspath ...`. The Maven Compiler plugin starting with version 3.6.0 specifies the release version like this:

```
<release>release_version</release>
```

The `<release>` tag configuration has greater precedence over the `<source>` and `<target>` tags. As a result, if we specify the `<release>` tag as well as the `<source>` and `<target>` tags, then the `<release>` tag will be taken into consideration.

■ **Note** Version 3.6.1 of the Apache Maven Compiler plugin was introduced in January 2017.

We would also like to recommend an interesting article written by Robert Scholte, the chairman of the Maven project, which explains why Maven is unable to automatically generate the module-info.java file. You can read it at www.sitepoint.com/maven-cannot-generate-module-declaration/.

Summary

In this chapter we saw what kind of support IDEs and build tools provide for Jigsaw. We started by briefly talking about three IDEs: Intellij IDEA, Eclipse, and NetBeans.

Then we switched to build tools and looked at the support Maven provides for JDK 9. We talked about backward compatibility with Maven and learned how the Java Compiler is called twice—once because the module-info.java file has to be compiled with `--release 9`, and once again so that all the Java sources except module-info.java are compiled with `source` and target less than 9. We also learned about the Maven JDeps plugin, which is used to find usages of JDK internal APIs throughout our code.

Index

© Alexandru Jecan 2017
A. Jecan, *Java 9 Modularity Revealed*, DOI 10.1007/978-1-4842-2713-8

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U, V, W

X, Y, Z