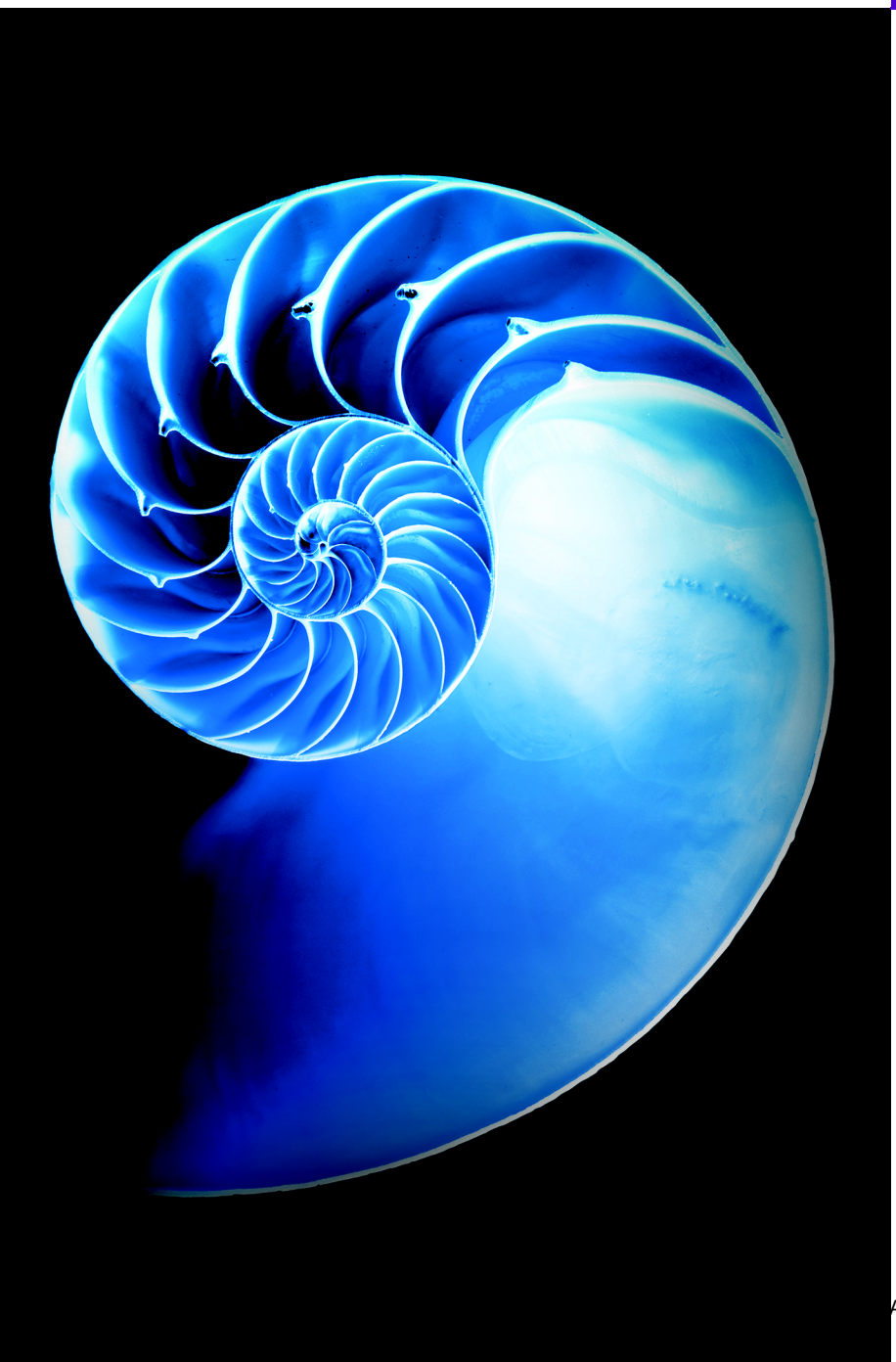
36

Java Platform Module System



Objectives

In this chapter you'll:

- Understand the motivation for modularity.
- Review the Java Platform Module System JEPs and JSRs.
- Create `module` declarations that specify module dependencies with `requires` and specify which packages a module makes available to other modules with `exports`.
- Allow runtime reflection of types with `open` and `opens`.
- Use services to loosely couple system components to make large-scale systems easier to develop and maintain.
- Indicate that a module uses a service or provides a service implementation with the `uses` and `provides...with` directives, respectively.
- Use the `jdeps` command to determine a module's dependencies.
- Migrate non-modularized code to Java 9 with unnamed and automatic modules.
- Use the NetBeans IDE to create module graphs.
- See how the runtime determines dependencies with the module resolver.
- Use `jlink` to create smaller runtimes appropriate for resource-constrained devices.

Outline

36.1 Introduction[1]

In this chapter, we introduce the **Java Platform Module System (JPMS)**—Java 9's most important new technology. Modularity—the result of **Project Jigsaw**[2]—helps developers at all levels be more productive as they build, maintain and evolve software systems, especially large systems. College students in upper-level programming courses will want to master modularity for career preparation.

1. We'd like to thank Brian Goetz, Alex Buckley, Alan Bateman, Lance Anderson, Mandy Chung and Paul Bakker for answering our questions and sharing insights.
2. "Project Jigsaw." `http://openjdk.java.net/projects/jigsaw/`.

Software Required

Before reading this chapter, install JDK 9 and the chapter's source-code examples as described in the Before You Begin section that follows the Preface. We'll present several module-dependency graphs that were created with an early access version of the NetBeans IDE that includes JDK 9 support:

```
http://wiki.netbeans.org/JDK9Support
```

Other IDE vendors will likely provide similar tools.

What is a Module?

Modularity adds a higher level of aggregation above packages. The key new language element is the **module**—a uniquely named, reusable group of related packages, as well as resources (like images and XML files) and a **module descriptor** specifying:

- the module's *name*,
- the module's *dependencies* (that is, other modules this module depends on),
- the packages it *explicitly* makes available to other modules (all other packages in the module are *implicitly* unavailable to other modules),
- the *services it offers*,
- the *services it consumes*, and
- to what other modules it allows *reflection*.

History

The Java SE Platform has been around since 1995. There are now approximately 10 million developers using it to build everything from small apps for resource-constrained devices—like those in the Internet of Things (IoT) and other embedded devices—to large-scale business-critical and mission-critical systems. There are massive amounts of legacy code out there, but until now, the Java platform has primarily been a monolithic one-size-fits-all solution. Over the years there have been various efforts geared to modularizing Java, but none is widely used.

Modularizing the Java SE Platform has been challenging to implement and the effort has taken many years. *JSR 277: Java Module System* was originally proposed in 2005[3] for Java 7. This JSR was later superseded by *JSR 376: Java Platform Module System* and targeted for Java 8. The Java SE Platform is now modularized in Java 9, but only after Java 9 was delayed until July 2017.

Goals

According to JSR 376, the key goals of modularizing the Java SE Platform are:[4]

- Reliable configuration—Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner that's recognized both at compile time and execution time. The system can walk through these dependencies to determine the subset of all modules required to support your app.

3. "JSR 277: Java Module System." `https://jcp.org/en/jsr/detail?id=277`.
4. "JSR 376: Java Platform Module System." `https://jcp.org/en/jsr/detail?id=376`.

- Strong encapsulation—The packages in a module are accessible to other modules only if the module explicitly "`exports`" them. Even then, another module cannot use those packages unless it explicitly states that it "requires" the other module's capabilities. This improves platform security because fewer classes are accessible to potential attackers. You may find that considering modularity helps you come up with cleaner, more logical designs.
- Scalable Java Platform—Previously the Java Platform was a monolith consisting of a massive numbers of packages, making it challenging to develop, maintain and evolve. It couldn't be easily subsetted. The platform is now modularized into 95 modules (this number will change as Java evolves). You can create custom runtimes consisting of only modules you need for your apps or the devices you're targeting. For example, if a device does not support GUIs, you could create a runtime that does not include the GUI modules, significantly reducing the runtime's size.
- Greater platform integrity—Before Java 9, it was possible to use many classes in the platform that were not meant for use by an app's classes. With strong encapsulation, these internal APIs are truly encapsulated and hidden from apps using the platform. One downside of this is that it can make migrating your legacy code to Java 9 problematic.
- Improved performance—The JVM uses various optimization techniques to improve application performance. JSR 376[5] indicates that these techniques are more effective when it's known in advance that required types are located only in specific modules.

Listing the JDK's Modules

A crucial aspect of Java 9 is dividing the JDK into modules to support various configurations (JEP 200[6]). Using the `java` command from the JDK's `bin` folder with the **`--list-modules`** option, as in:

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

lists the JDK's set of modules (Fig. 36.1), which includes the **standard modules** that implement the Java SE Specification (names starting with `java`), JavaFX modules (names starting with `javafx`), JDK-specific modules (names starting with `jdk`) and Oracle-specific modules (names starting with `oracle`). Each module name is followed by a *version string*. In this case, we used a JDK 9 early access version, so each module is followed by the version string `"@9-ea"`, indicating that it's a Java 9 early access (`"ea"`) module. The `"-ea"` will be removed when Java 9 is released.

5. Reinhold, Mark. "JSR 376: Java Platform Module System." `https://jcp.org/en/jsr/detail?id=376`.
6. Reinhold, Mark. "JEP 200: The Modular JDK." `http://openjdk.java.net/jeps/200`.

```
java.activation@9-ea
java.base@9-ea
java.compiler@9-ea
java.corba@9-ea
java.datatransfer@9-ea
java.desktop@9-ea
java.instrument@9-ea
java.jnlp@9-ea
java.logging@9-ea
java.management@9-ea
java.management.rmi@9-ea
java.naming@9-ea
java.prefs@9-ea
java.rmi@9-ea
java.scripting@9-ea
java.se@9-ea
java.se.ee@9-ea
java.security.jgss@9-ea
java.security.sasl@9-ea
java.smartcardio@9-ea
java.sql@9-ea
java.sql.rowset@9-ea
java.transaction@9-ea
java.xml@9-ea
java.xml.bind@9-ea
java.xml.crypto@9-ea
java.xml.ws@9-ea
java.xml.ws.annotation@9-ea
javafx.base@9-ea
javafx.controls@9-ea
javafx.deploy@9-ea
javafx.fxml@9-ea
javafx.graphics@9-ea
javafx.media@9-ea
javafx.swing@9-ea
javafx.web@9-ea
jdk.accessibility@9-ea
jdk.attach@9-ea
jdk.charsets@9-ea
jdk.compiler@9-ea
jdk.crypto.cryptoki@9-ea
jdk.crypto.ec@9-ea
jdk.crypto.mscapi@9-ea
jdk.deploy@9-ea
jdk.deploy.controlpanel@9-ea
jdk.dynalink@9-ea
jdk.editpad@9-ea
jdk.hotspot.agent@9-ea
jdk.httpserver@9-ea
jdk.incubator.httpclient@9-ea
jdk.internal.ed@9-ea
jdk.internal.jvmstat@9-ea
jdk.internal.le@9-ea
jdk.internal.opt@9-ea
jdk.internal.vm.ci@9-ea
jdk.jartool@9-ea
jdk.javadoc@9-ea
jdk.javaws@9-ea
jdk.jcmd@9-ea
jdk.jconsole@9-ea
jdk.jdeps@9-ea
jdk.jdi@9-ea
jdk.jdwp.agent@9-ea
jdk.jfr@9-ea
jdk.jlink@9-ea
jdk.jshell@9-ea
jdk.jsobject@9-ea
jdk.jstatd@9-ea
jdk.localedata@9-ea
jdk.management@9-ea
jdk.management.agent@9-ea
jdk.naming.dns@9-ea
jdk.naming.rmi@9-ea
jdk.net@9-ea
jdk.pack@9-ea
jdk.packager@9-ea
jdk.packager.services@9-ea
jdk.plugin@9-ea
jdk.plugin.dom@9-ea
jdk.plugin.server@9-ea
jdk.policytool@9-ea
jdk.rmic@9-ea
jdk.scripting.nashorn@9-ea
jdk.scripting.nashorn.shell@9-ea
jdk.sctp@9-ea
jdk.security.auth@9-ea
jdk.security.jgss@9-ea
jdk.snmp@9-ea
jdk.unsupported@9-ea
jdk.xml.bind@9-ea
jdk.xml.dom@9-ea
jdk.xml.ws@9-ea
jdk.zipfs@9-ea
oracle.desktop@9-ea
oracle.net@9-ea
```

Fig. 36.1 | Output of `java --list-modules` showing the JDK's 95 modules.

JEPs and JSRs of Java Modularity

We discussed what JEPs and JSRs are in the Preface. The Java modularity JEPs and JSRs are shown in Fig. 36.2. We cite these throughout the chapter.

Java Modularity JEPs and JSRs
JEP 200: The Modular JDK (`http://openjdk.java.net/jeps/200`)
JEP 201: Modular Source Code (`http://openjdk.java.net/jeps/201`)
JEP 220: Modular Run-Time Images (`http://openjdk.java.net/jeps/220`)
JEP 260: Encapsulate Most Internal APIs (`http://openjdk.java.net/jeps/260`)
JEP 261: Module System (`http://openjdk.java.net/jeps/261`)
JEP 275: Modular Java Application Packaging (`http://openjdk.java.net/jeps/275`)
JEP 282: `jlink`: The Java Linker (`http://openjdk.java.net/jeps/282`)
JSR 376: Java Platform Module System (`https://www.jcp.org/en/jsr/detail?id=376`)
JSR 379: Java SE 9 (`https://www.jcp.org/en/jsr/detail?id=379`)

Fig. 36.2 | Java Modularity JEPs and JSRs.

Quick Tour of the Chapter

This chapter introduces key modularity concepts you're likely to use when building large-scale systems. Some of the key topics you'll see throughout this chapter include:

- Module declarations—You'll create module declarations that specify a module's dependencies (with the `requires` directive), which packages a module makes available to other modules (with the `exports` directive), services it offers (with the `provides...with` directive), services it consumes (with the `uses` directive) and to what other modules it allows reflection (with the `open` modifier and the `opens` and `opens...to` directives).
- Module-dependency graphs—We'll use the NetBeans IDE's JDK 9 support to create module graphs that help you visualize the dependencies among modules.
- Module resolver—We'll show you the steps the runtime's module resolver performs to ensure that a module's dependencies are fulfilled.
- `jlink` (the Java linker)—You'll use this new JDK 9 tool to create smaller custom runtimes, then use them to execute apps. In fact, many of this book's command-line apps can be executed on a custom runtime consisting only of the most fundamental JDK module—**`java.base`**—which includes core Java API packages, such as `java.lang`, `java.io` and `java.util`. As you'll see, all modules *implicitly* depend on `java.base`.
- Reflection—*Reflection* enables a Java program to dynamically load types then create objects of those types and use them.[7] These capabilities can still be used, despite Java 9's strong encapsulation, but only with modules that *explicitly* allow it. We'll show how to specify that a module allows reflection with an `open` modifier and the `opens` and `opens...to` directive in a module declaration.
- Migration—The Java platform has been in use for over 20 years, so enormous amounts of non-modularized legacy code will need to be migrated to the modular

7. *The Java™ Tutorials*, "Trail: The Reflection API," `https://docs.oracle.com/javase/tutorial/reflect/`.

world of Java 9. Though there are traps and pitfalls due to Java 9's stronger encapsulation, we'll show how the unnamed module and automatic modules can help make migration straightforward. We'll use the `jdeps` tool to determine code dependencies among modules and on pre-Java-9 internal APIs (which are for the most part strongly encapsulated in Java 9). Much pre-Java-9 code will run without modification, but there are some issues that we explain in Section 36.6.

- Services and Service Providers—When you create substantial software systems that fulfill important needs, they can live on for decades. During that time, change is the rule rather than the exception. In Section 10.13, we discussed *tight coupling* and *loose coupling*. It's been proven that *tight coupling* makes it difficult to modify systems. We'll show how to create loosely coupled system components with service-provider interfaces and implementations and the `ServiceLoader` class. We'll also demonstrate the `uses` and `provides...with` directives in module declarations to indicate that a module uses a service or provides a service implementation, respectively.

We'll present the preceding concepts using several larger live-code examples with meaningful outputs, some code snippets, module graphs produced with the NetBeans IDE's **Graph** view of a module declaration and examples of various new commands (like `jlink`) and new options for existing commands (like `javac`, `java` and `jar`). Some additional example-rich sources are:

- Project Jigsaw: Module System Quick-Start Guide—`http://openjdk.java.net/projects/jigsaw/quick-start`
- Mak, Sander, and Paul Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications.* Sebastopol, CA: O'Reilly Media, 2017.

A Terminology Note

The Java Runtime Environment (JRE) includes the Java Virtual Machine (JVM) and other software for executing Java programs. As of Java 9, the JRE is now a proper subset of the Java Development Kit (JDK), which contains all the Java APIs and tools required to create and run Java programs. This chapter uses the terms Java Platform and Java SE Platform synonymously with the JDK.

36.2 Module Declarations

As we mentioned, a module must provide a module descriptor—metadata that specifies the module's dependencies, the packages the module makes available to other modules, and more. A module descriptor is the compiled version of a **module declaration** that's defined in a file named **`module-info.java`**. Each module declaration begins with the keyword **`module`**, followed by a unique **module name** and a **module body** enclosed in braces, as in

```
module modulename {
}
```

The module declaration's body can be empty or may contain various **module directives**, including `requires`, `exports`, `provides...with`, `uses` and `opens` (each of which we discuss). As you'll see in Section 36.3.5, compiling the module declaration creates the module descriptor, which is stored in a file named **`module-info.class`** in the module's root

folder. Here we briefly introduce each module directive. You'll see actual module declarations beginning in Section 36.3.3.

36.2.1 requires

A **`requires` module directive** specifies that this module depends on another module—this relationship is called a **module dependency**. Each module *must* explicitly state its dependencies. When module A `requires` module B, module A is said to **read** module B and module B **is read by** module A. To specify a dependency on another module, use `requires`, as in:

```
requires modulename;
```

Section 36.3.3 demonstrates a `requires` directive.[8]

36.2.2 requires transitive—Implied Readability

To specify a dependency on another module and to ensure that other modules reading your module also read that dependency—known as **implied readability**—use `requires transitive` as in:

```
requires transitive modulename;
```

Consider the following directive from the `java.desktop` module declaration:

```
requires transitive java.xml;
```

In this case, any module that reads `java.desktop` also *implicitly* reads `java.xml`. For example, if a method from the `java.desktop` module returns a type from the `java.xml` module, code in modules that read `java.desktop`, becomes dependent on `java.xml`. Without the `requires transitive` directive in `java.desktop`'s module declaration, such dependent modules will not compile unless they *explicitly* read `java.xml`.

According to JSR 379,[9] Java SE's standard modules *must* grant implied readability in all cases like the one described here. Also, though a Java SE standard module may depend on non-standard modules, it must *not* grant implied readability to them.

Portability Tip 36.1

*Because Java SE standard modules **must not** grant implied readability to non-standard modules, code depending only on Java SE standard modules is portable across Java SE implementations.*

36.2.3 exports and exports...to

An **`exports` module directive** specifies one of the module's packages whose `public` types (and their nested `public` and `protected` types) should be accessible to code in all other modules. An **`exports...to` directive** enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package—this is known as a **qualified export**. Section 36.4 demonstrates the `exports` directive.

8. There is also a `requires static` directive to indicate that a module is required at *compile time*, but *optional* at runtime. This is known as an *optional dependency* and is beyond this chapter's scope.
9. Clark, Iris, and Mark Reinhold. "Java SE 9 (JSR 379)." March 6, 2017. `http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html#s7`.

36.2.4 uses

A **uses module directive** specifies a service used by this module—making the module a **service consumer**. A service is an object of a class that implements the interface or extends the `abstract` class specified in the `uses` directive. Section 36.9.3 demonstrates the `uses` directive.

36.2.5 provides...with

A **`provides...with` module directive** specifies that a module provides a service implementation—making the module a **service provider**. The `provides` part of the directive specifies an interface or `abstract` class listed in a module's `uses` directive and the `with` part of the directive specifies the name of the class that `implements` the interface or `extends` the `abstract` class. Section 36.9.6 demonstrates the `provides...with` directive.

36.2.6 open, opens and opens...to[10,11]

Before Java 9, reflection could be used to learn about all types in a package and all members of a type—even its `private` members—whether you wanted to allow this capability or not. Thus, nothing was truly encapsulated.

A key motivation of the module system is *strong encapsulation*. By default, a type in a module is not accessible to other modules unless it's a `public` type *and* you export its package. You expose only the packages you want to expose. With Java 9, this also applies to reflection.

Allowing Runtime-Only Access to a Package

An **`opens` module directive** of the form

```
opens package
```

indicates that a specific *package*'s `public` types (and their nested `public` and `protected` types) are accessible to code in other modules at runtime only. Also, all of the types in the specified package (and all of the types' members) are accessible via reflection.

Allowing Runtime-Only Access to a Package By Specific Modules

An **`opens...to` module directive** of the form

```
opens package to comma-separated-list-of-modules
```

indicates that a specific *package*'s `public` types (and their nested `public` and `protected` types) types are accessible to code in the listed module(s) at runtime only. Also, all of the types in the specified package (and all of the types' members) are accessible via reflection to code in the specified modules.

10. Buckley, Alex. "JPMS: Modules in the Java Language and JVM." February 23, 2017. `http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html`.
11. Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Dan Smith. "The Java® Language Specification Java SE 9 Edition." Section 7.7.2. February 22, 2017. `http://cr.openjdk.java.net/~mr/jigsaw/spec/java-se-9-jls-pr-diffs.pdf`.

Allowing Runtime-Only Access to All Packages in a Module

If all the packages in a given module should be accessible at runtime and via reflection to all other modules, you may **open** the entire module, as in

```
open module modulename {
   // module directives
}
```

Reflection Defaults

By default, a module with runtime reflective access to a package can see the package's `public` types (and their nested `public` and `protected` types). However, the code in other modules *can* access *all* types in the exposed package and *all* members within those types, including `private` members. For more information on using reflection to access all of a type's members, visit

```
https://docs.oracle.com/javase/tutorial/reflect/
```

Dependency Injection

Reflection is commonly used with *dependency injection*. One example of this, is an FXML-based JavaFX app, like those you've seen in Chapters 12, 13, 22 and miscellaneous other examples. When an FXML app loads, the controller object and the GUI components on which it *depends* are dynamically created as follows:

- First, because the app *depends* on a controller object that handles the GUI interactions, the `FXMLLoader` *injects* a controller object into the running app—that is, the `FXMLLoader` uses reflection to locate and load the controller class into memory, and to create an object of that class.
- Next, because the controller *depends* on the GUI components declared in FXML, the `FXMLLoader` creates the GUI components objects declared in the FXML and *injects* them into the controller object by assigning each to the controller object's corresponding `@FXML` instance variable.

Once this process is complete, the controller can interact with the GUI and respond to its events. We'll use the `opens…to` directive in Section 36.7.2 to allow the `FXMLLoader` to use reflection on a JavaFX app in a custom module.

36.2.7 Restricted Keywords

The keywords `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` and `with` are *restricted keywords*. They're keywords only in module declarations and may be used as identifiers anywhere else in your code.

We mentioned in footnote 8 that there is also a `requires static` module directive. Of course, `static` is a regular keyword.

36.3 Modularized `Welcome` App

In this section, we create a simple `Welcome` app to demonstrate module fundamentals. We'll:

- create a class that resides in a module,
- provide a module declaration,

- compile the module declaration and `Welcome` class into a module, and
- run the class containing `main` in that module.

After covering these basics, we'll also demonstrate:

- packaging the `Welcome` app in a modular JAR file and
- running the app from that JAR file.

36.3.1 Welcome App's Structure

The app we present in this section consists of two `.java` files—`Welcome.java` contains the `Welcome` app class and `module-info.java` contains the module declaration. By convention, a modularized app has the following folder structure:

```
AppFolder
   src
      ModuleNameFolder
         PackageFolders
            JavaSourceCodeFiles
         module-info.java
```

For our `Welcome` app, which will be defined in the package `com.deitel.welcome`, the folder structure is shown in Fig. 36.3.

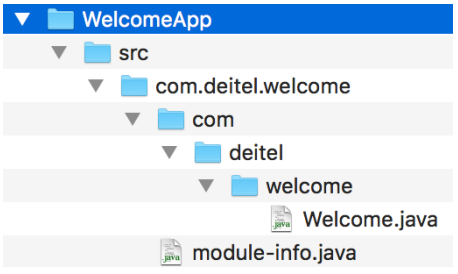


Fig. 36.3 | Folder structure for the `Welcome` app.

The `src` folder stores all of the app's source code. It contains the module's **root folder**, which has the module's name—`com.deitel.welcome` (we'll discuss module naming momentarily). The module's root folder contains nested folders representing the package's directory structure—`com/deitel/welcome`—which corresponds to the package `com.deitel.welcome`. This folder contains `Welcome.java`. The module's root folder contains the required module declaration `module-info.java`.

Module Naming Conventions

Like package names, module names must be *unique*. To ensure *unique* package names, you typically begin the name with your organization's Internet domain name in reverse order. Our domain name is `deitel.com`, so we begin our package names with `com.deitel`. By convention, module names also use the reverse-domain-name convention.

At compile time, if multiple modules have the same name, a compilation error occurs. At runtime, if multiple modules have the same name an exception occurs.

This example uses the same name for the module and its contained package, because there is only one package in the module. This is not required, but is a common convention. In a modular app, Java maintains the module names separately from package names and any type names in those packages, so duplicate module and package names *are* allowed.

Modules normally group related packages. As such, the packages will often have commonality among portions of their names. For example, if a module contains the packages

```
com.deitel.sample.firstpackage;
com.deitel.sample.secondpackage;
com.deitel.sample.thirdpackage;
```

you'd typically name the module with the common portion of the package names—`com.deitel.sample`. If there's no common portion, then you'd choose a name representing the module's purpose. For example, the `java.base` module contains core packages that are considered fundamental to Java apps (such as `java.lang`, `java.io`, `java.time` and `java.util`), and the `java.sql` module contains the packages required for interacting with databases via JDBC (such as `java.sql` and `javax.sql`). These are just two of the many standard modules that you saw in Fig. 36.1. The online documentation for each provides a complete list of its exported packages—for the `java.base` module, visit:

```
http://download.java.net/java/jdk9/docs/api/java.base-summary.html
```

Listing the `java.base` *Module's Contents*

You can use the `java` command's `--list-modules` option to display information from the `java.base` module's descriptor, including its list of exported packages, as in:

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

Figure 36.4 shows the *portion* of the preceding command's output which lists the `java.base` module's packages that *any* module can access. You've used several of these packages in the book, including `java.io`, `java.lang`, `java.math`, `java.nio`, `java.time` and `java.util`.

```
exports java.io
exports java.lang
exports java.lang.annotation
exports java.lang.invoke
exports java.lang.module
exports java.lang.ref
exports java.lang.reflect
exports java.math
exports java.net
exports java.net.spi
exports java.nio
exports java.nio.channels
exports java.nio.channels.spi
exports java.nio.charset
exports java.nio.charset.spi
exports java.nio.file
exports java.nio.file.attribute
exports java.nio.file.spi
exports java.security
```

```
exports java.security.acl
exports java.security.cert
exports java.security.interfaces
exports java.security.spec
exports java.text
exports java.text.spi
exports java.time
exports java.time.chrono
exports java.time.format
exports java.time.temporal
exports java.time.zone
exports java.util
exports java.util.concurrent
exports java.util.concurrent.atomic
exports java.util.concurrent.locks
exports java.util.function
exports java.util.jar
exports java.util.regex
exports java.util.spi
```

Fig. 36.4 | Partial output of the command `java --list-modules java.base`. (Part 1 of 2.)

```
exports java.util.stream
exports java.util.zip
exports javax.crypto
exports javax.crypto.interfaces
exports javax.crypto.spec
exports javax.net
exports javax.net.ssl
```

```
exports javax.security.auth
exports javax.security.auth.callback
exports javax.security.auth.login
exports javax.security.auth.spi
exports javax.security.auth.x500
exports javax.security.cert
```

Fig. 36.4 | Partial output of the command `java --list-modules java.base`. (Part 2 of 2.)

The complete output of the preceding command lists lots of additional information about the `java.base` module. Figure 36.5 shows some of the remaining output with sample lines from each category of information.

```
...
  uses java.util.spi.CurrencyNameProvider
  uses java.util.spi.ResourceBundleControlProvider
  uses java.util.spi.LocaleNameProvider
...
  provides java.nio.file.spi.FileSystemProvider
     with jdk.internal.jrtfs.JrtFileSystemProvider
...
  exports sun.net.sdp to oracle.net
  exports jdk.internal.jimage to jdk.jlink
  exports sun.net.www.protocol.http.ntlm to jdk.deploy
...
  contains com.sun.crypto.provider
  contains com.sun.java.util.jar.pack
  contains com.sun.net.ssl
 ...
```

Fig. 36.5 | Partial output of the command `java --list-modules java.base` showing other categories of information that it displays.

The `uses` lines, like

```
uses java.util.spi.CurrencyNameProvider
```

indicate that there are types in the `java.base` module's packages which use objects that implement various service-provider interfaces. The `provides...with`

```
provides java.nio.file.spi.FileSystemProvider
   with jdk.internal.jrtfs.JrtFileSystemProvider
```

indicates that this module's `jdk.internal.jrtfs` package contains a service-provider implementation class named `JrtFileSystemProvider` that implements the service-provider interface named `FileSystemProvider` from package `java.nio.file.spi`. Section 36.9 shows a substantial example demonstrating that service-provider interfaces and implementations can be used to create *loosely coupled* system components for systems that are easier to develop, maintain and evolve than tightly coupled systems.

The `exports...to` lines like

```
exports sun.net.sdp to oracle.net
```

indicate that the `java.base` module `exports` a given package (`sun.net.sdp`) *only to a specified module* (`oracle.net`). The `java.base` module has many of these qualified exports. Packages listed in such exports may be read only by the one or more designated modules in the comma-separated list after the keyword `to`. In the JDK, such qualified exports are used for packages (like `sun.net.sdp`) containing JDK internal implementations of types that should not be used by developers.

The `contains` lines, like

```
contains com.sun.crypto.provider
```

specify that the module contains packages that are not exported for use in other modules. Note that `contains` is not a directive like `requires` or `exports` that you can use in your modules. Rather, it's information inserted by the compiler to indicate that a module contains the specified package—the package is not exported for use by other modules. The JVM uses this information to improve performance when it loads classes from those packages at runtime.[12]

36.3.2 Class `Welcome`

Figure 36.6 presents a `Welcome` app that simply displays a `String` at the command line. When defining types that will be placed in modules, every type *must* be placed into a package (line 3).

```
1  // Fig. 36.6: Welcome.java
2  // Welcome class that will be placed in a module
3  package com.deitel.welcome; // all classes in modules must be packaged
4
5  public class Welcome {
6     public static void main(String[] args) {
7        // class System is in package java.lang from the java.base module
8        System.out.println("Welcome to the Java Platform Module System!");
9     }
10 }
```

Fig. 36.6 | `Welcome` class that will be placed in a module.

36.3.3 `module-info.java`

Figure 36.7 contains the module declaration for the `com.deitel.welcome` module. We call modules we create for our own use **application modules**.

```
1  // Fig. 36.7: module-info.java
2  // Module declaration for the com.deitel.welcome module
3  module com.deitel.welcome {
4     requires java.base; // implicit in all modules, so can be omitted
5  }
```

Fig. 36.7 | Module declaration for the `com.deitel.welcome` module.

12. Brian Goetz, e-mail message to authors, March 16, 2017.

Again, the module declaration begins with the keyword `module` followed by the module's name and braces that enclose the declaration's body. This module declaration contains a `requires` module directive, indicating that the app depends on types defined in module `java.base`. Actually, all modules depend on `java.base`, so the `requires` module directive in line 4 is *implicit* in all `module` declarations and may be omitted, as in:

```
module com.deitel.welcome {
}
```

Software Engineering Observation 36.1

Every module implicitly depends on `java.base`. Writing `requires java.base;` in a module declaration is redundant.

36.3.4 Module-Dependency Graph

Figure 36.8 shows the **module-dependency graph** for `com.deitel.welcome`, indicating that the module reads only the standard module `java.base`. This dependency is indicated in the diagram with the arrow from `com.deitel.welcome` to `java.base`. This graph will be identical regardless of whether you include line 4 in the module declaration.

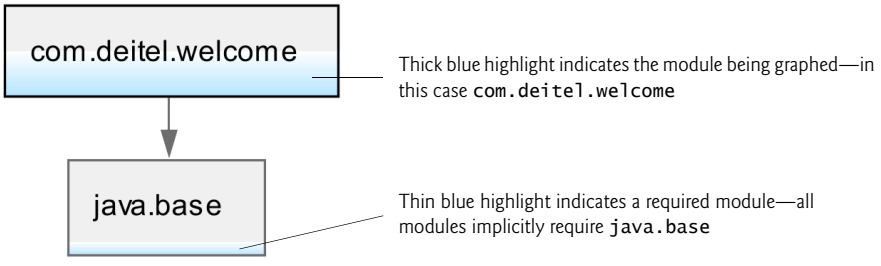


Fig. 36.8 | Module-dependency graph for the `com.deitel.welcome` module.

A module-dependency graph shows dependencies among **observable modules**[13]—that is, the built-in standard modules and any additional modules required by a given app or library module. The graph's *nodes* are modules and their *dependencies* are represented by directed edges (arrows) that connect the nodes. Some edges represent **explicit dependencies** on modules explicitly specified in a module declaration's `requires` clauses (as you'll see in Fig. 36.14). Some edges represent **implicit dependencies** in which one of the required modules in turn depends on other modules (as you'll see in Fig. 36.22). In Fig. 36.8, `java.base` is shown as an explicit dependency, because all modules depend on it.

This graph was produced with an early access version of NetBeans that has JDK 9 support—again, you can learn about this version of the IDE and download its installer from:

```
http://wiki.netbeans.org/JDK9Support
```

In a NetBeans project, when you open a module's `module-info.java` file, you can choose between the **Source** code and **Graph** views. In **Graph** view, NetBeans creates a module-de-

13. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." `http://openjdk.java.net/jeps/261`.

pendency- graph, based on the module declaration. When NetBeans graphs a module, it also graphs that module's dependencies, including the implicit dependency on `java.base`. Figure 36.8 shows that `java.base` itself does not have any dependencies.

To create this graph in NetBeans, we performed the following steps:

1. First, we created a `WelcomeApp` project containing the `com.deitel.welcome` package.
2. Next, we added the `com.deitel.welcome` module's `module-info.java` file by right-clicking the project, selecting **New > Other...**, selecting **Java Module Info** from the **Java** category of the dialog, then clicking **Next >** and **Finish**. The file is added to the project's default package automatically.
3. Finally, we opened `module-info.java` file, changed the module name from the default provided by NetBeans (the project name) to `com.deitel.welcome` and switched to **Graph** view.

You can arrange the nodes in NetBeans by dragging them or by right clicking the graph and selecting from various **Layout** options—we chose **Hierarchical**, in which the given module appears at the top and arrows point down to the module's dependencies. You may use **Zoom To Fit** to make the graph fill the available space in the window and **Export As Image** to save an image containing the graph.

36.3.5 Compiling a Module

To compile the `Welcome` app's module, open a command window, use the `cd` command to change to this chapter's `WelcomeApp` folder, then type:

```
javac -d mods/com.deitel.welcome ^
   src/com.deitel.welcome/module-info.java ^
   src/com.deitel.welcome/com/deitel/welcome/Welcome.java
```

The `-d` option indicates that `javac` should place the compiled code in the specified folder—in this case a `mods` folder that will contain a subfolder named `com.deitel.welcome` representing the compiled module. The name `mods` is used by convention for a folder that contains modules.

Note Regarding Lengthy Commands in This Chapter

For clarity, we split the preceding command into multiple lines, using line-continuation characters. Many of the commands we use in this chapter's examples are lengthy. This chapter shows the commands in Windows format, with the caret (^) line-continuation character. Linux and macOS users should replace the carets in the commands with the backslash (\) line-continuation character. You can also enter such lengthy commands as a single command without the line continuations.

`Welcome` App's Folder Structure After Compilation

If the code compiles correctly, the `WelcomeApp` folder's `mods` subfolder structure contains the compiled code (Fig. 36.9). This is known as the **exploded-module folder**, because the folders and `.class` files are not in a JAR (Java archive) file—collection of directories and files compressed into a single file, known as an archive. The exploded module's structure parallels that of the app's `src` folder described previously. We'll package the app as a JAR

shortly. Exploded module folders and modular JAR files (Section 36.3.7) together are **module artifacts**. These can be placed on the **module path**—a list of module artifact locations—when compiling and executing modularized code.[14,15]

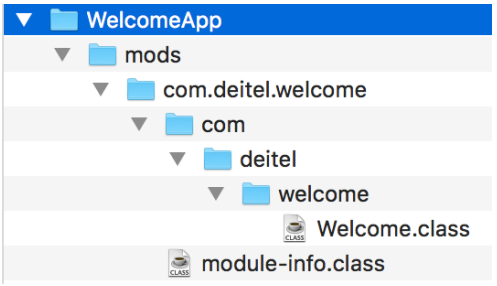


Fig. 36.9 | `Welcome` app's `mods` folder structure.

Listing the `com.deitel.welcome` Module's Contents

You can use the `java` command's `--list-modules` option to display information from the `com.deitel.welcome` module descriptor, as in:

```
java --module-path mods --list-modules com.deitel.welcome
```

The resulting output:

```
module com.deitel.welcome (file:///C:/examples/ch36/WelcomeApp/
mods/com.deitel.welcome/)
  requires java.base (@9-ea)
  contains com
  contains com.deitel
  contains com.deitel.welcome
```

shows that the module `requires` the standard module `java.base` and `contains` the packages `com`, `com.deitel` and `com.deitel.welcome` (each folder is viewed as a package). Though the module `contains` these packages, they are *not* exported. Therefore, its contents *cannot* be used by other modules. The module declaration for this example *explicitly* required `java.base` and the preceding listing included

```
requires java.base
```

If the module declaration had *implicitly* required `java.base`, then the listing instead would have included

```
requires mandated java.base
```

There is no `requires mandated` module directive—it is simply included in the `--list-modules` output to indicate the implicit dependence on `java.base`.

14. Reinhold, Mark. "The State of the Module System." March 8, 2016. `http://openjdk.java.net/projects/jigsaw/spec/sotms/#module-artifacts`.
15. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." `http://openjdk.java.net/jeps/261`.

36.3.6 Running an App from a Module's Exploded Folders

To run the `Welcome` app from the module's exploded folders, use the following command (again, from the `WelcomeApp` folder):

```
java --module-path mods ^
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

The **`--module-path`** option specifies the module path—in this case, the `mods` folder. The **`--module`** option specifies the module name and the fully qualified class name of the app's entry point—that is, a class containing `main`. The program executes and displays:

```
Welcome to the Java Platform Module System!
```

In the preceding command, `--module-path` can be abbreviated as `-p` and `--module` as `-m`.

36.3.7 Packaging a Module into a Modular JAR File

You can use the `jar` command to package an exploded module folder as a **modular JAR file**[16] that contains all of the module's files, including its `module-info.class` file, which is placed in the JAR's root folder. When running the app, you specify the JAR file on the module path. The folder in which you wish to output the JAR file must exist before running the `jar` command.

If a module contains an app's entry point, you can specify that class with the `jar` command's **`--main-class`** option, as in:

```
jar --create -f jars/com.deitel.welcome.jar ^
   --main-class com.deitel.welcome.Welcome ^
   -C mods/com.deitel.welcome .
```

The options are as follows:

- `--create` specifies that the command should create a new JAR file.
- `-f` specifies the name of the JAR file and is followed by the name—in this case, the file `com.deitel.welcome.jar` will be created in the folder named `jars`.
- `--main-class` specifies the fully qualified name of the app's entry point—a class that contains a `main` method.
- `-C` specifies which folder contains the files that should be included in the JAR file and is followed by the files to include—the dot (`.`) indicates that all files in the folder should be included.

You can simplify the `-create`, `-f` and `--main-class` options in the preceding command with the shorthand notation `-cfe`, followed by the JAR file name and main class, as in:

```
jar -cfe jars/com.deitel.welcome.jar ^
   com.deitel.welcome.Welcome ^
   -C mods/com.deitel.welcome .
```

16. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." `http://openjdk.java.net/jeps/261`.

36.3.8 Running the Welcome App from a Modular JAR File

Once you place an app in a modular JAR file for which you've specified the entry point, you can execute the app as follows:

```
java --module-path jars -m com.deitel.welcome
```

or

```
java -p jars -m com.deitel.welcome
```

The program executes and displays:

```
Welcome to the Java Platform Module System!
```

If you did not specify the entry point when creating the JAR, you may still run the app by specifying the module name and fully qualified class name, as in:

```
java --module-path jars ^
   -m com.deitel.welcome/com.deitel.welcome.Welcome
```

or

```
java -p jars -m com.deitel.welcome/com.deitel.welcome.Welcome
```

36.3.9 Aside: Classpath vs. Module Path

Before Java 9, the compiler and runtime located types via the *classpath*—a list of folders and library archive files containing compiled Java classes. In earlier Java versions, the classpath was defined by a combination of a `CLASSPATH` environment variable, extensions placed in a special folder of the JRE, and options provided to the `javac` and `java` commands.

Because types could be loaded from several different locations, the order in which those locations were searched resulted in brittle apps. For example, many years ago, one of the authors installed a Java app from a third-party vendor on his system. The app's installer placed an old version of a Java library into the JRE's extensions folder. Several Java apps on his system depended on a newer version of that library with additional types and enhanced versions of the library's older types. Because classes in the JRE's extensions folder were loaded *before* other classes on the classpath,[17] the apps that depended on the newer library version stopped working, failing at runtime with `NoClassDefFoundErrors` and `NoSuchMethodErrors`—sometimes long after the apps began executing.

The reliable configuration provided by modules and module descriptors helps eliminate many such runtime classpath problems. Every module explicitly states its dependencies and these are resolved *as an app launches*. In Section 36.8.5, we'll show the steps that the JRE's *module resolver* performs at launch time.

Common Programming Error 36.1

The module path may contain only one of each module and every package may be defined in only one module. If two or more modules have the same name or export the same packages, the runtime immediately terminates before running the program.

17. "Understanding Extension Class Loading." `https://docs.oracle.com/javase/tutorial/ext/basics/load.html`.

36.4 Creating and Using a Custom Module

To demonstrate a module that depends on another custom module in addition to standard modules, let's reorganize one of the book's earlier, non-modularized examples. We'll declare Section 8.2's `Time1` and `Time1Test` classes in separate modules, then use class `Time1` from the module containing `Time1Test`. As you'll see, we'll *export* class `Time1`'s package from one module and *require* `Time1`'s enclosing module from a module containing the `Time1Test` class. Figure 36.10 shows the `src` folder structure for the app's two modules.

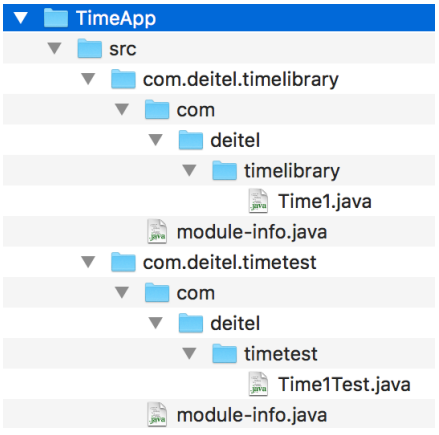


Fig. 36.10 | `TimeApp` example's `src` folder structure.

36.4.1 Exporting a Package for Use in Other Modules

As you learned previously, every class that you wish to place in a module *must* be declared in a package. For this reason, we added the `package` statement in line 3 (Fig. 36.11) to class `Time1` (which was originally declared in Fig. 8.1).

```
1  // Fig. 36.11: Time1.java
2  // Class Time1 that will be placed in a module.
3  package com.deitel.timelibrary;
4
5  public class Time1 {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; throw an
11    // exception if the hour, minute or second is invalid
12    public void setTime(int hour, int minute, int second) {
13       // validate hour, minute and second
14       if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
15          second < 0 || second >= 60) {
```

Fig. 36.11 | Class `Time1` that will be placed in a module. (Part 1 of 2.)

```
16             throw new IllegalArgumentException(
17                "hour, minute and/or second was out of range");
18          }
19
20          this.hour = hour;
21          this.minute = minute;
22          this.second = second;
23       }
24
25       // convert to String in universal-time format (HH:MM:SS)
26       public String toUniversalString() {
27          return String.format("%02d:%02d:%02d", hour, minute, second);
28       }
29
30       // convert to String in standard-time format (H:MM:SS AM or PM)
31       public String toString() {
32          return String.format("%d:%02d:%02d %s",
33             ((hour == 0 || hour == 12) ? 12 : hour % 12),
34             minute, second, (hour < 12 ? "AM" : "PM"));
35       }
36    }
```

Fig. 36.11 | Class `Time1` that will be placed in a module. (Part 2 of 2.)

`com.deitel.timelibrary` Module Declaration

After placing `Time1` in a package, we must declare the `module` via a module declaration (Fig. 36.12). Line 4 indicates that the module `com.deitel.timelibrary` `exports` the package `com.deitel.timelibrary`. Now the package's `public` classes (in this case, just class `Time1`) can be used by *any* module that reads the `com.deitel.timelibrary` module, provided that the module can be found on the module path, as you'll see in Section 36.4.3.

```
1  // Fig. 36.12: module-info.java
2  // Module declaration for the com.deitel.timelibrary module
3  module com.deitel.timelibrary {
4     exports com.deitel.timelibrary; // package available to other modules
5  }
```

Fig. 36.12 | Module declaration for the `com.deitel.timelibrary` module.

36.4.2 Using a Class from a Package in Another Module

The app's entry point—class `Time1Test` (which was originally declared in Fig. 8.2)—also must be packaged for placement in a module (line 3 of Fig. 36.13). In addition, class `Time1Test` manipulates an object of class `Time1`, which is declared in a package of another module. For this reason, we import `Time1` in line 5.

```
1  // Fig. 36.13: Time1Test.java
2  // Time1 object used in an app.
3  package com.deitel.timetest;
```

Fig. 36.13 | `Time1` object used in an app. (Part 1 of 2.)

```
4
5  import com.deitel.timelibrary.Time1;
6
7  public class Time1Test {
8     public static void main(String[] args) {
9        // create and initialize a Time1 object
10       Time1 time = new Time1(); // invokes Time1 constructor
11
12       // output string representations of the time
13       displayTime("After time object is created", time);
14       System.out.println();
15
16       // change time and output updated time
17       time.setTime(13, 27, 6);
18       displayTime("After calling setTime", time);
19       System.out.println();
20
21       // attempt to set time with invalid values
22       try {
23          time.setTime(99, 99, 99); // all values out of range
24       }
25       catch (IllegalArgumentException e) {
26          System.out.printf("Exception: %s%n%n", e.getMessage());
27       }
28
29       // display time after attempt to set invalid values
30       displayTime("After calling setTime with invalid values", time);
31    }
32
33    // displays a Time1 object in 24-hour and 12-hour formats
34    private static void displayTime(String header, Time1 t) {
35       System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
36          header, t.toUniversalString(), t.toString());
37    }
38 }
```

Fig. 36.13 | `Time1` object used in an app. (Part 2 of 2.)

`com.deitel.timetest` Module Declaration

Because class `Time1` is located in a package of the `com.deitel.timelibrary` module, the module containing class `Time1Test` (`com.deitel.timetest`) must declare its dependency on that other module. The module declaration (Fig. 36.14) indicates this dependency with the `requires` directive (line 4). Without this *and* the `exports` directive in Fig. 36.12, class `Time1Test` would not be able to import and use class `Time1`.

```
1  // Fig. 36.14: module-info.java
2  // Module declaration for the com.deitel.timetest module
3  module com.deitel.timetest {
4     requires com.deitel.timelibrary;
5  }
```

Fig. 36.14 | Module declaration for the `com.deitel.timetest` module.

com.deitel.timetest Module-dependency Graph

Figure 36.15 shows the `Time1Test` app's module-dependency graph indicating that:

- the module named `com.deitel.timetest` reads `com.deitel.timelibrary` and the standard module `java.base`, and
- the module named `com.deitel.timelibrary` reads the module `java.base`.

To create this graph in NetBeans, we performed the following steps:

1. Created a `TimeLibrary` project containing the `com.deitel.timelibrary` package and `com.deitel.timelibrary`'s `module-info.java` file.
2. Created a `TimeApp` project containing the `com.deitel.timetest` package and `com.deitel.timetest`'s `module-info.java` file.
3. Right clicked the `TimeApp` project's **Libraries** node and selected **Add Project...**, then selected the `TimeLibrary` project and clicked **Add Project JAR Files**—this adds the `TimeLibrary` project's modular JAR file to the `TimeApp` project.
4. Finally, we opened the `TimeApp` project's `module-info.java` file in **Graph** view.

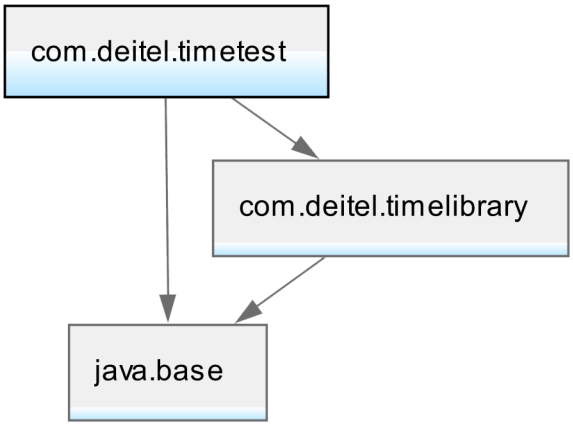


Fig. 36.15 | Module-dependency graph for the `com.deitel.timetest` module.

36.4.3 Compiling and Running the Example

You must compile both modules before running this app. The `com.deitel.timelibrary` module must be compiled first, because `com.deitel.timetest` depends on it. IDEs and other build tools (like Ant, Gradle and Maven) typically can deal with order-of-compilation issues like this for you.

Compiling Module com.deitel.timelibrary

To compile the `com.deitel.timelibrary` module, open a command window, use the `cd` command to change to this chapter's `TimeApp` folder on your system, then type:

```
javac -d mods/com.deitel.timelibrary ^
   src/com.deitel.timelibrary/module-info.java ^
   src/com.deitel.timelibrary/com/deitel/timelibrary/Time1.java
```

Compiling Module `com.deitel.timetest`

Next, to compile the `com.deitel.timetest` module, type:

```
javac --module-path mods -d mods/com.deitel.timetest ^
   src/com.deitel.timetest/module-info.java ^
   src/com.deitel.timetest/com/deitel/timetest/Time1Test.java
```

Here we added the option `--module-path` to indicate that the `mods` folder contains modules on which the `com.deitel.timetest` module depends—in this case, we previously compiled the `com.deitel.timelibrary` module into the `mods` folder.

Running the Example

Finally, to run this example, type:

```
java --module-path mods ^
   -m com.deitel.timetest/com.deitel.timetest.Time1Test
```

In this command:

- the option `--module-path` indicates where the app's modules are located, and
- the option `-m` specifies which class should be used as the app's entry point—that is, a class containing the `main` method that the JVM calls to launch the app.

For the `main` class, note that you must specify its module name followed by a slash and its fully qualified class name, because the class is now in a package contained in a module. The program's output is shown below:

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

36.4.4 Packaging the App into Modular JAR Files

In this section, we'll package each app into a modular JAR file then run the app. To package `com.deitel.timelibrary` into a modular JAR file, type:

```
jar --create -f jars/com.deitel.timelibrary.jar ^
   -C mods/com.deitel.timelibrary .
```

To package `com.deitel.timetest` into a modular JAR file, type:

```
jar --create -f jars/com.deitel.timetest.jar ^
   --main-class com.deitel.timetest.Time1Test ^
   -C mods/com.deitel.timetest .
```

Running the App from a Modular JAR File

Once you place an app in a modular JAR file for which you've specified the `main` class, you can execute the app as follows:

```
java --module-path jars -m com.deitel.timetest
```

The program executes and displays the same output shown in Section 36.4.3.

36.4.5 Strong Encapsulation and Accessibility

Before Java 9, you could use any `public` class that you imported into your code. Whether you could access the class's members was determined by how they were declared—`public`, `protected`, package access or `private` (as described in Chapters 3–8). Due to Java 9's **strong encapsulation** in modules, `public` types in a module are no longer *accessible* to your code by default—so `public` no longer means available to all:

- If a module exports a package, the `public` types in that package are accessible by *any* module that reads the package's module.
- If a module exports a package to a specific module (via `exports...to`), the `public` types in that package are accessible *only* to the specific module and only if that module *reads* the package's module.
- If a module does not export a package, the `public` types in that package are accessible *only* within their enclosing module.

Once you have access to a type in another module, then the normal rules of `public`, `protected`, package access and `private` apply.

Compilation Error When Attempting to Use an Inaccessible Type

The project `TimeAppMissingExports` in this chapter's `ExamplesShowingErrors` folder demonstrates that *explicitly named modules* have *strong encapsulation* and do not export packages unless you *explicitly* list them in `exports` directives. In this project, we removed the `exports` directive from the `com.deitel.timelibrary`'s module declaration, then recompiled the module. Next, we tried to recompile the `com.deitel.timetest` module. The compiler produced the following error message, which indicates that the package `com.deitel.timelibrary` is not exported and thus is inaccessible:

```
src\com.deitel.timetest\com\deitel\timetest\Time1Test.java:5:
error: package com.deitel.timelibrary is not visible
import com.deitel.timelibrary.Time1;
                   ^
  (package com.deitel.timelibrary is declared in module
com.deitel.timelibrary, which is not in the module graph)
1 error
```

Common Programming Error 36.2

When a `requires` dependency is not fulfilled by an `exports` clause in another module a compilation error occurs.

36.5 Module-Dependency Graphs: A Deeper Look

Previously, we've shown two module-dependency graphs. Here we continue our discussion of module graphs and show the errors that occur if a module directly or indirectly `requires` itself—known as a cycle.

36.5.1 `java.sql`

Figure 36.16 shows the module-dependency graph for a module named `modulegraphtest` that depends on the `java.sql` module, per the following module declaration:

```
module modulegraphtest {
   requires java.sql;
}
```

NetBeans highlights the module declared by the module declaration (`modulegraphtest`) with a thick blue line. It also highlights `java.sql`, because it's *explicitly* listed in a `requires` directive and `java.base`, because it's implicitly required by all modules. The other modules shown (`java.xml` and `java.logging`) are included in the graph, because `java.sql` depends on them.

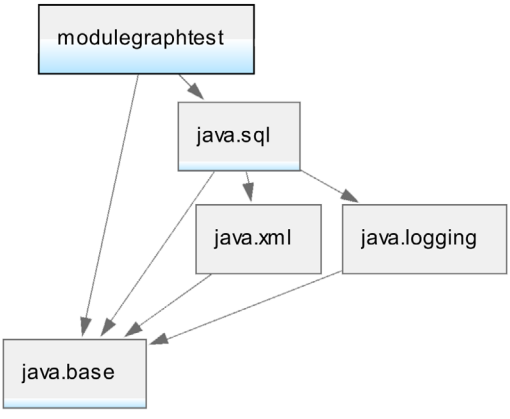


Fig. 36.16 | Dependency graph for a module that depends on `java.sql`.

36.5.2 `java.se`

Figure 36.17 shows the significantly more complex **`java.se`** module's dependency graph—this is an **aggregator module** that specifies via `requires transitive` all the modules necessary to support Java SE 9 apps. To produce this graph, we first downloaded the JDK 9 source code, as described at

```
http://hg.openjdk.java.net/jdk9/jdk9/raw-file/tip/common/doc/
   building.html
```

We then opened the `java.se` module's declaration (located in the source-code folder's `jdk/src/java.se/share/classes` folder) in NetBeans **Graph** view. We rotated the graph 90° for readability. There is also a **`java.se.ee`** aggregator module, which includes everything in the `java.se` module and additional Java SE modules with packages that overlap with the Java Enterprise Edition (EE) Platform.

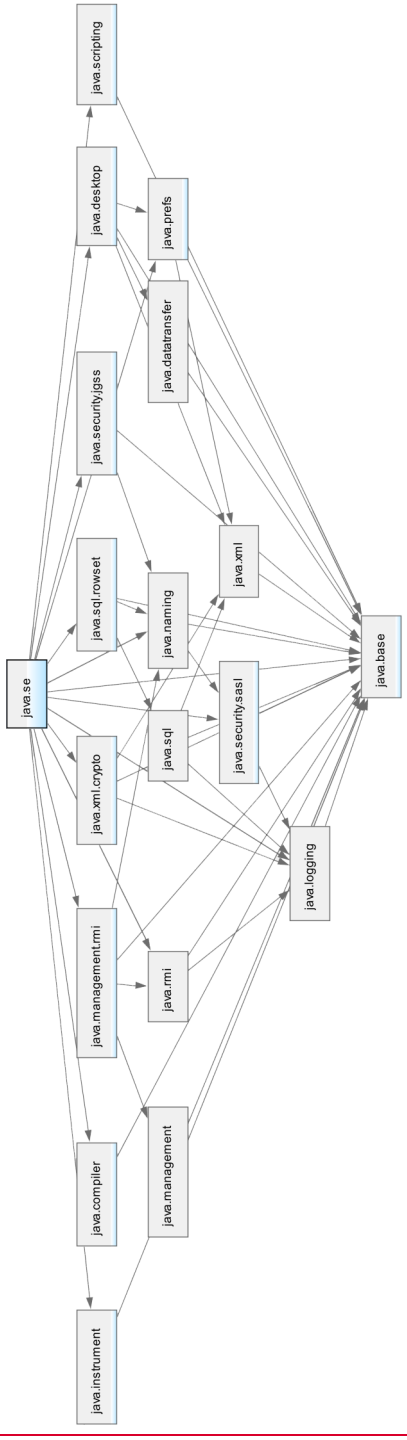


Fig. 36.17 | `java.se` module-dependency graph.

36.5.3 Browsing the JDK Module Graph

It's interesting to look at the JDK's full module-dependency graph. This is the largest of the module graphs we show. You can view the graph on our website at:

```
http://deitel.com/bookresources/jhtp11/ModularJDKGraph.png
```

When you open it with your web browser, it will initially display the complete image in the browser's window. Click the image to zoom in, then scroll horizontally and vertically to view the graph's details. We produced this image using the Graphviz tool available from

```
http://www.graphviz.org/
```

36.5.4 Error: Module Graph with a Cycle

A module is not allowed to directly or indirectly reference itself. Doing so would result in a *cycle* when computing the module's dependency graph.

Common Programming Error 36.3

A compilation error occurs if a module graph contains a cycle.

A Module That (Incorrectly) Requires Itself

Consider the following module declaration in which the module requires itself:

```
module mymodule {
   requires mymodule;
}
```

When you compile this declaration, the following error occurs, indicating a cycle in the `module`'s dependencies:

```
module-info.java:2: error: cyclic dependence involving mymodule
   requires mymodule;
            ^
1 error
```

Two Modules That (Incorrectly) Require One Another

Similarly, consider a project named `CircularDependency` containing two modules—`module1` and `module2`—with the structure shown in Fig. 36.18.

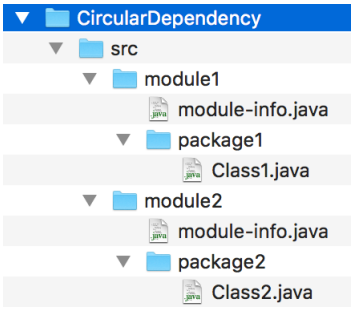


Fig. 36.18 | `CircularDependency` example's `src` folder structure.

If the module declarations for these two modules indicate that each module `requires` the other, as in

```
module module1 {
   exports package1;
   requires module2;
}
```

and

```
module module2 {
   exports package2;
   requires module1;
}
```

then, when you compile these modules

```
javac --module-source-path src ^
   --module-path mods -d mods ^
   src/module1/module-info.java ^
   src/module1/package1/Class1.java ^
   src/module2/module-info.java ^
   src/module2/package2/Class2.java
```

the compiler again issues an error indicating a cycle in the module dependencies:

```
src\module1\module-info.java:9: error: cyclic dependence involving
module2
   requires module2;
            ^
1 error
```

Modules in a Cycle Are Really "One Thing"

Ultimately all the modules in a cycle are really one module—not separate modules.[18] While we were writing this chapter, a friend of ours who works for a large organization told us that his group is preparing for Java 9 modularity. He indicated that they have multiple large pre-Java-9 JAR files. Initially they thought they'd make each JAR a separate module, but their JARs turned out to be so interdependent that they've decided to combine them into a single module. This kind of interdependency is what leads to cycles in your design. Ideally, when you modularize a previously monolithic system, you want to break that system into separate modules that are easier to maintain and secure. This can pose significant refactoring challenges in large code bases.

36.6 Migrating Code to Java 9

Many pre-Java-9 apps will run unaltered on Java 9. In fact, as we prepared this book, we tested every app using JDK 9 and they all compiled and ran without issue. In Java 9, all programs are compiled and executed using the module system. Java 9 strongly encapsulates types that are not exported by modules, so it's possible that some apps will fail to compile because types that were accessible to them prior to Java 9 no longer are. For example, there are many pre-Java-9 **internal APIs** that were not meant for use outside the JDK, but were in fact used outside the JDK—many of these are not exported in Java 9 and thus are

18. Alex Buckley, e-mail message to authors, March 24, 2017.

inaccessible.[19] If your code uses such internal APIs directly or indirectly, it will fail to compile.

Some internal APIs considered critically important are still available in Java 9. Various JEPs referenced by JSR 379[20] define new public APIs that replace these internal APIs. These internal APIs will eventually be removed.

Software Engineering Observation 36.2

Modularity enables strong encapsulation. Code that is not exported cannot be accessed by other modules.

Error-Prevention Tip 36.1

You can use the `jdeps` *tool (Section 36.6.3) released with Java 8 to locate a type's dependencies or the dependencies for all types in a JAR file. In Java 9, the tool also supports modules. The* `jdeps` *option* `--jdk-internals` *specifically identifies uses of JDK internal APIs in code. Some pre-Java-9 internal APIs have been placed into packages that are exported in Java 9 and some are now strongly encapsulated. For each internal API that* `jdeps` *locates, you can review JEP 260 and update your code accordingly.*

Common Programming Error 36.4

JDK 9 hides most pre-Java-9 internal APIs, so pre-Java-9 code that uses them will not compile and run on Java 9.

Java is more than two decades old so there's vast amounts of legacy Java code to migrate to Java 9. The module system provides mechanisms that can automatically place your code in modules to help you with migration.

36.6.1 Unnamed Module

In Java 9, all code is required to be placed in modules. When you execute code that's not in a module, the code is loaded from the *classpath* and placed in the **unnamed module**. This is why we can run some non-modularized code in the modularized JDK, but unfortunately without the benefits of modularization.

The unnamed module:

- *implicitly* `exports` all of its packages, and
- *implicitly reads* all other modules.

However, because the module is *unnamed*, there's no way to refer to it in a `requires` directive from a named module, so a named module cannot depend on the unnamed module.

36.6.2 Automatic Modules

There are enormous numbers of preexisting libraries that you can use in your apps. Many of these are not yet modularized. However, to facilitate migration, you can add *any* library's JAR file to an app's module path, then use the packages in that JAR. When you do, the JAR file implicitly becomes an **automatic module** and can be specified in a `module`

19. Reinhold, Mark. "JEP 260: Encapsulate Most Internal APIs." `http://openjdk.java.net/jeps/260`.
20. Clark, Iris, and Mark Reinhold. "Java SE 9 (JSR 379)." March 6, 2017. `http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html`.

declaration's `requires` directives. The JAR's file name—minus the `.jar` extension—becomes its module name, which *must be a valid Java identifier* for use in a `requires` directive. Also, an automatic module:

- *implicitly* `exports` all of its packages—so, any module that reads the automatic module (including the unnamed module) has access to the `public` types in the automatic module's packages.
- *implicitly reads* (`requires`) all other modules, including other automatic modules and the unnamed module—so, an automatic module has access to all the `public` types exposed by the system's other modules.

We demonstrate an automatic module in Section 36.7.

36.6.3 `jdeps`: Java Dependency Analysis

Another tool to help you migrate your code to Java 9 is the **`jdeps` command**, which was introduced in Java 8 to help you determine a type's *class* and *package* dependencies. A key use of `jdeps` is to locate dependencies on pre-Java-9 internal APIs that are now strongly encapsulated in Java 9. To determine whether a class has any such dependencies, use the following command on your compiled pre-Java-9 code:

```
jdeps --jdk-internals YourClassName.class
```

or if you have many classes in a JAR file, use:

```
jdeps --jdk-internals YourJARName.jar
```

If this command produces no output, then your class or set of classes does not have any dependence on JDK internal APIs that are no longer accessible.

Error-Prevention Tip 36.2

Check every pre-Java-9 compiled class/JAR file with the `jdeps` command to ensure that your code does not depend on JDK internal APIs.

Determining the Modules You Need

Java 9 adds the ability to discover *module* dependencies in Java 9 code. When you're preparing to create *custom* runtimes, you also can use `jdeps` to determine your app's dependencies, so you know which modules to include. For example, this chapter's `Welcome` app depends only on `java.base`. We can confirm that by executing the following command from the `WelcomeApp` folder, which checks the `com.deitel.welcome` module's dependencies:

```
jdeps --module-path jars -m com.deitel.welcome
```

This produces the following output, showing the packages and modules the app uses:

```
com.deitel.welcome
 [file:///C:/examples/ch36/WelcomeApp/jars/com.deitel.welcome.jar]
   requires java.base (@9-ea)
com.deitel.welcome -> java.base
   com.deitel.welcome     -> java.io       java.base
   com.deitel.welcome     -> java.lang     java.base
```

The output shows that our module `com.deitel.welcome` depends on the `java.base` module, and that our module specifically uses types from the `java.base` module's `java.io` and `java.lang` packages.

The preceding command may also be written as

```
jdeps jars/com.deitel.welcome.jar
```

In addition, you can use `jdeps` on a specific `.class` file, as in:

```
jdeps mods/com.deitel.welcome/com/deitel/welcome/Welcome.class
```

which produces

```
Welcome.class -> java.base
   com.deitel.welcome     -> java.io       java.base
   com.deitel.welcome     -> java.lang     java.base
```

Verbose `jdeps` Output

If you'd like more details, you can specify the `-v` (verbose) option as in:

```
jdeps -v jars/com.deitel.welcome.jar
```

which produces:

```
com.deitel.welcome
 [file:///C:/examples/ch36/WelcomeApp/jars/com.deitel.welcome.jar]
   requires java.base (@9-ea)
com.deitel.welcome -> java.base
   com.deitel.welcome.Welcome   -> java.io.PrintStream    java.base
   com.deitel.welcome.Welcome   -> java.lang.Object       java.base
   com.deitel.welcome.Welcome   -> java.lang.String       java.base
   com.deitel.welcome.Welcome   -> java.lang.System       java.base
```

showing precisely which packages, types and modules the app uses. Knowing that the app requires only `java.base`, we can then use `jlink` to create a custom runtime containing only that module, which we'll do in Section 36.8.

Using `jdeps` to Produce DOT Files for Graphing Tools

You can use graphing tools—such as Graphviz (`www.graphviz.org`) and its web-based version (`www.webgraphviz.com`)—to produce module-dependency graphs using the DOT graph description language,[21] which specifies a graph's nodes and edges. The `jdeps` tool can create DOT (`.dot`) files with the `--dot-output` option as in:

```
jdeps --dot-output . jars/com.deitel.welcome.jar
```

which produces two `.dot` files in the current folder (`.`):

- `summary.dot`—the description of module `com.deitel.welcome`'s dependencies.
- `com.deitel.welcome.dot`—the description of module `com.deitel.welcome`'s specific package dependencies.

Figure 36.19 shows the graph we produced by opening `summary.dot` in a text editor, then copying and pasting its contents

21. `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`.

```
digraph "summary" {
  "com.deitel.welcome" -> "java.base (java.base)";
}
```

into the textbox at `webgraphviz.com` and clicking **Generate Graph**.[22]

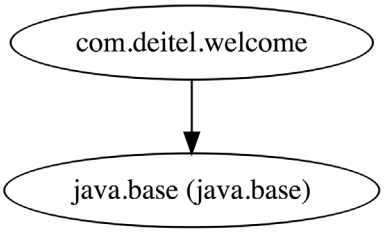


Fig. 36.19 | `Webgraphviz.com` graph based on `summary.dot`

Additional `jdeps` *Options*

For a complete list of `jdeps` options, visit

```
http://download.java.net/java/jdk9/docs/technotes/tools/windows/
jdeps.html
```

for Windows or visit

```
http://download.java.net/java/jdk9/docs/technotes/tools/unix/
jdeps.html
```

for macOS and Linux.

36.7 Resources in Modules; Using an Automatic Module

When the types in a module require resources—such images, videos, XML documents and more—those resources should be packaged with the module to ensure that they're available when the module's types are used at execution time. This is known as **resource encapsulation**.[23] In this section, we'll migrate our non-modularized JavaFX `VideoPlayer` example from Section 22.6 into a module that also encapsulates the app's resources—the FXML file that describes the GUI and its video file that will be loaded and played at execution time. By convention, resources typically are placed in a folder named `res`.

Recall that the original `VideoPlayer` example consisted of the following files all in Chapter 22's `VideoPlayer` folder:

- `VideoPlayer.xml`—The FXML file that describes the app's GUI.
- `VideoPlayer.java`—The `Application` subclass that begins the app's execution.
- `VideoPlayerController.java`—The controller class that responds to the GUIs events and loads the video.

22. The `.dot` extension is also used by Microsoft Word document templates. On systems with Microsoft Word installed, open the `jdeps`-produced `.dot` files directly from a text editor.
23. "Java Platform Module System Requirements." `http://openjdk.java.net/projects/jigsaw/spec/reqs/#resource-encapsulation`

- `sts117.mp4`—The NASA video[24] that the app loads and plays.
- `controlsfx-8.40.12`—The ControlsFX library containing the dialog class `ExceptionDialog`. We display an `ExceptionDialog` if the `MediaPlayer` encounters any errors.

Reorganizing for Modularization

For the purpose of this example, we reorganized the files into the folder structure shown in Fig. 36.20 to support modules. Notice the following about the structure:

- The files `VideoPlayer.fxml` and `sts117.mp4`, which are not Java source code files, are located in the module directory's `res` folder. These files will be read from the module's `res` folder when the app executes.
- As required for modularization, we placed the classes `VideoPlayer` and `VideoPlayerController` in a package—the folder structure `com/deitel/videoplayer` corresponds to the package `com.deitel.videoplayer`.
- As required, we created a `module-info.java` file in the module's root folder.

In addition, we renamed `controlsfx-8.40.12.jar` to `controlsfx.jar` and placed it directly in the `VideoPlayer` folder's `mods` subfolder.

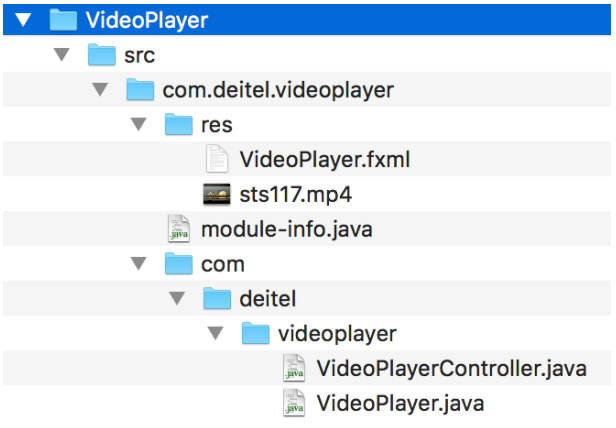


Fig. 36.20 | Modularized `VideoPlayer` `src` folder structure.

36.7.1 Automatic Modules

The ControlsFX library we used when developing the `VideoPlayer` in Section 22.6 was not designed to be a Java module. However, you can add *any* library's JAR file to an app's module path, then use the packages in that JAR. When you do, the JAR file implicitly becomes an **automatic module** and can be specified in a `module` declaration's `requires` directives. The JAR's file name—minus the `.jar` extension—becomes its module name, which *must be a valid Java identifier* for use in a `requires` directive. This is why we renamed the JAR by removing `-8.40.12` from the original filename. Also, an automatic module:

24. For NASA's terms of use, visit `http://www.nasa.gov/multimedia/guidelines/`.

- *implicitly* `exports` all of its packages—so, any module that reads the automatic module has access to the `public` types in the automatic module's packages.
- *implicitly reads* all other modules in the app, including other automatic modules—so, an automatic module has access to all the `public` types exposed by the system's other modules.

Code Changes for Modularization

We made the following code changes:

- `VideoPlayer.fxml`—We modified the controller class's name to use its fully qualified name `com.deitel.videoplayer.VideoPlayerController` so that the `FXMLLoader` can find the controller class.
- `VideoPlayer.java`—We changed the name of the FXML file to load from `"VideoPlayer.fxml"` to `"/res/VideoPlayer.fxml"`, which indicates that the FXML file is located in the module's `res` folder. We also added the `package` statement

```
package com.deitel.videoplayer;
```

- `VideoPlayerController.java`—We modified the name of the video file from `"sts117.mp4"` to `"/res/sts117.mp4"`, which indicates that the video file is located in the module's `res` folder. We also added the `package` statement

```
package com.deitel.videoplayer;
```

The rest of the code is identical to what we presented in Section 22.6.

36.7.2 Requiring Multiple Modules

The `com.deitel.videoplayer` module declaration (Fig. 36.21) indicates that the module requires `javafx.controls`, `javafx.fxml`, `javafx.media` and `controlsfx` (the automatic module discussed in Section 36.7.1). The module `exports` the `com.deitel.videoplayer` package (line 9), because class `VideoPlayerController` is used by class `FXMLLoader` (module `javafx.fxml`) when it creates the controller object and the app's GUI.

```
1  // Fig. 36.21: module-info.java
2  // Module declaration for the com.deitel.videoplayer module
3  module com.deitel.videoplayer {
4     requires javafx.controls;
5     requires javafx.fxml;
6     requires javafx.media;
7     requires controlsfx; // automatic module for ControlsFX
8
9     exports com.deitel.videoplayer;
10    opens com.deitel.videoplayer to javafx.fxml;
11 }
```

Fig. 36.21 | Module declaration for the `com.deitel.videoplayer` module.

36.7.3 Opening a Module for Reflection

In Fig. 36.21, the `opens…to` directive (line 10) indicates that the accessible types in the package `com.deitel.videoplayer` should be available via *reflection* at runtime to types in

the `javafx.fxml` module. As we discussed in Section 36.2.6, this enables the `FXMLLoader` to locate and load class `VideoPlayerController`. The `FXMLLoader` the creates a `VideoPlayerController` object and *injects* into it references to the GUIs components that the `FXMLLoader` creates from the app's FXML file. For one module to *open* a package to another module, that package must first be exported (possibly as a qualified export using `exports...to`).

36.7.4 Module-Dependency Graph

Figure 36.22 shows the `com.deitel.videoplayer` module-dependency graph. Again, the ones with light blue highlights are explicitly specified in requires directives—except for `java.base`, which is implicitly required by all modules. The other modules shown are dependencies of the modules specified in the `requires` directives.

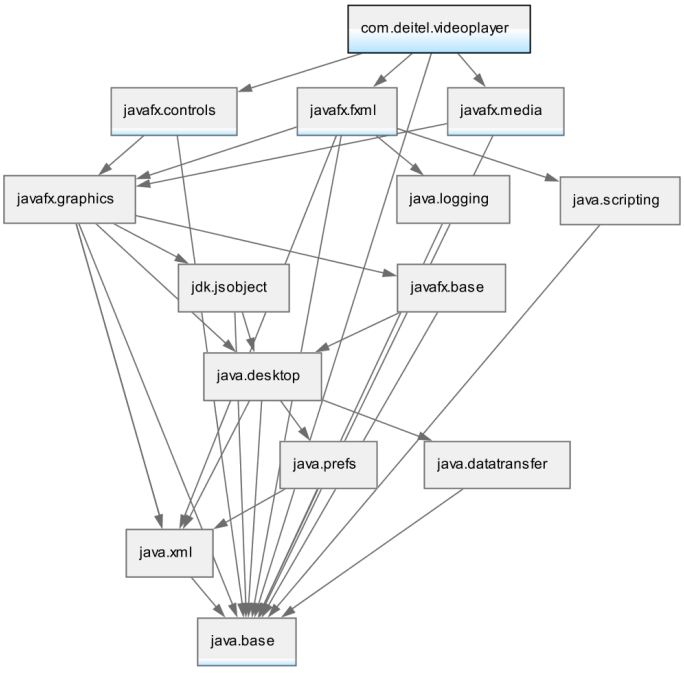


Fig. 36.22 | `com.deitel.videoplayer` module-dependency graph.

36.7.5 Compiling the Module

To compile the `com.deitel.videoplayer` module, type:

```
javac --module-path mods -d mods/com.deitel.videoplayer ^
   src/com.deitel.videoplayer/module-info.java ^
   src/com.deitel.videoplayer/com/deitel/videoplayer/*.java
```

Note that we included the `--module-path` option, because the `mods` folder contains `controlsfx.jar`—the automatic module that is required to compile this app.

Copying the Resource Files into the Module

Though some IDEs and build tools will automatically put the module's resources into the compiled module, the preceding `javac` command does not. Once you've compiled the module, copy the `res` folder from this project's `src/com.deitel.videoplayer` folder into the `mods/com.deitel.videoplayer` folder.

36.7.6 Running a Modularized App

To execute class `VideoPlayer` from the `com.deitel.videoplayer` module, type:

```
java --module-path mods ^
   -m com.deitel.videoplayer/com.deitel.videoplayer.VideoPlayer
```

Figure 36.23 shows the app executing on Windows.

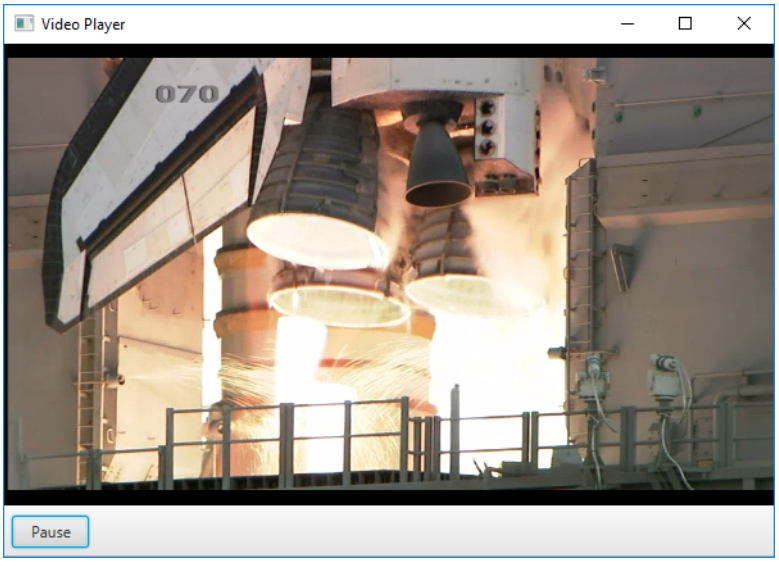


Fig. 36.23 | Modularized `VideoPlayer` app executing.

36.8 Creating Custom Runtimes with `jlink`

A new tool in JDK 9 is the **`jlink` command**—Java's linker for creating custom runtime images.[25] In a custom runtime, you can include just what's necessary for a given app or set of apps to execute. For example, if you're creating a runtime for a device that does not support GUIs, you can create a runtime without the corresponding modules that support Swing and JavaFX. In fact, many of this book's text-only, command-line examples can execute on a runtime that contains only the `java.base` module.

25. Denise, Jean-Francois. "JEP 282: `jlink`: The Java Linker." `http://openjdk.java.net/jeps/282`.

36.8.1 Listing the JRE's Modules

With modularization the JRE is a proper subset of the JDK.[26] If you run the command:

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

from the JRE's `bin` folder, the result contains only the JRE's 73 modules (Fig. 36.24), rather than the full listing of the JDK's 95 modules. This number will change as Java evolves. In Section 36.8.3, we do this on a custom runtime produced with the `jlink` command—in that case, only the single module bundled with that runtime will be displayed.

Software Engineering Observation 36.3

You can use the modularized Java platform to conveniently form custom runtimes for smaller capacity devices.

```
java.activation@9-ea
java.base@9-ea
java.compiler@9-ea
java.corba@9-ea
java.datatransfer@9-ea
java.desktop@9-ea
java.instrument@9-ea
java.jnlp@9-ea
java.logging@9-ea
java.management@9-ea
java.management.rmi@9-ea
java.naming@9-ea
java.prefs@9-ea
java.rmi@9-ea
java.scripting@9-ea
java.se@9-ea
java.se.ee@9-ea
java.security.jgss@9-ea
java.security.sasl@9-ea
java.smartcardio@9-ea
java.sql@9-ea
java.sql.rowset@9-ea
java.transaction@9-ea
java.xml@9-ea
java.xml.bind@9-ea
java.xml.crypto@9-ea
java.xml.ws@9-ea
java.xml.ws.annotation@9-ea
javafx.base@9-ea
javafx.controls@9-ea
javafx.deploy@9-ea
javafx.fxml@9-ea
javafx.graphics@9-ea
javafx.media@9-ea
javafx.swing@9-ea
javafx.web@9-ea
jdk.accessibility@9-ea
jdk.charsets@9-ea
jdk.crypto.cryptoki@9-ea
jdk.crypto.ec@9-ea
jdk.crypto.mscapi@9-ea
jdk.deploy@9-ea
jdk.deploy.controlpanel@9-ea
jdk.dynalink@9-ea
jdk.httpserver@9-ea
jdk.incubator.httpclient@9-ea
jdk.internal.le@9-ea
jdk.internal.vm.ci@9-ea
jdk.javaws@9-ea
jdk.jdwp.agent@9-ea
jdk.jfr@9-ea
jdk.jsobject@9-ea
jdk.localedata@9-ea
jdk.management@9-ea
jdk.management.agent@9-ea
jdk.naming.dns@9-ea
jdk.naming.rmi@9-ea
jdk.net@9-ea
jdk.pack@9-ea
jdk.plugin@9-ea
jdk.plugin.dom@9-ea
jdk.plugin.server@9-ea
jdk.scripting.nashorn@9-ea
jdk.scripting.nashorn.shell@9-ea
jdk.sctp@9-ea
jdk.security.auth@9-ea
jdk.security.jgss@9-ea
jdk.snmp@9-ea
jdk.unsupported@9-ea
jdk.xml.dom@9-ea
jdk.zipfs@9-ea
oracle.desktop@9-ea
oracle.net@9-ea
```

Fig. 36.24 | Output of `java --list-modules` showing the modules that compose the JRE.

26. Brian Goetz, e-mail message to authors, March 20, 2017.

36.8.2 Custom Runtime Containing Only java.base

For the purpose of this example, change to the WelcomeApp folder—after creating the custom runtime, you'll execute the Welcome app using it. The following command creates a runtime containing only the module java.base:

```
jlink --module-path "%JAVA_HOME%"/jmods --add-modules java.base ^
   --output javabaseruntime
```

The commands options are as follows:

- --module-path specifies one or more folders in which to locate the modules that will be included in the runtime—in this case, the JDK's jmods folder, which contains the modular JAR files for all of the JDK's modules.
- --add-modules specifies which modules to include in the runtime—in this case, just java.base.
- --output specifies the folder in which the runtime's contents are placed—in this case, the folder javabaseruntime. This folder will be placed in the folder from which you execute the preceding command (unless you specify additional path information). If the folder already exists, an error occurs.

This runtime can execute an app that depends only on types from the packages in module java.base, including many of this book's command-line apps.

Note Regarding the JAVA_HOME Variable

The JAVA_HOME environment variable must refer to JDK 9's installation folder on your system—see the Before You Begin section before the preface for information on configuring this environment variable. On Windows, you specify %JAVA_HOME% to use JAVA_HOME's value in a command. Linux and macOS users should replace %JAVA_HOME% with $JAVA_HOME. So, for example, the preceding command on Linux and macOS would be:

```
jlink --module-path "$JAVA_HOME"/jmods --add-modules java.base \
   --output javabaseruntime
```

In either case, if the path contains spaces, place the environment variable in quotes ("").

Executing the Welcome App Using This Custom Runtime

To run the app with the custom runtime, on Windows use:

```
javabaseruntime\bin\java --module-path mods ^
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

or on macOS/Linux use:

```
javabaseruntime/bin/java --module-path mods \
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

The program executes and displays:

```
Welcome to the Java Platform Module System!
```

Listing the Modules in a Custom Runtime

Previously we used the command

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

to list all the modules in the JDK. Once you have a custom runtime, you can use the `java` command from the custom runtime's `bin` folder to confirm the modules it includes, as in:

```
javabaseruntime\bin\java --list-modules
```

When executing the custom runtime's `java` command, use `\` to separate folder names on Windows and `/` to separate the folder names on macOS and Linux. The preceding command produces the following output:

```
java.base@9-ea
```

Similarly the following command creates a custom runtime containing only the module `java.desktop` and any other modules on which it depends:

```
jlink --module-path "%JAVA_HOME%"/jmods ^
   --add-modules java.desktop --output javadesktopruntime
```

For this custom runtime, running

```
javadesktopruntime\bin\java --list-modules
```

(again, use forward slashes on macOS and Linux) produces the following output

```
java.base@9-ea
java.datatransfer@9-ea
java.desktop@9-ea
java.prefs@9-ea
java.xml@9-ea
```

36.8.3 Creating a Custom Runtime for the `Welcome` App

To create a custom runtime containing only the modules `com.deitel.welcome` and its dependencies (in this case, `java.base`), use:

```
jlink --module-path jars;"%JAVA_HOME%"/jmods ^
   --add-modules com.deitel.welcome --output welcomeruntime
```

This creates a custom runtime in the folder `welcomeruntime`. The preceding command specifies multiple folders—`jars` and `%JAVA_HOME%`. On Windows, the path-separator character for lists of folders is a semicolon (`;`). Linux and macOS users should replace the semicolons in the commands with the colon (`:`) path-separator character, as in

```
jlink --module-path jars:"$JAVA_HOME"/jmods \
   --add-modules com.deitel.welcome --output welcomeruntime
```

To see the list of modules included in the custom runtime, on Windows use:

```
welcomeruntime\bin\java --list-modules
```

(again, use forward slashes on macOS and Linux) which produces the following list of modules:

```
com.deitel.welcome
java.base@9-ea
```

36.8.4 Executing the `Welcome` App Using a Custom Runtime

To run the app with the custom runtime, on Windows use:

```
welcomeruntime\bin\java -m com.deitel.welcome
```

(Again, use forward slashes on macOS and Linux.) The program executes and displays:

```
Welcome to the Java Platform Module System!
```

36.8.5 Using the Module Resolver on a Custom Runtime

When you run a modularized app, the JVM uses a **module resolver** to determine which modules are required at execution time and ensure that their dependencies are satisfied—this is known as the **transitive closure** of those modules. To locate modules, the module resolver looks at the **observable modules**—that is, those built into the runtime (like `java.base`) and those located on the module path. For a required module that cannot be found, the runtime throws a `java.lang.module.FindException`.

For a given app and runtime, you can view the steps the module resolver follows to determine module dependencies and ensure that the required modules are available to the program. To do so, include **`-Xdiag:resolver` option**[27] in the `java` command, as in:

```
welcomeruntime\bin\java -Xdiag:resolver -m com.deitel.welcome
```

(Again, use forward slashes on macOS and Linux.) This uses the custom `welcomeruntime`'s `java` command to display the resolver's steps for locating modules, followed by the program's output:

```
[Resolver] Root module com.deitel.welcome located
[Resolver]   (jrt:/com.deitel.welcome)
[Resolver] Module java.base located, required by com.deitel.welcome
[Resolver]   (jrt:/java.base)
[Resolver] Result:
[Resolver]   com.deitel.welcome
[Resolver]   java.base
Welcome to the Java Platform Module System!
```

The module-resolution process for the `Welcome` app proceeds as follows:

1. First, the resolver locates the app's **initial module**—`com.deitel.welcome`—containing the app's entry point. The resolver refers to this as the *root module*. This is the root node in the module-dependency graph.
2. Next, the resolver locates `java.base`, because the `com.deitel.welcome` module descriptor specifies that `com.deitel.welcome` requires `java.base`.
3. Since `java.base` does not depend on other modules, the dependency graph is now complete and the resolver displays the resulting list of modules required to execute the program.

Next, the program executes and displays its output. If a required module were not found during this process, a `java.lang.module.FindException` would be displayed in this output and the program would not execute.

36.9 Services and ServiceLoader

In Section 10.13, we discussed "programming to an interface, not an implementation" as a mechanism for creating loosely coupled objects. We'll use these concepts in this section

27. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." `http://openjdk.java.net/jeps/261`.

as we introduce services and class `ServiceLoader`, which help you create loosely coupled system components. This can make large-scale systems easier to develop and maintain.

`MathTutor` App

We'll develop a `MathTutor` app (consisting of three modules) that supports various types of randomly generated math problems. Rather than hard-coding these into the app, we'll load math problems through a *service-provider interface* that describes how to obtain math problems. We'll then define two *service providers*—classes that implement this interface. One service provider will create addition problems and the other multiplication problems. At runtime, we'll load objects of these service-provider implementation classes and use them. The completed app structure consisting of three modules is shown in Fig. 36.25.

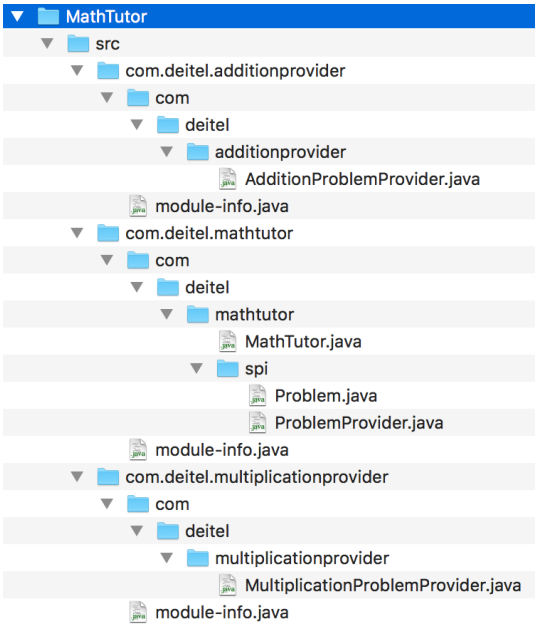


Fig. 36.25 | Folder structure for the `MathTutor` app's modules.

`MathTutor` App's Modules

Module `com.deitel.mathtutor` aggregates two related packages:

- `com.deitel.mathtutor`: This package contains class `MathTutor`—a command-line app that displays random math problems to the user, inputs the user's responses and displays whether each response is correct or incorrect.
- `com.deitel.mathtutor.spi`: This package contains the `ProblemProvider` service-provider interface and the supporting `abstract` class `Problem`, which represents a math problem. Class `MathTutor` uses `ProblemProviders` to obtain `Problem` objects.

Module `com.deitel.additionprovider` contains a package of the same name in which we declare class `AdditionProblemProvider`. This implementation of the service-provider interface `ProblemProvider` generates random addition `Problems`.

Module com.deitel.multiplicationprovider contains a package of the same name in which we declare class MultiplicationProblemProvider. This implementation of the service-provider interface ProblemProvider generates random multiplication Problems.

How We'll Demonstrate the App

We'll initially run the MathTutor app without placing the service-provider implementation modules on the module path to demonstrate what happens when *no* service providers are found at runtime. Next, we'll "plug in" the module com.deitel.additionprovider on the module path, then re-run the app to demonstrate that we're able to obtain Problems from an AdditionProblemProvider. Finally, we'll "plug in" both the com.deitel.additionprovider and com.deitel.multiplicationprovider modules on the module path, then re-run the app to demonstrate that we're able to obtain Problems generated by both an AdditionProblemProvider and a MultiplicationProblemProvider.

Plug-in Architecture

This "plug-in" architecture using a service-provider interface and multiple service-provider implementations makes the MathTutor app easy to extend. Simply create a module containing a ProblemProvider implementation, then add it to the module path when you run the app. It also makes the app more configurable, because you can choose which modules to include on the module path when you execute the app.

Reliable Configuration

The mechanisms for creating loosely coupled systems like the MathTutor app have been used extensively in Java since its early versions. A key new concept in Java 9—which also applies to modules in general—is *reliable configuration*. For the MathTutor app to be able to display Problems to the user, it must be able to locate and load ProblemProvider implementations. As you'll see, module declarations enable you to specify which service-provider interfaces a module uses and whether a module contains types that implement those interfaces.

36.9.1 Service-Provider Interface

The package com.deitel.mathtutor.spi contains the com.deitel.mathtutor module's service-provider interface ProblemProvider and the supporting abstract class Problem. The final component of this package's name—spi—is commonly used in packages that declare one or more service-provider interfaces. Interface ProblemProvider (Fig. 36.26) declares method getProblem (line 6) that returns a Problem.

```
1  // Fig. 36.26: ProblemProvider.java
2  // Service-provider interface for obtaining a Problem
3  package com.deitel.mathtutor.spi;
4
5  public interface ProblemProvider {
6     public Problem getProblem();
7  }
```

Fig. 36.26 | Service-provider interface for obtaining a Problem.

Abstract class `Problem` (Fig. 36.27) provides the common features of math problems in this example. Each has two `int` operands and an `int` result as well as a `String` representing the operation—the `MathTutor` displays this `String` with each math problem it presents to the user. Class `Problem`'s `abstract` method `getResult` is overridden in each service-provider implementation's concrete subclass of `Problem`.

```
1  // Fig. 36.27: Problem.java
2  // Problem superclass that contains information about a math problem.
3  package com.deitel.mathtutor.spi;
4
5  public abstract class Problem {
6     private int leftOperand;
7     private int rightOperand;
8     private int result;
9     private String operation;
10
11    // constructor
12    public Problem(int leftOperand, int rightOperand, String operation) {
13       this.leftOperand = leftOperand;
14       this.rightOperand = rightOperand;
15       this.operation = operation;
16    }
17
18    // gets the leftOperand
19    public int getLeftOperand() {return leftOperand;}
20
21    // gets the rightOperand
22    public int getRightOperand() {return rightOperand;}
23
24    // gets the operation
25    public String getOperation() {return operation;}
26
27    // gets the result
28    public abstract int getResult();
29 }
```

Fig. 36.27 | `Problem` superclass that contains information about a math problem.

36.9.2 Loading and Consuming Service Providers

Class `MathTutor` (Fig. 36.28) is the app's entry point. It provides the logic for locating and loading `ProblemProvider` implementations, then using them to present math problems to the user.

```
1  // Fig. 36.28: MathTutor.java
2  // Math tutoring app using ProblemProviders to display math problems.
3  package com.deitel.mathtutor;
4
5  import java.util.List;
6  import java.util.Random;
```

Fig. 36.28 | Math tutoring app using `ProblemProvider`s to display math problems. (Part 1 of 3.)

```
import java.util.Scanner;
import java.util.ServiceLoader;
import java.util.ServiceLoader.Provider;
import java.util.stream.Collectors;
import com.deitel.mathtutor.spi.Problem;
import com.deitel.mathtutor.spi.ProblemProvider;

public class MathTutor {
   private static Scanner input = new Scanner(System.in);

   public static void main(String[] args) {
      // get a service loader for ProblemProviders
      ServiceLoader<ProblemProvider> serviceLoader =
        ServiceLoader.load(ProblemProvider.class);

      // get the list of service providers
      List<Provider<ProblemProvider>> providersList =
         serviceLoader.stream().collect(Collectors.toList());

      // check whether there are any providers
      if (providersList.isEmpty()) {
         System.out.println(
            "Terminating MathTutor: No problem providers found.");
         return;
      }

      boolean shouldContinue = true;
      Random random = new Random();

      do {
         // choose a ProblemProvider at random
         ProblemProvider provider =
            providersList.get(random.nextInt(providersList.size())).get();

         // get the Problem
         Problem problem = provider.getProblem();

         // display the problem to the user
         showProblem(problem);
      } while (playAgain());
   }

   // show the math problem to the user
   private static void showProblem(Problem problem) {
      String problemStatement = String.format("What is %d %s %d? ",
         problem.getLeftOperand(), problem.getOperation(),
         problem.getRightOperand());

      // display problem and get answer from user
      System.out.printf(problemStatement);
      int answer = input.nextInt();

```

Fig. 36.28 | Math tutoring app using `ProblemProvider`s to display math problems. (Part 2 of 3.)

```
59          while (answer != problem.getResult()) {
60             System.out.println("Incorrect. Please try again: ");
61             System.out.printf(problemStatement);
62             answer = input.nextInt();
63          }
64
65          System.out.println("Correct!");
66       }
67
68       // play again?
69       private static boolean playAgain() {
70          System.out.printf("Try another? y to continue, n to terminate: ");
71          String response = input.next();
72
73          return response.toLowerCase().startsWith("y");
74       }
75    }
```

Fig. 36.28 | Math tutoring app using `ProblemProviders` to display math problems. (Part 3 of 3.)

Using `ServiceLoader` *to Locate Service Providers*

Lines 19–20

```
ServiceLoader<ProblemProvider> serviceLoader =
   ServiceLoader.load(ProblemProvider.class);
```

create a **`ServiceLoader`** (package `java.util`) that loads `ProblemProvider` implementations. `ServiceLoader`'s `static` **`load`** method receives as its argument the `Class` object representing the service-provider interface's type—`ProblemProvider.class` is a **class literal** that's equivalent to creating a `Class<ProblemProvider>` object, as in:

```
new Class<ProblemProvider>()
```

Method `load` returns a `ServiceLoader<ProblemProvider>` that knows only how to load `ProblemProvider` implementations.

There are several ways to get service-provider implementations from a `ServiceLoader`. In lines 23–24

```
List<Provider<ProblemProvider>> providersList =
   serviceLoader.stream().collect(Collectors.toList());
```

we obtain a `List` of the available service-provider implementations using `ServiceLoader`'s **`stream`** method. This returns a `Stream<Provider<ProblemProvider>>` representing all the available `ProblemProvider` implementations, if any. Interface `Provider` (imported at line 9) is a nested type of class `ServiceLoader`. For each available `ProblemProvider` implementation, the stream contains one `Provider<ProblemProvider>` object. Line 24 uses `Stream` method `collect` and the predefined `Collector` defined by `Collectors.toList` to get the `List` containing all the available implementations. If that `List` is empty (line 27) the program displays an appropriate message and terminates.

Using a Service-Provider Interface

If the `List` contains any service-provider implementations, lines 36–46 use them to display one math problem at a time to the user. Lines 38–39

```
ProblemProvider provider =
   providersList.get(random.nextInt(providersList.size())).get();
```

randomly select one `Provider<ProblemProvider>` object from the `providersList`, then invoke that object's `get` method to obtain its `ProblemProvider`. Line 42

```
Problem problem = provider.getProblem();
```

then gets a `Problem` from whichever `ProblemProvider` was selected.

Note the *loose coupling* of the `MathTutor` app and its `ProblemProviders`. The app does not refer in any way to `AdditionProblemProviders` or `MultiplicationProblemProviders` that generate math problems.

36.9.3 uses Module Directive and Service Consumers

Figure 36.29 shows the `com.deitel.mathtutor` module declaration. Note that this module `exports` the package `com.deitel.mathtutor.spi` containing the service-provider interface `ProblemProvider` and its supporting `Problem` class. This enables modules that implement interface `ProblemProvider` to access those types. The new feature in this declaration is the **uses module directive** (line 6). This directive indicates that there is a type in the `com.deitel.mathtutor` module that *uses* objects which implement the `ProblemProvider` interface. Such a module is called a **service consumer**.

```
1  // Fig. 36.29: module-info.java
2  // Module declaration for the com.deitel.mathtutor module
3  module com.deitel.mathtutor {
4     exports com.deitel.mathtutor.spi; // package for provider interface
5
6     uses com.deitel.mathtutor.spi.ProblemProvider;
7  }
```

Fig. 36.29 | Module declaration for the `com.deitel.mathtutor` module.

To be able to consume `ProblemProviders`, the `ServiceLoader` must be able to locate and load their implementations dynamically using Java's *reflection* capabilities. When you run this app, the module resolver will see in the descriptor that this module *uses* `ProblemProvider` implementations and thus is dependent on such providers. It will then search the modules on the module path looking for any modules that provide implementations of this interface. If it finds any such modules, it will add them to the module-dependency graph.

36.9.4 Running the App with No Service Providers

To compile the `com.deitel.mathtutor` module, type:

```
javac -d mods/com.deitel.mathtutor ^
   src/com.deitel.mathtutor/module-info.java ^
   src/com.deitel.mathtutor/com/deitel/mathtutor/MathTutor.java ^
   src/com.deitel.mathtutor/com/deitel/mathtutor/spi/*.java
```

Next, run the app with no `ProblemProvider` implementations on the module path by using the following `java` command, which places only the `com.deitel.mathtutor` module on the module path:

```
java --module-path mods/com.deitel.mathtutor ^
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The result is

```
Terminating MathTutor: No problem providers found.
```

36.9.5 Implementing a Service Provider

Next, let's create class `AdditionProblemProvider` (Fig. 36.30), which implements the service-provider interface `ProblemProvider` (line 10). This class's `com.deitel.additionprovider` package will be placed in the `com.deitel.additionprovider` module (Section 36.9.6). We `import` interface `ProblemProvider` and class `Problem` from the `com.deitel.mathtutor` module's exported package `com.deitel.mathtutor.spi` (lines 7–8). When the `MathTutor` calls an `AdditionProblemProvider`'s `getProblem` method (lines 14–23), the method creates an anonymous subclass of `Problem` (lines 16–22), passing to `Problem`'s constructor two random `int` values as the operands and the `String` `"+"` as the operation. Lines 18–21 override superclass `Problem`'s `getResult` method to return the sum of the left and right operands.

```
1  // Fig. 36.30: AdditionProblemProvider.java
2  // AdditionProblemProvider implementation of interface
3  // ProblemProvider for the MathTutor app.
4  package com.deitel.additionprovider;
5
6  import java.util.Random;
7  import com.deitel.mathtutor.spi.Problem;
8  import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class AdditionProblemProvider implements ProblemProvider {
11    private static Random random = new Random();
12
13    // returns a new addition problem
14    @Override
15    public Problem getProblem() {
16       return new Problem(random.nextInt(10), random.nextInt(10), "+") {
17          // override getResult to add the operands
18          @Override
19          public int getResult() {
20             return getLeftOperand() + getRightOperand();
21          }
22       };
23    }
24 }
```

Fig. 36.30 | `AdditionProblemProvider` implementation of interface `ProblemProvider` for the `MathTutor` app.

36.9.6 provides...with Module Directive and Declaring a Service Provider

Figure 36.31 shows the `com.deitel.additionprovider` module declaration. Note that this module `requires` the module `com.deitel.mathtutor`. Recall from Fig. 36.29 that this module `exports` the package `com.deitel.mathtutor.spi` containing the types used in class `AdditionProblemProvider`. The new feature in this module declaration is the **`provides...with` module directive**. Lines 6–7 specify that this module

- `provides` an implementation of interface `ProblemProvider`—declared in the `com.deitel.mathtutor` module's `com.deitel.mathtutor.spi` package
- `with` class `AdditionProblemProvider`—declared in this module's `com.deitel.additionprovider` package.

Such a module is called a **service provider**. The directive's `provides` part is followed by the name of an interface or `abstract` class that's specified in a module's `uses` directive. The directive's `with` part is followed by the name of a class that `implements` the interface or `extends` the `abstract` class.

```
1  // Fig. 36.31: module-info.java
2  // Module declaration for the com.deitel.additionprovider module
3  module com.deitel.additionprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider
7        with com.deitel.additionprovider.AdditionProblemProvider;
8  }
```

Fig. 36.31 | Module declaration for the `com.deitel.additionprovider` module.

36.9.7 Running the App with One Service Provider

Next, we'll run the app with the `AdditionProblemProvider` included in the module path. First, compile the `com.deitel.additionprovider` module, as follows:

```
javac --module-path mods -d mods/com.deitel.additionprovider ^
   src/com.deitel.additionprovider/module-info.java ^
   src/com.deitel.additionprovider/com/deitel/additionprovider/ ^
      AdditionProblemProvider.java
```

Then run the app with the following `java` command:

```
java --module-path mods ^
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The following sample output shows addition problems:

```
What is 9 + 6? 15
Correct!
Try another? y to continue, n to terminate: y
What is 2 + 6? 7
Incorrect. Please try again:
What is 2 + 6? 8
Correct!
Try another? y to continue, n to terminate: n
```

36.9.8 Implementing a Second Service Provider

Class `MultiplicationProblemProvider` (Fig. 36.32) also implements the service-provider interface `ProblemProvider` (line 10). This class's `com.deitel.multiplicationprovider` package will be placed in the `com.deitel.multiplicationprovider` module (Fig. 36.33). Class `MultiplicationProblemProvider` is nearly identical to class `AdditionProblemProvider`, except that line 16 passes the `String` `"*"` for the `Problem`'s operation and the overridden `Problem` method `getResult` returns the product of the left and right operands.

```
1  // Fig. 36.32: MultiplicationProblemProvider.java
2  // MultiplicationProblemProvider implementation of interface
3  // ProblemProvider for the MathTutor app.
4  package com.deitel.multiplicationprovider;
5
6  import java.util.Random;
7  import com.deitel.mathtutor.spi.Problem;
8  import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class MultiplicationProblemProvider implements ProblemProvider {
11    private static Random random = new Random();
12
13    // returns a new addition problem
14    @Override
15    public Problem getProblem() {
16       return new Problem(random.nextInt(10), random.nextInt(10), "*") {
17          // override getResult to add the operands
18          @Override
19          public int getResult() {
20             return getLeftOperand() * getRightOperand();
21          }
22       };
23    }
24 }
```

Fig. 36.32 | `MultiplicationProblemProvider` implementation of interface `ProblemProvider` for the `MathTutor` app.

Figure 36.33 shows the `com.deitel.multiplicationprovider` module declaration. Again, this module `requires` the module `com.deitel.mathtutor`. Lines 6–7 specify that this module `provides` an implementation of the `ProblemProvider` interface `with` class `MultiplicationProblemProvider`.

```
1  // Fig. 36.33: module-info.java
2  // Module declaration for the com.deitel.multiplicationprovider module
3  module com.deitel.multiplicationprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider with
7        com.deitel.multiplicationprovider.MultiplicationProblemProvider;
8  }
```

Fig. 36.33 | Module declaration for the `com.deitel.multiplicationprovider` module.

36.9.9 Running the App with Two Service Providers

Next, we'll run the app with both the `AdditionProblemProvider` and the `MultiplicationProblemProvider` included in the module path. First, compile the `com.deitel.multiplicationprovider` module, as follows:

```
javac --module-path mods ^
   -d mods/com.deitel.multiplicationprovider ^
   src/com.deitel.multiplicationprovider/module-info.java ^
   src/com.deitel.multiplicationprovider/com/deitel/
      multiplicationprovider/MultiplicationProblemProvider.java
```

Then run the app with the following `java` command:

```
java --module-path mods ^
   -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The following is a sample output showing both addition and multiplication problems:

```
What is 4 * 8? 20
Incorrect. Please try again:
What is 4 * 8? 32
Correct!
Try another? y to continue, n to terminate: y
What is 3 * 6? 18
Correct!
Try another? y to continue, n to terminate: y
What is 3 + 7? 10
Correct!
Try another? y to continue, n to terminate: y
What is 9 + 3? 12
Correct!
Try another? y to continue, n to terminate: n
```

36.10 Wrap-Up

In this chapter, we introduced Java 9's new Java Platform Module system. We introduced key modularity concepts you're likely to use when building large-scale systems.

You saw that all modules implicitly depend on `java.base`. You created module declarations that specify a module's dependencies (with the `requires` directive), which packages a module makes available to other modules (with the `exports` directive), services it offers (with the `provides...with` directive), services it consumes (with the `uses` directive) and to what other modules it allows reflection (with the `open` modifier and the `opens` and `opens...to` directives).

To help you visualize the dependencies among modules, we showed several module-dependency graphs that we created using the NetBeans IDE's JDK 9 support. We discussed the steps that the runtime's module resolver performs to ensure that a module's dependencies are fulfilled.

You used JDK 9's new `jlink` tool (the Java linker) to create smaller custom runtimes, then used them to execute apps. We discussed the module system's strong encapsulation and showed the steps required to explicitly allow runtime reflection via the `open` modifier or the `opens` and `opens...to` directives in a module declaration.

We discussed the enormous amount of non-modularized legacy code that will need to be migrated to modular Java 9, then showed how the unnamed module and automatic modules can help make migration straightforward. We used the `jdeps` tool to determine code dependencies among modules and showed how the tool can be used to check for uses of pre-Java-9 internal APIs (which are for the most part strongly encapsulated in Java 9).

Finally, we discussed services and service providers for building loosely coupled systems by using service-provider interfaces and implementations and the `ServiceLoader` class. We also demonstrated the `uses` and `provides...with` directives in module declarations to indicate that a module uses a service or provides a service implementation, respectively. In the next chapter, we discuss various additional Java 9 topics.