Java Module System 19

Chapter Topics

- Understanding the issues the modules address and the benefits they provide
- Understanding the most important goals of the Java Module System: strong encapsulation, reliable configuration, and better performance
- Understanding how the modular JDK is organized, and how to discover and use its salient modules
- Writing a module declaration that specifies the properties of a module: its module name, and the `requires` and `exports` directives in such a declaration
- Understanding the module graph of an application
- Differentiating between readability of a module and accessibility of `public` types in an exported package of a required module
- Understanding the accessibility rules for members of a `public` type whose package is exported
- Understanding how implied dependencies are introduced in the module graph by the `requires transitive` directive, and its implication
- Creating the directory structure of a modular application, where an exploded module contains the source code for all packages in a module
- Compiling and running a multi-module application
- Creating modular JARs as repositories of compiled modules
- Running an application from its modular JARs
- Implications of open modules and the `opens` directives for reflection at runtime
- Creating, loading, and using services with `provides-with` and `uses` directives: specifying a service interface, and implementing service provider, service locator, and service consumer
- Creating and executing runtime images
- Differentiating between different kinds of modules: unnamed, automatic, and named modules
- Applying code migration strategies to modularize plain code
- Exploring modular JARs: listing JAR contents, extracting JAR contents, listing observable modules, describing the module descriptor, and viewing module-level, package-level, and class-level dependencies
- Providing a summary of relevant options accepted by the JDK command-line tools (`javac`, `java`, `jar`, `jdeps`, and `jlink`)

Java SE 17 Developer Exam Objectives	
[7.1] Define modules and their dependencies, expose module content including for reflection. Define services, producers, and consumers	§19.1, p. 1163 *to* §19.5, p. 1179 §19.8, p. 1191 §19.9, p. 1196
[7.2] Compile Java code, produce modular and non-modular jars, runtime images, and implement migration using unnamed and automatic modules	§19.6, p. 1186

	§19.7, p. 1189 §19.10, p. 1204 *to* §19.14, p. 1218
Java SE 11 Developer Exam Objectives	
[7.1] Deploy and execute modular applications, including automatic modules	§19.2, p. 1164 §19.6, p. 1186 §19.7, p. 1189 §19.10, p. 1204 *to* §19.13, p. 1214
[7.2] Declare, use, and expose modules, including the use of services	§19.3, p. 1168 §19.4, p. 1177 §19.5, p. 1179 §19.8, p. 1191 §19.9, p. 1196

The *Java Platform Module System* (JPMS) provides the software engineering technology that makes it possible to efficiently build, deploy, maintain, and evolve very large software systems. It was successfully used to modularize the JDK before finally being released in Java 9.

This chapter discusses how the language, the JDK tools, and the runtime environment provide the support for creating modular applications in Java.

19.1 Making the Case for Modules

We begin by briefly examining the issues that modules address and the benefits they provide.

Stronger Encapsulation

Modules in Java provide a new level of encapsulation above packages. Whereas a package encapsulates classes and interfaces, a module encapsulates a set of packages and any other resources required by the module. An application is thus composed of reusable and related modules, where each module groups a set of packages and each package groups a set of classes and interfaces.

Fine-Grained Visibility of Public Types

The need for modules in Java arose from many concerns. For programming in the large, the concept of packages has many shortcomings, particularly in providing more fine-grained visibility of public types which are accessible in *all* packages, with no possibility of curtailing the accessibility of such a public type. A module can provide stronger encapsulation with a well-defined public API that other modules can use, and packages that are meant to be internal to a module are now guaranteed to be inaccessible to other modules.

Reusability, Maintainability, and Optimization

Applying the principles of encapsulation at each encapsulation level increases the reusability and aids the maintainability of the code, resulting in modules that can be optimized independently.

Reliable Configuration

Prior to modules, the Java language did not support any notion of *dependencies* between artifacts that comprise an application. Ad hoc use of JARs (Java Archives) proved inadequate for such relationships, as the compiler and the JVM saw the JAR files merely as containers of types and resources.

Now there is better encapsulation of modules in JARs, with one module in each JAR. The Java Module System explicitly utilizes the dependencies between modules to reason and maintain these dependencies both at compile time and at runtime, resulting in reliable application configurations. Before an application is launched, all modules needed by the application are resolved in order to avoid problems relating to missing modules or types during execution.

Better Performance

Modules in Java contribute to developing scalable systems with improved performance, as modules can be optimized independently. The module system guarantees that all classes of the same package are in the same module—that is, there are no *split packages*, and the modules describe explicit dependencies. The class loader thus knows which module to look into when loading a class, improving the startup time of applications.

Scalable Applications

As the packages in the Java API are now organized into modules, it is possible for an application to include only those modules that are needed, thus reducing the binary footprint of the application. The modular JDK now caters to customized Java builds that can be scaled to small devices, and can also be shipped with the runtime environment included.

Improved Security

Organizing an application in modules can also improve security as critical code can be strongly encapsulated in modules, potentially decreasing the attack surface of an application.

Backward Compatibility

Modules are not a mandatory requirement for an application to run in the modular JDK. However, the benefits of using modules for large-scale application development are substantial as mentioned in this section. The modular JDK provides support to incrementally migrate legacy code into modules.

19.2 The Modular JDK

Support for modules is provided by the Java language and the JDK. The concept of a *module* is incorporated into the language. Existing tools in the JDK, such as `javac`, `java`, `jar`, and `jdeps`, have been enhanced to aid module-based development. Support for modules provided by these tools is covered in later sections. This section provides an overview of how the modular JDK is organized and what it has to offer.

Reasons for Modularity

The need to modularize the JDK arose from many reasons, among them the following:

- A monolithic runtime library (epitomized by the `rt.jar` file) was not conducive to the maintainability and evolution of the Java ecosystem with its ever-increasing APIs.
- Bloat of obsolete technologies could not be addressed properly, and was often at the risk of breaking backward compatibility.
- Maintaining backward compatibility in the face of ever-evolving packages made this goal more difficult.
- The one-size-fits-all runtime library started to incur a penalty in terms of increased space, more memory at runtime, and a need for lower consumption of CPU resources.
- JARs as containers of Java code were inadequate to elicit relationships between packages, having no means to expose their dependencies other than by resorting to third-party solutions (e.g., Maven and OSGi).
- The class-path solution to specify the location of package repositories was not optimal in locating types, leading to unforeseen problems when loading them and often throwing an exception at runtime because some type was missing.

Goals of the Modular JDK

Specific goals of the modular JDK, not surprisingly, align with the benefits of modules extolled in §19.1, p. 1163.

- *Stronger encapsulation*: This results in well-defined communication boundaries between modules and stricter encapsulation of their internal implementation.
- *Reliable configuration*: Any combination of modules must satisfy all dependencies both at compile time and runtime.
- *Enhanced performance*: Modularized code is more amiable to program optimizations.
- *Improved scalability*: The evolution of the Java platform can continue in a modularized fashion, as modules can be worked on in parallel.
- *Better security*: Modularized code decreases the attack surface, and access through *reflection* is strictly controlled at runtime.

Overview of the Modular JDK

The monolithic library of the JDK has now been split into many modules, with each module defining a specific functionality of the JDK. The installation directory of the JDK, shown in Figure 19.1, gives an overview of how the JDK is organized. The path of the installation directory will vary depending on the OS platform.

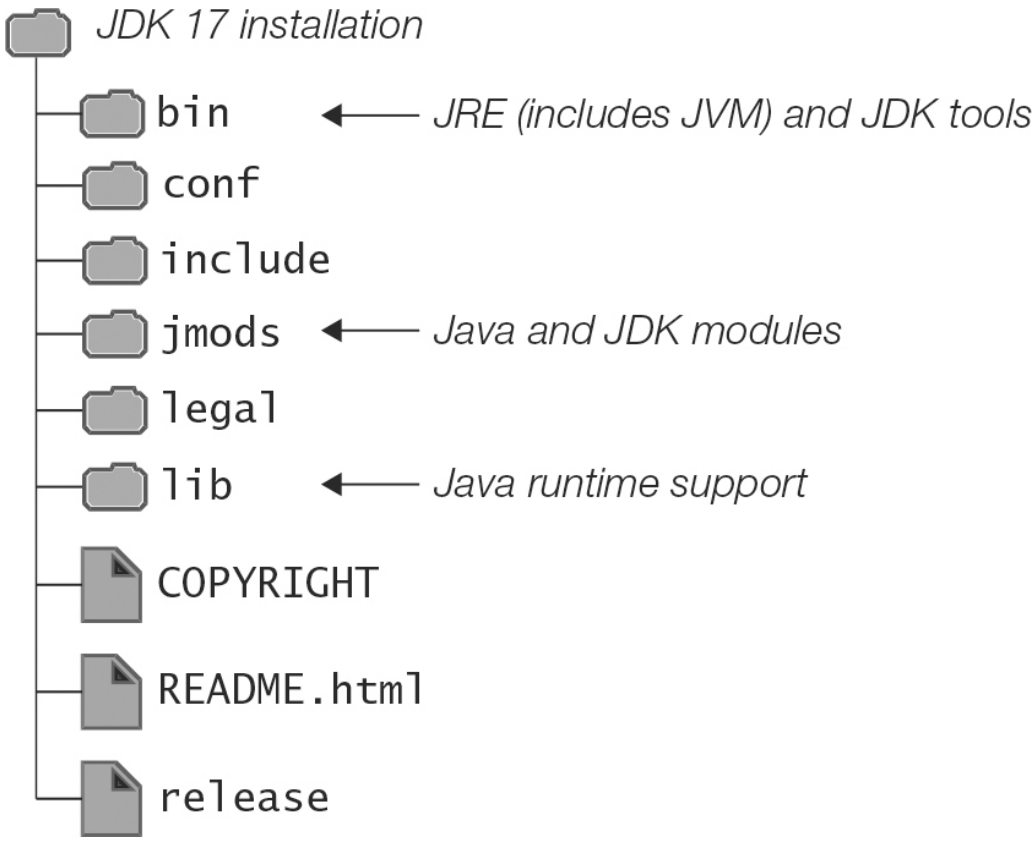


Figure 19.1 *Structure of JDK 17 Installation*

- The `bin` directory: This is where the executables are found—the Java Runtime Environment (JRE), which includes the Java Virtual Machine (JVM), and the JDK command-line tools, among others, that are necessary to develop and run Java programs.
- The `jmods` directory: This directory epitomizes the modular JDK. It contains the *platform modules* of the JDK. These are compiled modules used to create custom runtime images of applications.
  Each platform module is archived in a file having the name of the module and the extension `.jmod`—for example, `java.base.jmod`, `java.se.jmod`, and `jdk.jartool.jmod`. The modules beginning with the prefix `"java."` implement the Java SE specification, often called *standard modules*. These are present in all implementations of the Java SE platform. The modules beginning with the prefix `"jdk."` are platform specific, often called *non-standard modules*. These are not necessarily available for all implementations of the Java SE platform.
  JMOD archives use a special archive format reserved for the platform modules, that are created using the JDK tool `jmod`. All JDK tools understand JMOD archives. The platform modules can be explored like any other application module. Examples of exploring such modules can be found in §19.13, p. 1211. There is seldom any need to create a JMOD archive or change the contents of the `jmods` directory.
  Note that modules can be packaged both as JARs and as JMOD archives.
- The `lib` directory: This directory contains additional class libraries and support required by the JDK. These files are not meant for external use.

Module Graph of the JDK

The modules in the JDK define *dependencies* on other modules. These dependencies can be viewed as a graph in which the modules are the *nodes* and the *unidirectional edges* between the nodes define the dependencies between the modules. A partial *module graph* of the JDK is shown in Figure 19.2. The complete module graph of the JDK is of course substantially larger, but Figure 19.2 will suffice to convey its modular structure. It is also a *reduced graph* where by redundant dependencies, like the *implicit dependencies* on the `java.base` module (p. 1167), have been omitted.

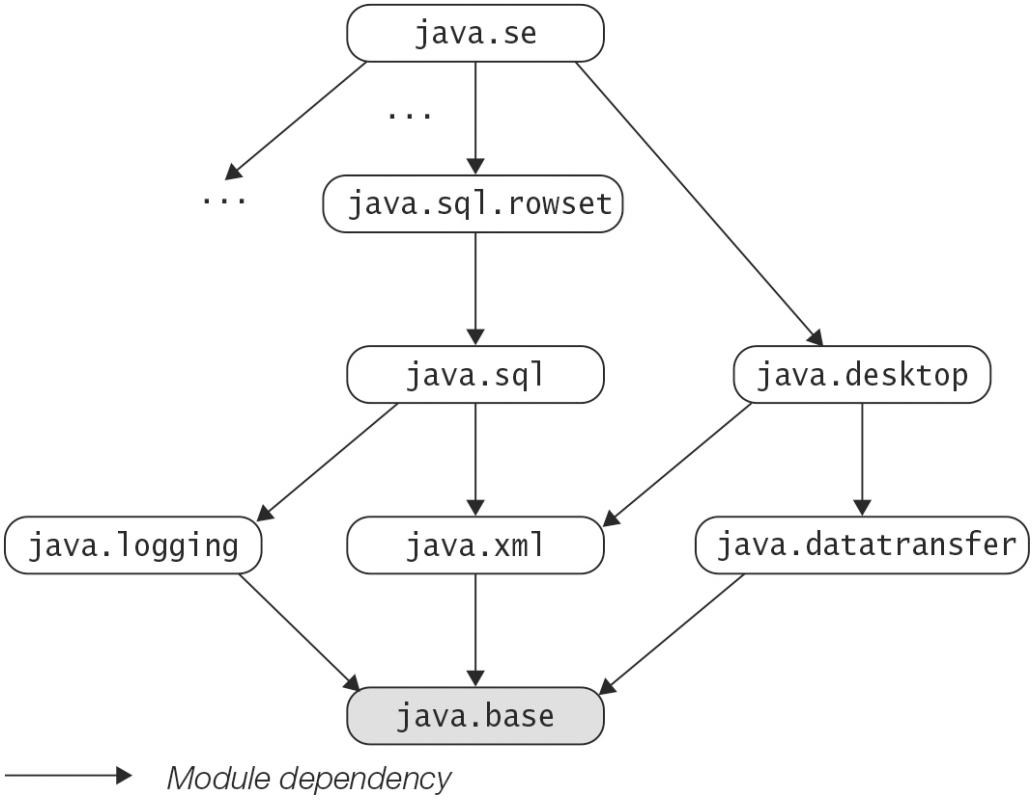


Figure 19.2 *Partial Module Graph of Java SE 17 Modules*

A unidirectional edge, for example, from the module `java.se` to the `java.desktop` module signifies that the former has a dependency on the latter. The (complete) module graph of the JDK allows us to read which modules any given module depends on. The structure of the module graph in **Figure 19.2** highlights two important modules: `java.se` at the zenith and `java.base` at the base of the graph, showing that the former depends on each module in the Java SE platform configuration and the latter on none of them.

Module graphs visualize the modular structure of an application, providing insights into its architecture and in discovering any anomalies. Such graphs can be built using the JDK tool `jdeps` (**p. 1214**). Other JDK tools, such as `javac` and `java`, perform analysis on the module graph of the application to ensure that all module dependencies are satisfied.

The `java.base` Module

The `java.base` module implements the *fundamental* APIs of the Java SE platform. It is necessary for any Java program to function, and deemed necessary for every configuration of the Java platform. Not surprisingly, it is required by every module in the JDK. In fact, *every* application module has an *implicit dependence* on the `java.base` module. However, it is seldom necessary for a module to explicitly specify a dependence on the `java.base` module.

Typically when constructing a module graph, dependency on the `java.base` module is only shown for a module that does not depend on any other module. Otherwise, this dependency is implicit.

The `java.base` module contains many important packages, such as `java.lang`, `java.util`, and `java.io`. The `java.lang` package, which contains the primordial class `java.lang.Object`, is imported automatically by every Java compilation unit.

Even though every module depends on the `java.base` module, this does not mean that all packages in it are also automatically *imported*. This only applies to the `java.lang` package. To use simple names of types in the source code, the relevant packages must be explicitly imported.

The `java.se` Module

The `java.se` module represents the *core* Java platform. It depends on each module in the Java SE platform configuration. Technically called an *aggregate module*, it does not contain any packages, but acts as a conduit to channel dependencies on other modules. Any module that depends on the `java.se` module would effectively imply that it depends on all modules of the Java SE platform. A typical scenario is for an application to depend on `java.base` to start with, and add dependencies to other modules in the JDK as and when needed.

19.3 Module Basics

Underlying the Java Module System is the concept of a *module* which is defined in the language and supported by the various JDK tools (such as `javac`, `java`, `jar`, and `jdeps`) and the Java runtime environment.

An understanding of packages is a prerequisite for understanding modules. The discussion of packages in §6.3, p. 326, is therefore highly relevant to this chapter. Going forward, we assume that creating and using packages in Java is well understood.

Module Declaration

A *module declaration* embodies the definition of a module that contains packages— that is, it specifies a module's properties. A module declaration must specify the *name* of the module and can declare the following in its declaration:

- *Dependencies* the module has on other modules—that is, modules it *requires* to implement its functionality
- Packages it *exports* for other modules to use
- Packages it *opens* for *reflection* to other modules (p. 1191)
- Services it *provides* (p. 1196)
- Services it *consumes* (p. 1196)

In essence, a modular declaration specifies any modules its packages (and hence their classes) require to implement their functionality, and any packages that are provided for the benefit of other modules. Unless otherwise specified, other packages in the module are internal to the module—that is, they are not accessible to other modules.

The *basic form* of a module declaration is shown below. A module has a unique *name*. The *body* of the modular declaration can include *module directives*. The `requires` and `exports` *directives* are shown in the module declaration below. The modules required by a module for its internal implementation are specified using a `requires` directive, and the packages that other modules can utilize from it are specified using an `exports` directive. The latter is called an *unqualified export* as the exported package is available to any module that requires this module. (See §19.4, p. 1177, for an overview of all module directives.)

Click here to view code image

```
module moduleName {
  // Body:
  requires moduleName1;          // A requires directive.
  ...
  requires moduleNamen;
  exports packageName1;          // An exports directive.
  ...
  exports packageNamem;
}
```

There can be zero or more such directives in such a module declaration. So a module declaration with no directive is valid, as it neither requires any modules (bar the `java.base` module) nor exports any of its packages.

The same directive cannot be duplicated. The directives can occur in any order, but conventionally they are grouped with `requires` directives preceding `exports` directives. In a module declaration, the word `module` is treated as a keyword only in the context of specifying the module name, and the words `requires` and `exports` are treated as keywords only in the context of specifying a module directive. They can be used as normal identifiers in all other contexts.

Only one module declaration can be specified in a file, which must be named `module-info.java`. No other declarations can be included in this file, and there can be no `package` statement either, but the `import` statement is allowed. This file also cannot be empty. As any other Java source code file, the `module-info.java` file with the module declaration is compiled and its bytecode is placed in a class file named `module-info.class`, called the *module descriptor*.

The graphical notation in **Figure 19.3(a)** and **(b)** shows two modules, `client` and `seller`, with their corresponding module declarations in **Figure 19.3(c)** and **(d)**, respectively.

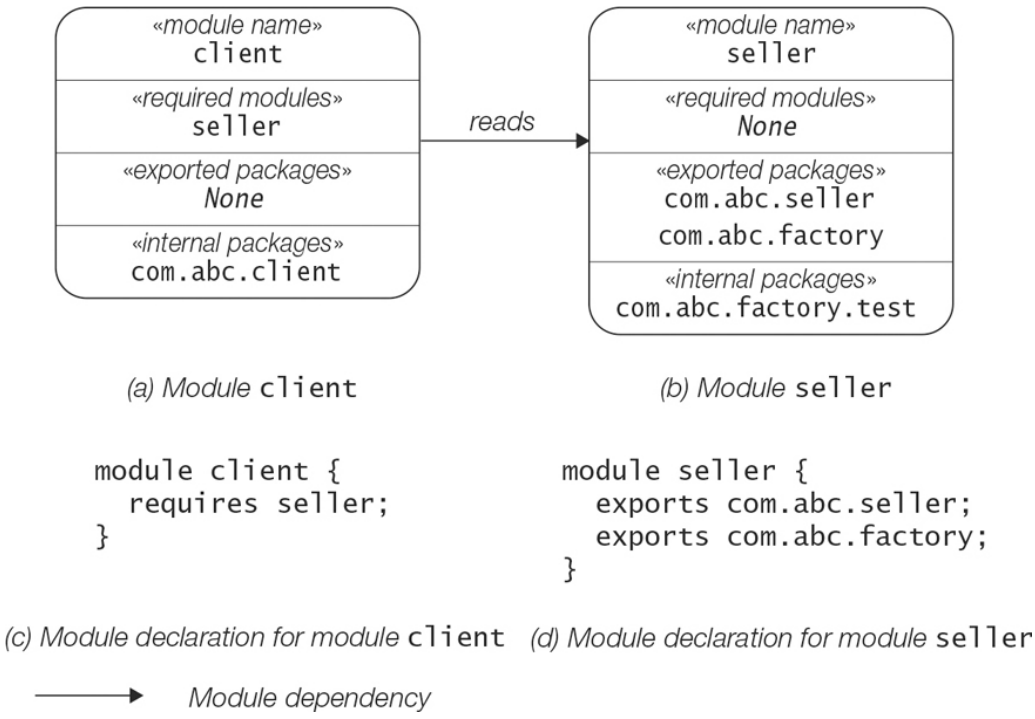


Figure 19.3 *Modules and Module Declarations*

- *The name of the module*: This is the unique *name* of the module to distinguish it from other modules.
  In **Figure 19.3(a)** and **(b)**, the names of the modules are `client` and `seller`, respectively. These module names are used in the respective module declaration in **Figure 19.3(c)** and **(d)**.
- *The required modules*: These are the modules that a module *requires* for the types (i.e., classes and interfaces) in its packages.
  In **Figure 19.3(a)** and **(b)**, the `client` module requires the `seller` module and the `seller` module does not require any module, respectively. This is reflected in the respective module declarations in **Figure 19.3(c)** and **(d)**: The module declaration for `client` has a `requires` directive specifying the `seller` module, but the module declaration for `seller` has no `requires` directive. The `requires` directive states that the specified module is required at compile time and runtime by the declared module.

The `client` module has a dependence on the `seller` module, called a *module dependency*, shown with a directed arrow from `client` module to `seller` module in **Figure 19.3**. Technically, the `client` module is said *read* the `seller` module. Often, we say module `client` *depends on* module `seller`, or module `client` *requires* module `seller`, or module `seller` *is readable* by module `client`. Although technically not correct, for the purpose of this discussion, they are all synonyms.

- *The exported packages*: These are packages that a module *exports* so other modules can utilize them for their own implementation. Exported packages define the *public API* of the declared module, where *accessibility rules* govern what types (and hence which members in these types) in an exported package are accessible to other modules (**p. 1174**).
  In **Figure 19.3(a)** and **(b)**, the `client` module does not export any packages and the `seller` module exports packages `com.abc.seller` and `com.abc.factory`. This is reflected in the respective module declarations: The module declaration for `client` has no `exports` directive, but the module declaration for `seller` has two `exports` directives that specify the exported packages.
  Although the package `com.abc.factory` in module `seller` is exported, it does *not* imply that the package `com.abc.factory.test` is exported as well. This is analogous to the `import` statement, where packages must be imported individually.
- *The internal packages*: These are the packages that are *internal* to the module and therefore not accessible to other modules.
  In **Figure 19.3(a)** and **(b)**, the `client` module has an internal package `com.abc.client` and the `seller` module has an internal package `com.abc.factory.test`. However, there is no directive to specify an internal package in the module declaration. The question of how we implement a module and its packages in the file system is answered in **§19.5**, **p. 1179**.

Module Name

A legal module name can consist of one or more (by convention, lowercase) legal identifiers separated by a `.` (dot), as in the case of a package name. Module names and package names are in different *namespaces*. From the context in which such a name occurs, the compiler can deduce which namespace it belongs to. For example, the name `com.abc.seller` is a module name if used with the context-dependent keyword `module` in a module declaration, but it is a package name when used in an `exports` directive.

Analogous to package names, reverse DNS (*Domain Name System*) notation can be used to form unique module names. The convention is to use the name of the *principal exported package* (if there is one) in a module as its module name. When naming a module, it is recommended that the name should reflect the *structure* of the module—that is, its package hierarchy. For example, the module declarations in **Figure 19.3(c)** and **(d)** can be written as follows, respectively:

Click here to view code image

```
module com.abc.client {
  requires com.abc.seller;
}
```

```
module com.abc.seller {
  exports com.abc.seller;
  exports com.abc.factory;
}
```

If care is not exercised in naming modules and packages, distinguishing between module names and package names can become a problem when reading the code, as we can see in the module declarations above, where the name `com.abc.seller` is used both as a module name and as a package name. For brevity, we will not adhere to the naming scheme described above and will continue to use simple names for modules, as in **Figure 19.3**.

A package is implemented in the file system by a directory hierarchy that mirrors its name. For example, a package named `com.abc.seller` is implemented by its source code files being located under the directory `./com/abc/seller`. As we shall see in §19.5, p. 1181, a module named `com.abc.seller` is implemented by a directory whose name is `com.abc.seller` under which all its packages are located.

Some examples of legal module names (as well as legal package names):

Click here to view code image

```
my.big.fat.module, com.passion.controller, org.geek.gui, view
```

Some examples of illegal module names (as well as illegal package names):

Click here to view code image

```
net.soda.7up, org:factory, com-factory-gizmo
```

Module Graph

The structure of an application is reflected by its module graph constructed from dependencies between its modules. An important aspect of the module system is to perform *module resolution* at compile time and runtime to ensure that the application has a *reliable configuration*. This process *resolves* all dependencies, making sure that they are satisfied—that is, the `requires` directives are matched by `exports` directives with all types required being accessible.

The module resolution mandates that the module graph is *acyclic*, meaning that the graph cannot have *dependency cycles* where a module depends on itself, either directly or indirectly. The module system also does not allow a package to be split between different modules, primarily because this would mean searching in several modules to locate a type in a split package. Therefore, all code in a package must be in the same module.

Every module depends on the proverbial `java.base` module in the JDK. Its packages are available to all modules—for example, the package `java.lang` containing classes like `Object` and `String`. Specifying `java.base` in a `requires` directive is optional in a module declaration. If not specified explicitly, an *implicit dependency* on `java.base` is automatically applied.

Figure 19.4(a) shows the module graph from Figure 19.3 augmented by the two dependencies on `java.base`. The declarations of module `client` and module `seller` in Figure 19.4(b) and (c) include explicit `requires` directives specifying `java.base`. The convention is to leave the dependency on `java.base` implicit. Showing all implicit dependencies from all modules on `java.base` in a module graph can quickly make the graph look like spaghetti. Often this dependency is only shown for modules that do not depend on other modules (a.k.a. *sink nodes* in graph terminology).

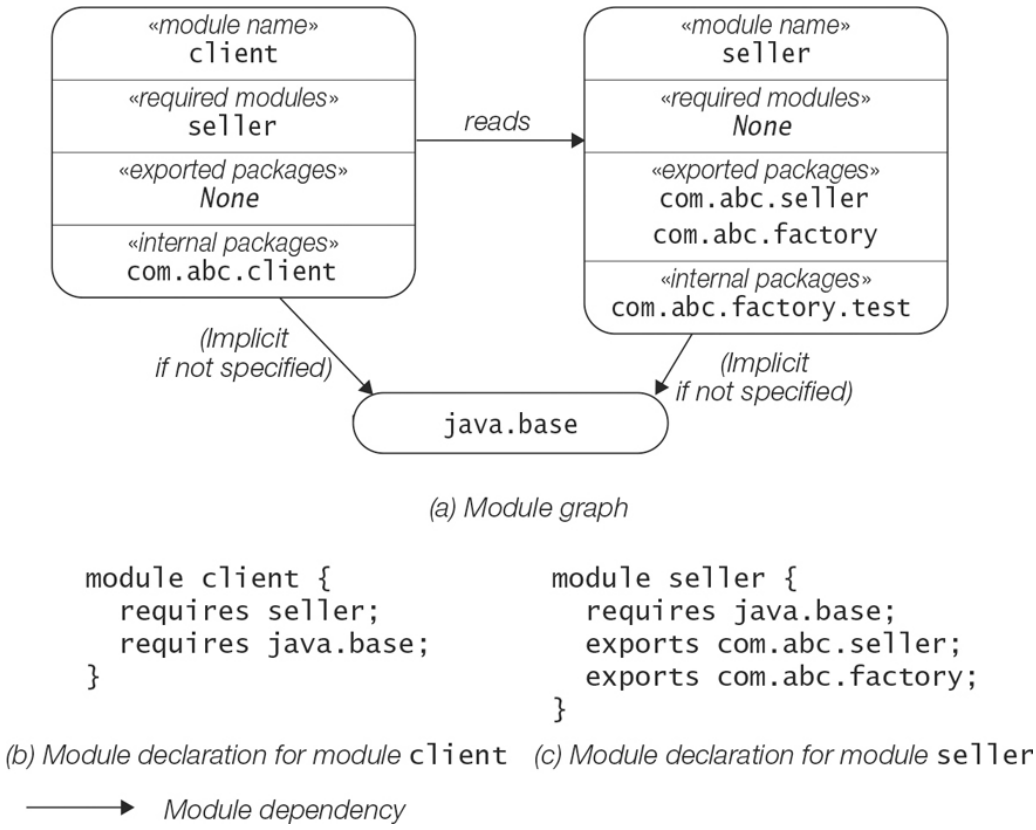


Figure 19.4 *Module Graph and Implicit Dependencies*

Readability and Accessibility

A *readability* relationship is implied by the `requires` directive. Module `client` *reads* module `seller`, since the module declaration of `client` specifies the following `requires` directive:

Click here to view code image

```
requires seller;                            // (1)
```

This `requires` directive alone is not enough to allow code in module `client` to access types in packages contained in module `seller`. Module `seller` must reciprocally *export* the relevant packages required by module `client`, as in the following `exports` directive in the declaration of module `seller`:

Click here to view code image

```
exports com.abc.seller;                     // (2)
```

What if the code in module `client` wants to access the class `GizmoSeller` in the package `com.abc.seller` of module `seller`? In this case, this class is only accessible to module `client` if it is declared `public` in the package `com.abc.seller`:

Click here to view code image

```
public class GizmoSeller { /*...*/ }        // (3)
```

The three conditions, as exemplified by (1), (2), and (3) above, must be satisfied for the accessibility relationship. The module system ensures, both at compile time and runtime, that these conditions are met before allowing access to code from other modules.

We see that only a `public` type in an exported package of a module is accessible by modules that read this module. The exported `public` types comprise the *public API* of a module.

Accessibility Rules for Members

Table 19.1 summarizes the accessibility rules for *members* of a `public` class `A` contained in a package `pkg1` of module `M`. Note that class `A` is `public` so that it satisfies one of the three conditions discussed earlier for accessibility of a type from another module.

Table 19.1 *Module Accessibility Rules*

Access modifier for members in `public` class `A` inside package `pkg1` of module `M`	**Inside module `M`**	**Inside module `N`**
`public`	Accessible in all packages in module `M`	Accessible in all packages in module `N`, if and only if module `N` *requires* module `M` and module `M` *exports* `pkg1`
`protected`	Accessible in package `pkg1`, and accessible only by subclasses of class `A` in other packages in module `M`	Accessible by subclasses of class `A` in all packages in module `N`, if and only if module `N` *requires* module `M` and module `M` *exports* `pkg1`
package-private	Only accessible in package `pkg1`	Not accessible
`private`	Only accessible in class `A`	Not accessible

In Table 19.1, the middle column shows how the members of `public` class `A` are accessible inside module `M`. Accessibility rules for members that are applied before modules are now only applicable for accessing members inside a module.

In module `N`, the `public` and the `protected` members of `public` class `A` are accessible if and only if class `A` is accessible in module `N`. In order for class `A` to be accessible in module `N`, the conditions for accessibility of types must be satisfied: Module `N` *requires* module `M`, module `M` *exports* package `pkg1`, and class `A` in package `pkg1` is `public`. Not surprisingly, `private` and package-private members of `public` class `A` are not accessible in module `N`.

Implied Module Dependencies

As an example of refactoring modules, we will refactor the `seller` module in Figure 19.3(b). We decide to move the package `com.abc.factory` and its subpackage `com.abc.factory.test` to a new module called `factory`. The effects of this factoring are illustrated in Figure 19.5. Now both `client` and `seller` require `factory`, as can be seen from their module declarations in Figure 19.5(d) and (e). The package `com.abc.factory` is now exported by the module `factory` in Figure 19.5(f). Apart from fixing the module declarations and moving the relevant packages from `seller` to `factory`, no other changes are necessary.

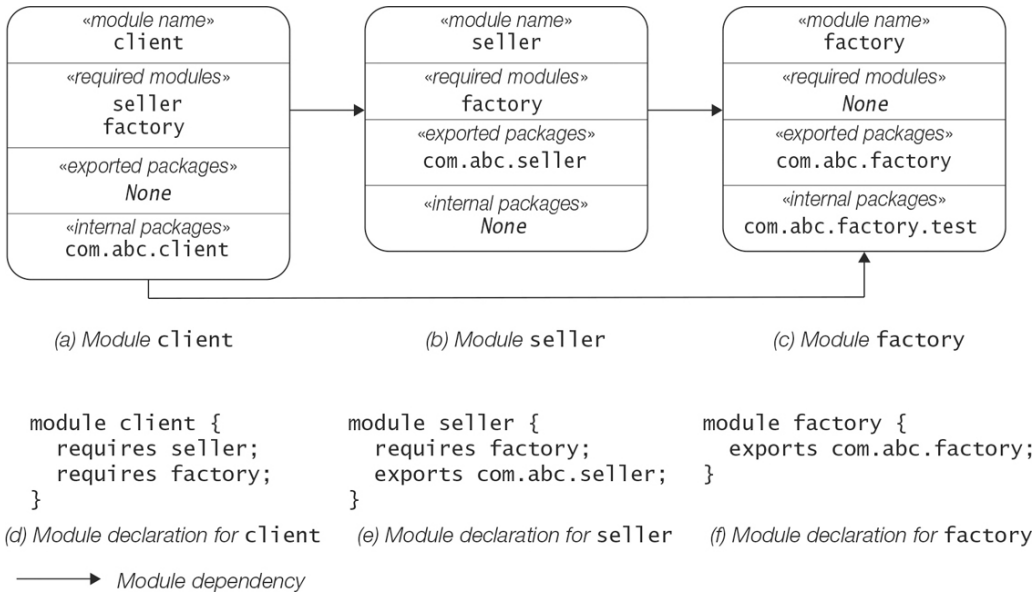


Figure 19.5 *Module Declarations and Module Graph*

The module graph in **Figure 19.5** shows the dependencies between the modules after refactoring. Note that there are two paths from `client` to `factory` in the module graph. One might be tempted to think that since `client` requires `seller` and `seller` requires `factory`, then `client` requires `factory` transitively, and we could omit the `requires` directive on `factory` in `client`. However, the compiler will diligently flag an error. Dependencies specified by the `requires` directive are *not* transitive.

The `requires transitive` *Directive*

Implied dependencies can be created by the `requires transitive` directive. This is illustrated in **Figure 19.6**. If it is the case that all modules that require `seller` will also require `factory`, we can omit the `requires` directive on `factory` in `client` and use a `requires transitive` directive in `seller`. In **Figure 19.6**, `client` no longer requires `factory`, but `seller` specifies `factory` in a `requires transitive` directive. All changes are confined to the module declarations in **Figure 19.6(d)** and **(e)**, so no other changes are required in the code. Implications of using a `requires transitive` directive are that any module that requires `seller` now will also implicitly require `factory`. This implied dependency is shown from `client` to `factory` in the module graph in **Figure 19.6**.

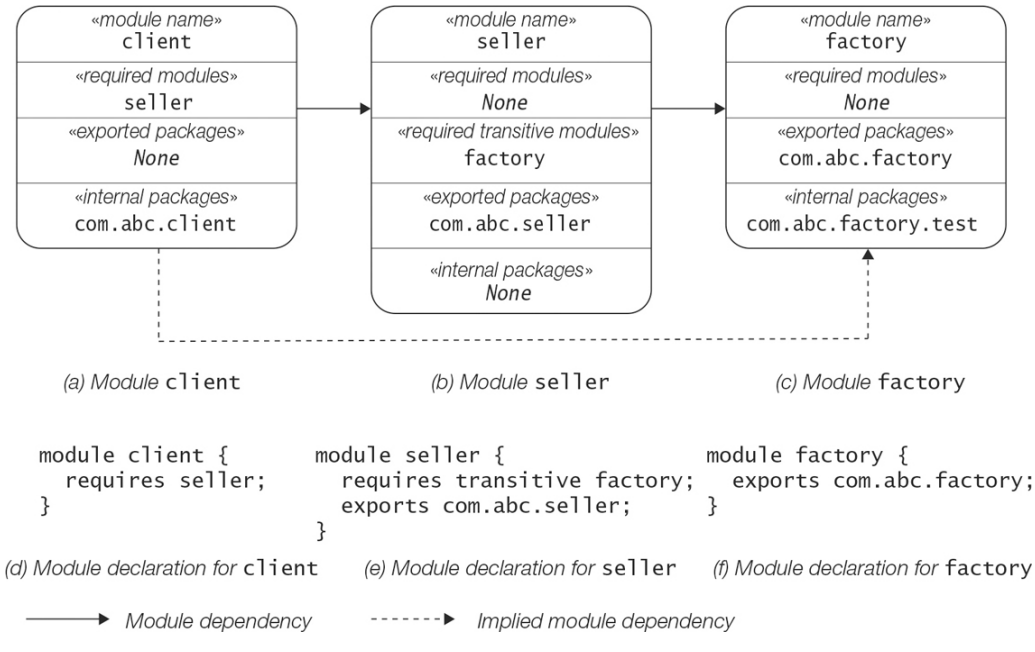


Figure 19.6 *Implied Module Dependencies*

Qualified Export

Any package that is exported by a module is readable by any module that requires its module. However, sometimes it is necessary that only certain modules can access an exported package. This can be achieved by using the `exports-to` directive, as shown below at (1), where `packageOne` in `moduleX` is *only* exported to `moduleA` and `moduleB`:

Click here to view code image

```
module moduleX {
  exports packageOne to moduleA, moduleB;   // (1) Qualified export.
  exports packageTwo;                       // (2) Unqualified export.
}
```

This is known as *qualified export*. Apart from `moduleA` and `moduleB`, no other module can access `packageOne` in `moduleX`. The compiler will issue an error if an attempt is made by any other module. Qualified export does not prevent the modules in the `to` clause to access other packages that are exported. Unqualified export of `packageTwo`, shown at (2), allows *any* module that requires `moduleX` to access `packageTwo`. Qualified export should be used judiciously, as its usage results in tight coupling between modules that can increase the degree of dependency between modules—a hallmark of poor software design.

Figure 19.7 shows how the module declaration of the `factory` module that uses the `exports-to` directive allows its package `com.abc.factory` to be accessible only by the `seller` and the `client` module. Any other module that requires module `factory` will not be able to access the `com.abc.factory` package in the `factory` module.

```
module client {
  requires seller;
  requires factory;
}
```

(d') Module declaration for `client`

```
module seller {
  requires factory;
  exports com.abc.seller;
}
```

(e') Module declaration for `seller`

```
module factory {
  exports com.abc.factory
      to seller, client;
}
```

(f') Module declaration for `factory` *with qualified export of its package*

Figure 19.7 *Qualified Export*

19.4 Overview of Module Directives

An overview of all module directives is given in Table 19.2, together with references to where they are covered in this chapter.

Table 19.2 *Overview of Module Directives*

Directive	Description
`requires` *`module`* (p. 1170)	The current module specifies a dependence on the specified *`module`*, allowing the current module to access exported public types in the specified *`module`*.
`requires transitive` *`module`* (p. 1175)	Any module that requires the current module will have an implicit dependence on the specified *`module`*, and can access exported public types in the specified *`module`* on par with the current module.
`requires static` *`module`*	The current module has a mandatory dependence on the specified *`module`* at compile time, but is optional at runtime. This is known as *optional dependency*. It is not discussed in this book.
`exports` *`package`*	The specified *`package`* exported by the current module is available to other modules that require the current module—that is, they can access

Directive	Description
(p. 1170)	public types in the specified `package`. This is called an *unqualified export*.
`exports package to module1, ...` *(p. 1176)*	The specified `package` exported by the current module is solely available to the modules specified in the `to` clause. That is, only those modules in the `to` clause that require the current module can access public types in the specified `package`. This is called a *qualified export*.
`opens package` *(p. 1191)*	Access to the specified `package` is only granted through *reflection* (§25.6, p. 1587) at runtime for code in other modules. This is called an *unqualified opens directive*.
`opens package to module1, ...` *(p. 1195)*	Access to the specified `package` is only granted through *reflection* at runtime solely for code in modules specified in the `to` clause. This is called a *qualified opens directive*.
`provides type with type1, ...` *(p. 1196)*	The current module specifies that it provides a *service* (specified by `type`) that is implemented by one or more *service providers* (`type1, ...` in the `with` clause implement the service). The current module is said to be a *service provider*. The service provided can be consumed by other modules that want to use this service.
`uses type` *(p. 1196)*	The current module specifies that it uses a *service* (specified by `type`) and that a *service provider* that implements it may be discoverable at runtime. Typically, the current module is said to be a *service locator*, but can also be a *service consumer* if the locator is merged with the consumer.

Modules add 10 restricted keywords to the language: `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses`, and `with`. The words `open` (see Table 19.3) and `module` are keywords only in the header of a module declaration, and the remaining in a particular context of a specific directive. The word `static` is already a keyword, but has a different meaning in the `requires static` directive.

Table 19.3 *The* `open` *Modifier for Modules*

Modifier	Description
`open module module { ... }` *(p. 1196)*	Unrestricted access to *all* packages in the *module* is granted through *reflection* at runtime for code in other modules.

19.5 Creating a Modular Application

After understanding the basic role of a module declaration, we can proceed to create a modular application. As support for Java modules is not quite up to par at present in integrated development environments (IDEs), we will use the command-line tools provided by the JDK. Using the command-line tools also provides a deeper understanding of the Java Module System. We will be using the following JDK tools in the rest of this chapter:

- *Java language compiler: the* `javac` *command (p. 1186)*
- *Java application launcher: the* `java` *command (p. 1186)*
- *Java Archive tool: the* `jar` *command (p. 1189)*
- *Java Class Dependency Analyzer tool: the* `jdeps` *command (p. 1214)*
- *Java Linker tool: the* `jlink` *command (p. 1204)*

The JDK tools are versatile tools with numerous features and many capabilities. The myriad of options for each tool can be overwhelming. Tables with summaries of selected options for each tool are provided, starting with **Table 19.6** on **page 1218**.

As a running example, we will develop an elementary multi-module application that is based on a very simplified Model-View-Controller (MVC) design pattern. Prior knowledge of the MVC design pattern is not essential. To keep things manageable, the emphasis is more on the mechanics of creating a modular application than on providing an industrial-strength implementation of the MVC design pattern.

To make it less tedious in experimenting with the modular application code, a file with the relevant commands is provided, together with the source code of the application, all of which can readily be downloaded from the book's website. Hands-on coding of a modular application is highly encouraged, to understand both the Java modules and the tools essential for creating modular applications.

Running Example: Dispensing Canned Advice

In the MVC design pattern, the controller handles the user requests. The model provides the business logic and manages the data in the application. The view creates the response to the user, usually by updating the data display. Simplified, the interaction in an MVC-based application proceeds as follows:

1. A user sends a request to the controller.
2. The controller interacts with the model and with the view.
   First it requests the model to update its data according to the user request, and then it requests the view to create a response. The view accesses the model and updates the data display accordingly.

The dispensing canned advice application (called `adviceApp`) embodies the simplified MVC design pattern outlined above. It comprises four modules: the `model`, `view`, `controller`, and `main` modules, shown in **Figure 19.8**. The `main` module in **Figure 19.8** simulates the user mentioned above. From the notation used for a module in **Figure 19.8**, we can read the name of the module, the modules it requires, the packages it exports, and any packages that are internal to the module. Based on the modules required by each module, the resulting *module graph* is also shown in **Figure 19.8**, with *dependencies* between them. For example, the `controller` module requires the modules `view` and `model`, thus there are two dependencies from the `controller` module to the `view` and the `model` modules, respectively.

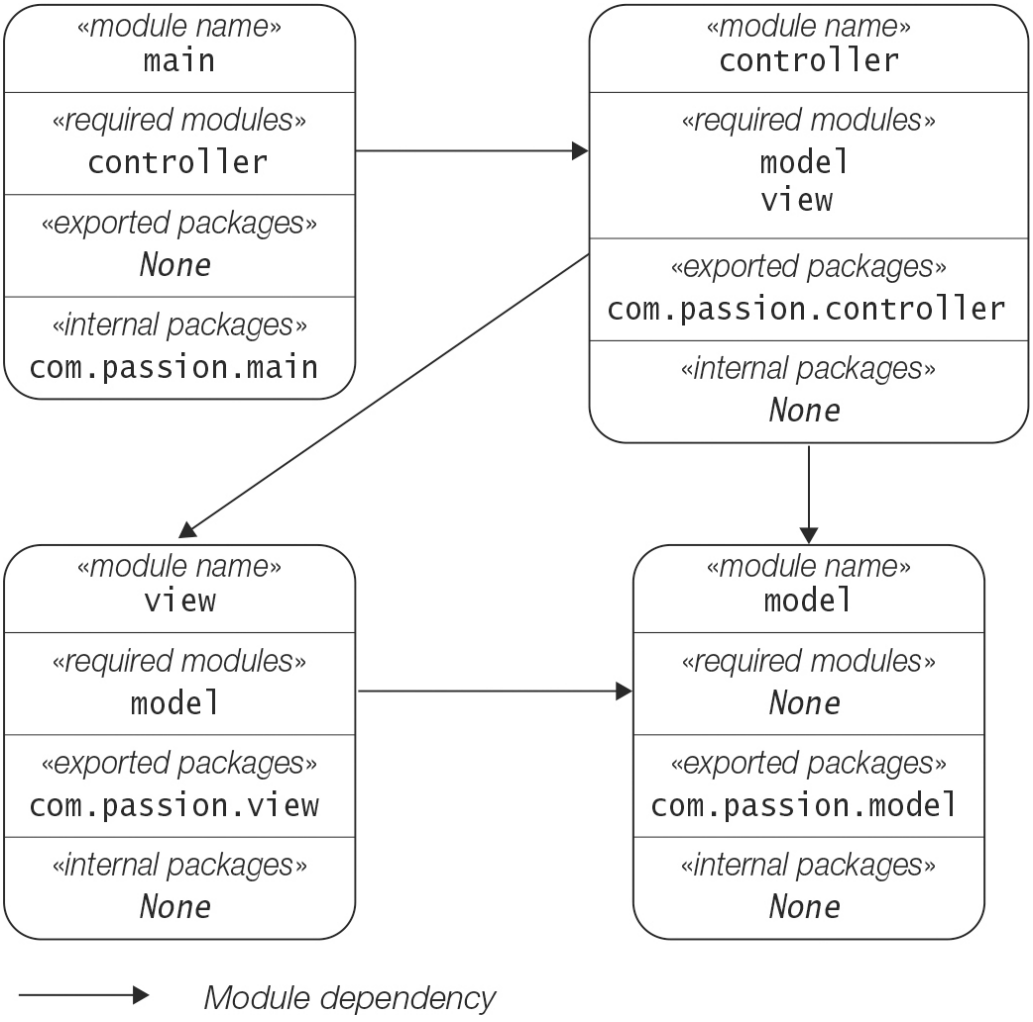


Figure 19.8 *Module Graph of the* `adviceApp` *Application*

Creating the Application Directory Structure

Typically, the code for a modular application is organized in a *directory structure*, similar to the one shown in **Figure 19.9**. The modules are created in the `src` directory and compiled into the `mods` directory. The compiled modules are bundled into modular JARs and placed in the `mlib` directory. The different JDK tools can thus operate on the relevant directories.

Figure 19.9 *Application Directory Structure*

The application root directory for the `adviceApp` application bears the same name. It is also the *working directory* in which all commands are issued to the JDK tools. Under the `adviceApp` directory, the following command creates the directories shown in **Figure 19.9**:

```
>mkdir mlib mods src
```

Creating an Exploded Module Directory

The `model` module does not require any user-defined modules—that is, it does not depend on any user-defined module, and can therefore be implemented first. Its module declaration is straightforward, as it exports its only package, `com.passion.model` (shown in **Figure 19.10**). The directory layout for the `model` module is created manually under the `src` directory as shown in **Figure 19.10**, and referred to as the *exploded module*.

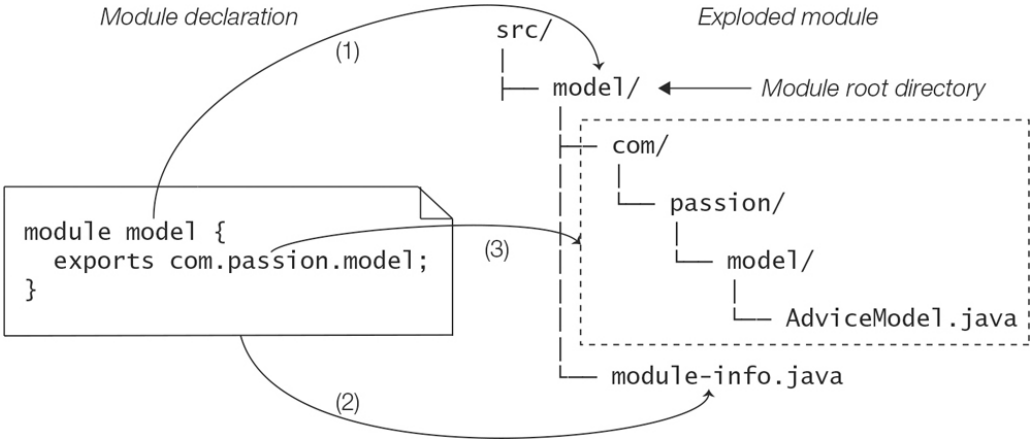


Figure 19.10 *Exploded Module Structure for the* `model` *Module*

Referring to **Figure 19.10**, (1) shows that the module name and that of the *module root directory* must be the same, and (2) shows that the `module-info.java` file containing the module declaration must reside immediately under the module root directory.

Packages were discussed in **§6.3**, **p. 326**. Any package (or subpackage) that a module contains is mapped to a corresponding directory hierarchy under the module root directory. In **Figure 19.10**, (3) shows how the package `com.passion.model` of the module is laid out under the module root directory, with the directory hierarchy `com/passion/model` mirroring its package name. It also shows that the package has only one source file, `AdviceModel.java`. If the package had any other source files, they would also be placed in the same location.

A package is included in a module by virtue of its location in the module root directory. The organization of the module root directory, as in **Figure 19.10**, defines what constitutes a module. Similarly, any other packages in the module are mapped to their directory structure under the module root directory.

Figure 19.11 shows the module graph of the `adviceApp` application together with the module declaration of each module that is in the module graph. Each module declaration is in a source file named `module-info.java`, and is placed in its corresponding exploded module directory, as shown in **Figure 19.12**.

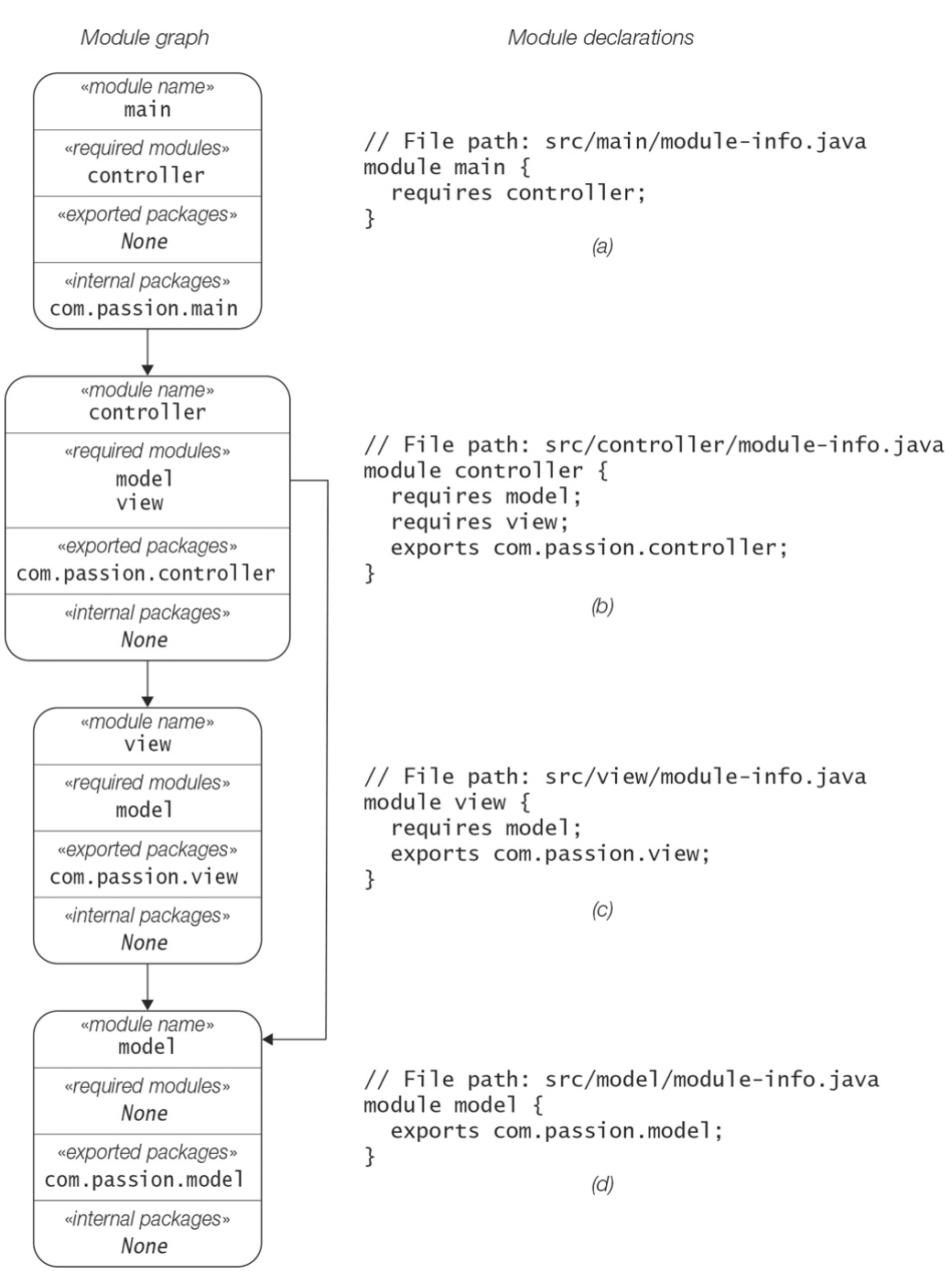


Figure 19.11 *Module Graph and Module Declarations*

```
src/
├── controller/
│   ├── com/
│   │   └── passion/
│   │       └── controller/
│   │           └── AdviceController.java
│   └── module-info.java
├── main/
│   ├── com/
│   │   └── passion/
│   │       └── main/
│   │           └── Main.java
│   └── module-info.java
├── model/
│   ├── com/
│   │   └── passion/
│   │       └── model/
│   │           └── AdviceModel.java
│   └── module-info.java
└── view/
    ├── com/
    │   └── passion/
    │       └── view/
    │           └── AdviceView.java
    └── module-info.java
```

(a) Exploded modules containing source code

```
(3)
>javac --module-path mods \
      -d mods/controller \
      src/controller/module-info.java \
      src/controller/com/passion/ \
        controller/AdviceController.java
(4)
>javac --module-path mods \
      -d mods/main \
      src/main/module-info.java \
      src/main/com/passion/main/ \
                        Main.java
(1)
>javac --module-path mods \
      -d mods/model \
      src/model/module-info.java \
      src/model/com/passion/model/ \
                    AdviceModel.java
(2)
>javac --module-path mods \
      -d mods/view \
      src/view/module-info.java \
      src/view/com/passion/view/ \
                  AdviceView.java
```

(b) Order of compiling exploded modules individually

Figure 19.12 *Compiling Exploded Modules Containing Source Code*

Figure 19.12(a) shows the contents of the `src` directory where each module is mapped to its corresponding exploded module. Each module has one package that contains one source file. This package is mapped to its directory hierarchy under the root directory of its module.

Creating the Source Files in the Exploded Modules

The classes that are in packages included in the modules are declared in their respective source files. **Example 19.1** shows the source files for the `adviceApp` application.

The source code of the `AdviceModel.java` file in the `model` module is shown at (1) in **Example 19.1**. This source file should reside in the package `com.passion.model` in the `model` module. Its file path `src/model/com/passion/model/AdviceModel.java` in the exploded module is shown as a comment at (1). The class `AdviceModel` is declared to be in package `com.passion.model`. It keeps track of the current advice (the `String` field `currentAdvice`). Its constructor creates an `AdviceModel` with the field `current-Advice` initialized to the string `"No advice."` The method `setCurrentAdvice()` sets the field `currentAdvice` to the advice corresponding to its `int` parameter value. The method `getCurrentAdvice()` returns the value of the field `currentAdvice`.

The source code of the `AdviceView.java` file in the `view` module is shown at (2) in **Example 19.1**. This source file should reside in the package `com.passion.view` in the `view` module. Its file path `src/view/com/passion/view/AdviceView.java` in the exploded module is shown as a comment at (2). The class `AdviceView` is declared to be in package `com.passion.view`. It uses an `AdviceModel`. This model is passed to an `AdviceView` in the constructor. The method `updateAdviceDisplay()` accesses its model for the current advice and prints it.

Analogously, the source code of the `AdviceController.java` file in the `controller` module is shown at (3) in **Example 19.1**. An `AdviceController` uses an `AdviceModel` and an `AdviceView`, both of which are created in the constructor, where the `Advice-Model` is injected into the `AdviceView`. The method `showAdvice()` calls the `AdviceModel` to set the current advice based on its `int` parameter value, followed by a call to the `AdviceView` to update the advice display.

Finally, the source code of the `Main.java` file in the `main` module is shown at (4) in **Example 19.1**. The `Main` class is the entry point of the application. It simulates a user by calling an `AdviceController` several times to show different advice.

Note that all source code should be explicitly contained in packages, except for the module declaration—that is, the `module-info.java` file, which is always located immediately under the module root directory. The name of the module in the module declaration must match the name of the module root directory.

Example 19.1 *Source Code Files for the* `adviceApp` *Application*

Click here to view code image

```
// File path: src/model/com/passion/model/AdviceModel.java                (1)
package com.passion.model;

public class AdviceModel {

  private String currentAdvice;

  public AdviceModel() { this.setCurrentAdvice(0); }
```

```
  public void setCurrentAdvice(int i) {
    String advice;
    switch(i) {
      case 1 : advice = "See no evil."; break;
      case 2 : advice = "Speak no evil."; break;
      case 3 : advice = "Hear no evil."; break;
      default: advice = "No advice.";
    }
    currentAdvice = advice;
  }

  public String getCurrentAdvice() { return currentAdvice; }
}
```

Click here to view code image

```
// File path: src/view/com/passion/view/AdviceView.java                    (2)
package com.passion.view;
import com.passion.model.AdviceModel;             // From model module.

public class AdviceView {

  private AdviceModel model;

  public AdviceView(AdviceModel model) { this.model = model; }

  public void updateAdviceDisplay(){
    System.out.println(model.getCurrentAdvice());
  }
}
```

Click here to view code image

```
// File path: src/controller/com/passion/controller/AdviceController.java   (3)
package com.passion.controller;
import com.passion.model.AdviceModel;               // From model module.
import com.passion.view.AdviceView;                 // From view module.

public class AdviceController {

  private AdviceModel model;
  private AdviceView view;

  public AdviceController() {
    model = new AdviceModel();
    view = new AdviceView(model);                   // Inject the model.
  }

  public void showAdvice(int adviceNumber) {
    model.setCurrentAdvice(adviceNumber);
    view.updateAdviceDisplay();
  }
}
```

Click here to view code image

```
// File path: src/main/com/passion/main/Main.java                          (4)
package com.passion.main;
import com.passion.controller.AdviceController;    // From controller module.
```

```
public class Main {
  public static void main(String... args) {
    AdviceController controller = new AdviceController();
    controller.showAdvice(1);
    controller.showAdvice(2);
    controller.showAdvice(3);
    controller.showAdvice(0);
  }
}
```

19.6 Compiling and Running a Modular Application

Both the `javac` tool to compile Java source code and the `java` tool to launch an application include new command-line options to specifically support building of modular applications.

When the `javac` tool or the `java` tool is called, initially a *module resolution* is performed on the application's structure. This process checks that dependencies among the modules are resolved in the application, thus ensuring a reliable configuration, by catching any problems as early as possible.

The astute reader will soon notice that the tool commands in this chapter are run on a Unix-based platform. Platform-dependent idiosyncrasies are pointed out where appropriate. The first one to note is the line-continuation character to break up a command line over several lines. This is to make the command and its options easier to read. The *line-continuation character* is a `\` (backslash) on the Unix-based platform and a `^` (caret) on the Windows platform. Below, the command on a single line and the command that spans over two lines are equivalent.

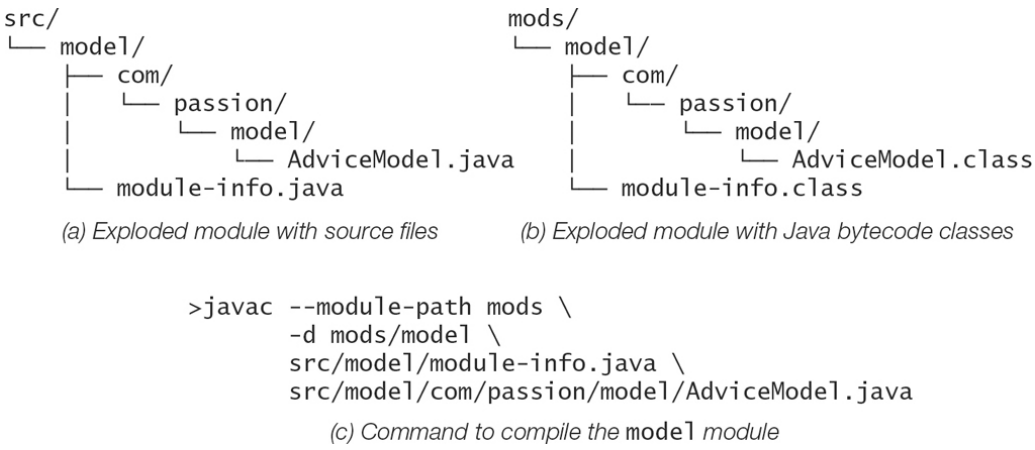
```
>javac One.java Two.java

>javac One.java\
 Two.java
```

Individual Module Compilation

We first look at how to compile the modules of the `adviceApp` application individually. Typically, in order to compile a module, the compiler needs to know the source directory to find the source code of the module, the destination directory to place the Java bytecode class files for the module, and any additional modules that the module requires.

The modules in an application can only be compiled in an order that respects the module dependencies—that is, all modules a module requires have already been compiled—or if it does not depend on any other modules. In the case of the `advice-App` application, there is only one such order, shown in **Figure 19.12(b)**. We can compile the `model` module first since it does not require any other modules (apart from the `java.base` module that is readily accessible by default). **Figure 19.13** shows how the `model` module is compiled. The `javac` command in **Figure 19.13(c)** shows how the source code of the `model` module under the `src/model` directory is compiled to the `mods/model` directory where the necessary package is created and the class files are placed. Note how the `mods/model` directory mirrors the `src/model` directory.



(a) Exploded module with source files *(b) Exploded module with Java bytecode classes*

(c) Command to compile the `model` *module*

Figure 19.13 *Compiling the* `model` *Module*

Click here to view code image

```
>javac --module-path mods \
       -d mods/model \
       src/model/module-info.java \
       src/model/com/passion/model/AdviceModel.java
```

The `--module-path` option (*short form*: `-p`) is used to specify where to find the modules required to compile the source code files specified in the command line. In the case of the `model` module, this option is superfluous as this module does not depend on any other module. However, the other modules in the `adviceApp` application do so. As modules are compiled, their exploded modules containing the class files will be readily found in the `mods` directory specified by this option. This option can also specify a colon-separated (`:`) list of directories (semicolon (`;`) separated in Windows) to look up required modules in different directories.

The `-d` option (*no long form*) specifies the *destination directory* (also called *target directory*) for placing the Java bytecode class files. In this case they will be placed under the `mods/model` directory, which is created by the compiler if necessary.

The source files in the module are specified *last* in the command line with their full pathname. The compiler creates the necessary directory hierarchy under the destination directory `mods/model` specified by the `-d` option. The module declaration in the `module-info.java` file is also compiled to a class file, as any other Java source file. This class file is called the *module descriptor*. It resides immediately under the module root directory, which in this case is `mods/model`.

After the `model` module is compiled, the next candidate to compile is the `view` model, as its required module `model` has now been compiled to the `mods/model` directory. **Figure 19.14(a)** shows the final result of compiling the modules in the `adviceApp` application by executing the `javac` commands in the order shown in **Figure 19.12(b)**.

Figure 19.14 *Creating Modular JARs*

Multi-Module Compilation

It is tedious compiling modules individually. It is more convenient to use the following command:

Click here to view code image

```
>javac --module-source-path src -d mods --module main
```

The `--module-source-path` option (*no short form*), as the name suggests, indicates the `src` directory where the exploded modules with the source files for the modules can be found.

The `-d` option (*no short form*) indicates the directory `mods` for the exploded modules with the class files. The compiler creates the module root directories and the package hierarchies as necessary under the `mods` directory.

The `--module` option (*short form*: `-m`) can be used to specify the name of a single module or a list of module names separated by a comma (`,`) with no intervening space characters. For each module specified, the compiler will only compile those modules that this module depends on. The compiler does the necessary module dependency analysis and compiles all the modules in the right order. If only the `model` module had been specified, only it will be compiled, as it does not depend on any other module.

The result of the above command is the same as before, shown in **Figure 19.14(a)**.

Running a Modular Application from Exploded Modules

Typically, the `java` command needs to know where to find the modules needed to run the application and also the entry point of the application, meaning the class whose `main()` method should be executed to launch the application.

Click here to view code image

```
>java --module-path mods --module main/com.passion.main.Main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

The `--module-path` option (*short form*: `-p`) is used to specify where to find the modules needed to run the application. In this case it is the `mods` directory containing the exploded modules with the class files. This option can also specify a colon-separated (`:`) list of directories (semicolon (`;`) separated in Windows) to look up the required modules.

The `--module` option (*short form*: `-m`) in the command explicitly specifies the *entry point* of the application. In this case, it is the class `Main` in the `com.passion.main` package of the `main` module that provides the `main()` method. This requires the *fully qualified name* of the class (`com.passion.main.Main`) and the *full name* of the module (`main`). Note the *directory-level separator* (`/`) in the specification of the entry point.

19.7 Creating JAR Files

A *JAR* (Java Archive) file is a convenient way of bundling and deploying Java executable code and any other resources that are required (e.g., image or audio files). A JAR file is created by using the `jar` tool. The `jar` command has many options, akin to the Unix `tar` command (p. 1221).

In addition to creating *plain JARs* (also referred to as *non-modular JARs*), the `jar` tool of the JDK has been enhanced to create and manipulate *modular JARs*, making it convenient to bundle and deploy modules. The procedure for creating a plain or a modular JAR is the same; the main difference being that a modular JAR always contains the module descriptor—that is, the `module-info.class` file.

Creating Modular JARs

Figure 19.14(b) shows the `jar` commands that can be used to create a modular JAR for each of the modules in the `adviceApp` application. The resulting modular JARs are placed under the `mlib` directory, as shown in **Figure 19.14(c)**. We take a closer look at the `jar` command to create a modular JAR for the `model` module:

```
>jar --verbose \
     --create \
     --file mlib/model.jar \
     -C mods/model .
```

Possible output to the terminal:

Click here to view code image

```
added manifest
added module-info: module-info.class
adding: com/(in = 0) (mods= 0)(stored 0%)
adding: com/passion/(in = 0) (mods= 0)(stored 0%)
adding: com/passion/model/(in = 0) (mods= 0)(stored 0%)
adding: com/passion/model/AdviceModel.class(in = 641) (mods= 415)(deflated 35%)
```

The `--verbose` option (*short form*: `-v`) instructs the `jar` tool to print information about its operation on the terminal, as evident from the output on the terminal.

The `--create` option (*short form*: `-c`) results in a modular JAR to be created, as the module descriptor will be included in the JAR file (see below).

The `--file` option (*short form*: `-f`) specifies the file name of the modular JAR. In this case, it is `model.jar` that will be created and located under the `mlib` directory. Note there are no restrictions on the file name of the modular JAR, as long as it is a legal file name. Of course, having a descriptive file name that reflects what the module in the modular JAR stands for can aid readability.

The `-C` option instructs the `jar` tool to change to the `mods/model` directory and include the contents of this directory which is designated by the dot (`.`) special notation. From the output on the terminal we can see what is added to the JAR file. Note that the modular JAR does not include information about the module root directory (in the above case, `mods/model`) in the modular JAR because the module descriptor already has information about the name of the module. The contents included in the modular JARs are shown by the dashed boxes in **Figure 19.14(a)**.

The observant reader will notice that the `jar` command in **Figure 19.14(b)** to create the `main.jar` has an extra option to specify the application entry point:

Click here to view code image

```
>jar --verbose \
     --create \
     --file mlib/main.jar \
     --main-class com.passion.main.Main \
     -C mods/main .
```

Since the entry point of the application is in the `model` module, the `--main-class` option (*short form*: `-e`) specifies the fully qualified name of the class containing the `main()` method —that is, `com.passion.main.Main`. The module name is *not* specified.

Using the short form of the first four options in the above `jar` command, respectively (see **Table 19.6**), the command can be written as follows:

Click here to view code image

```
>jar -vcfe mlib/main.jar com.passion.main.Main -C mods/main .
```

Note that if the files named `module-info.class` are deleted in **Figure 19.14(a)**, the `jar` commands in **Figure 19.14(b)** will still create JAR files, but these will be plain JARs, and not modular JARs.

Running an Application from a Modular JAR

The `adviceApp` application can be run from the modular JARs in the `mlib` directory by any of the following `java` commands, giving the same output as before:

Click here to view code image

```
>java --module-path mlib --module main
```

or

Click here to view code image

```
>java --module-path mlib --module main/com.passion.main.Main
```

The `--module-path` option (*short form*: `-p`) specifies the `mlib` directory in which to find the modular JARs to run the application. Previously the exploded module directory `mods` containing the class files for each module was specified.

As seen previously, the `--module` option (*short form*: `-m`) specifies the entry point of the application in the `java` command. In the first `java` command, only the module name `main` is specified. The `main` module can be found in the `main.jar` file under the `mlib` directory by examining the module descriptor in each module on the module path. The `main.jar` file contains the information about the class with the `main()` method, as it was specified when the modular JAR for the `main` module was created—that is, `com.passion.main.Main`. In the second `java` command, the entry point is explicitly given, with the module name and the fully qualified class name. Specifying the entry point explicitly overrides the entry point in any modular JAR. However, it makes no difference which command we use in the example above, as there is only one entry point for the application.

We have seen how to create, compile, bundle, and run a modular application. Now we take up the discussion about the remaining directives that can be declared in a module declaration: opening code for reflection (**p. 1191**) and providing services (**p. 1196**).

19.8 Open Modules and the `opens` Directive

Reflection is a powerful feature in Java that enables inspection and modification of runtime behavior of Java programs. The Reflection API is primarily in the `java.lang.reflect` package. Reference types (classes, interfaces, and enums) can be inspected at runtime to access information about the declarations contained in them (e.g., fields, methods, and constructors)—often referred to as *reflective access*. In fact, classes can be programmatically instantiated and methods can be invoked and values of fields can be changed. An example of using reflection to process annotations in Java programs is provided in **§25.6**, **p. 1587**.

Some frameworks, such as Spring and Hibernate, make heavy use of reflection in the services they provide. They rely on *deep reflection*: being able to use reflection, not only on `public` classes and their `public` members, but also on non-`public` classes and non-`public` members. Note that these frameworks require access for reflection at runtime, and not at compile time.

However, things changed with the introduction of modules in Java, where encapsulation is in the front seat. Reflection is now only allowed on `public` members of `public` types in *exported* packages from a module. In other words, the `exports` directive does not permit reflection on non-`public` types and non-`public` members from exported packages.

The `opens` directive in a module declaration addresses this issue of reflection at runtime, allowing deep reflection on *all* classes and *all* members contained in an *open package*.

Click here to view code image

```
module org.liberal {
  opens org.liberal.sesame;     // An open package
}
```

The `adviceApp` application (**Example 19.1**, **Figure 19.12**) is now modified to illustrate the `opens` directive. We will refer to the modified application as `adviceOpen`. The module `controller` opens the `com.passion.controller` package to enable clients to inspect its class `AdviceController` at runtime, and elicit advice by invoking the method `showAdvice()`. The declaration of the module `controller` specifies this vital information using an `opens` directive, as shown below in the module declaration at (1).

The idea is that module `main` will use reflective access on the `controller` module to elicit advice. Therefore, the `main` module does not *require* the `controller` module, as we can see from the module declaration below at (2).

It is not necessary for a module to explicitly require a module that has open packages for the purposes of reflection. As long as the module with the open packages is available at runtime, reflection can be used to access the contents of the open packages. The emphasis here is on use of the `opens` directive rather than a fullblown treatise on reflection.

Click here to view code image

```
// Filepath: src/controller/module-info.java
module controller {                                  // (1)
  requires model;
  requires view;
  opens com.passion.controller;                     // Open package
}
```

Click here to view code image

```
// Filepath: src/main/module-info.java
module main {}                                       // (2)
```

In **Example 19.2**, the `main()` method at (1) in the class `Main` of the package `com.passion.main` in the module `main` uses reflection to invoke the `showAdvice()` method of the `AdviceController` class in the `com.passion.controller` open package.

The comments in the `main()` method are self-explanatory, but a few things should be noted.

The `forName() static` method of the `Class` class at (2) loads the specified class and returns the `Class` object associated with the class whose name is `com.passion.controller.AdviceController`. Note that module `controller` is not specified. We will make it available when we run the application with the `java` command.

The `Class` object from (2) is used at (3) to obtain the object representing the no-argument constructor of the `AdviceController` class. The `newInstance()` method is invoked at (4) on the constructor object to create an instance of the `AdviceController` class.

The class object from (2) is used at (5) to obtain all `Method` objects that represent methods in the `AdviceController` class. The method `getDeclaredMethods()` returns an array containing `Method` objects reflecting *all* declared methods of the `Advice-Controller` class represented by the `Class` object `cObj`, including `public`, `protected`, default access, and `private` methods in the class. As this class only has one `private` method—`showAdvice()`—index 0 designates this method in the array of `Method` objects, as shown at (6).

The `if` statement at (7) ensures that the `method` reference indeed designates the `private` method `showAdvice()`.

However, before a non-`public` method is invoked via reflection, it is necessary to explicitly suppress checks for language access control at runtime by calling the `setAccessible()` method of the `Method` class with the value `true`, as shown at (8).

At (9) through (12), the `private` method `showAdvice()` designated by the `method` reference is invoked on the `AdviceController` instance specified in the argument to the `invoke()` method, together with an `int` parameter required by the `showAdvice()` method for specific advice.

The code in the `main()` method is enclosed in a `try-catch` construct to catch the checked exceptions that can be thrown by the various methods.

Example 19.2 *Selected Source Code Files in the* `adviceOpen` *Application*

Click here to view code image

```
// File path: src/main/com/passion/main/Main.java
package com.passion.main;
import java.lang.reflect.*;                              // Types for reflection

public class Main {
  public static void main(String... args) {                                  // (1)
    try {
      // Get the runtime object representing the class.
      Class<?> cObj
          = Class.forName("com.passion.controller.AdviceController");        // (2)

      // Get the no-argument constructor of the class.
      Constructor<?> constructor = cObj.getDeclaredConstructor();            // (3)

      // Create an instance of the class.
      Object instance = constructor.newInstance();                           // (4)

      // Get all declared methods, including those that are non-public.
      Method[] methods = cObj.getDeclaredMethods();                          // (5)
      // Class has only one method.
      Method method = methods[0];                                            // (6)
      // Check if it is the right method.
      if (!method.getName().equals("showAdvice")) {                          // (7)
        System.out.printf("Method showAdvice() not found in %s.%n",
```

```
                            cObj.getName());
          return;
        }

        // Disable access control checks on the method as it is a private method.
        method.setAccessible(true);                                          // (8)

        // Invoke the method on the instance, passing any arguments.
        method.invoke(instance, 1);                                          // (9)
        method.invoke(instance, 2);                                          // (10)
        method.invoke(instance, 3);                                          // (11)
        method.invoke(instance, 0);                                          // (12)

      } catch (ClassNotFoundException | NoSuchMethodException |
               InstantiationException | IllegalAccessException |
               IllegalArgumentException | InvocationTargetException ex) {
        ex.printStackTrace();
      }
    }
}
```

Click here to view code image

```
// File path: src/controller/com/passion/controller/AdviceController.java
package com.passion.controller;

import com.passion.model.AdviceModel;               // From model module.
import com.passion.view.AdviceView;                 // From view module.

public class AdviceController {

  private AdviceModel model;
  private AdviceView view;

  public AdviceController() {
    model = new AdviceModel();
    view = new AdviceView(model);
  }

  private void showAdvice(int adviceNumber) {       // (13)
    model.setCurrentAdvice(adviceNumber);
    view.updateAdviceDisplay();
  }
}
```

The accessibility of the `showAdvice()` method at (13) in class `AdviceController` is changed to `private` in **Example 19.2** to illustrate that an open package allows reflection on `private` members.

The other two modules, `model` and `view`, are the same as in the `adviceApp` application (**Example 19.1**, **Figure 19.12**).

We can compile all modules in the `adviceOpen` application, as we did for the `adviceApp`:

Click here to view code image

```
>javac -d mods --module-source-path src --module model,view,controller,main
```

However, trying to run the application throws an exception:

Click here to view code image

```
>java --module-path mods --module main/com.passion.main.Main
java.lang.ClassNotFoundException: com.passion.controller.AdviceController
...
```

The reason for the exception is that the `controller` module has not been resolved at runtime, since it is not required by any module in the application. We can include modules needed by the application using the `--add-modules` option:

Click here to view code image

```
>java --add-modules controller --module-path mods \
      --module main/com.passion.main.Main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

Qualified Opens Directive

Sometimes it is necessary that only certain modules can access an open package for reflection. This can be achieved by using the `opens-to` directive, as shown below at (1), where package `org.liberal.sesame` in module `org.liberal` is *only* open to modules `com.aladdin` and `forty.thieves`. This is known as a *qualified opens* directive. The `to` clause can be used to specify a comma-separated list of modules that can inspect the open package.

Click here to view code image

```
module org.liberal {
  opens org.liberal.sesame to com.aladdin, forty.thieves; // (1) Qualified opens
  opens org.liberal.pandorasbox;                          // (2) Unqualified opens
}
```

Apart from modules `com.aladdin` and `forty.thieves`, no other module can inspect package `org.liberal.sesame` in module `org.liberal`. Any attempt made by other modules to inspect package `org.liberal.sesame` will result in a `java.lang.Illegal-AccessException`. Unqualified open package `org.liberal.pandorasbox`, shown at (2), allows *any* module to inspect this package.

The declaration of module `controller` can be modified to use the qualified `opens` directive, as shown below. The application `adviceOpen` can be compiled and run as before.

Click here to view code image

```
module controller {
  requires model;
  requires view;
  opens com.passion.controller to main;   // Package is only open to main module.
}
```

Open Modules

An *open module* can be declared with the restricted keyword `open` in its declaration file `module-info.java`:

Click here to view code image

```
open module module_name {
  // Any number of module directives, except the opens directive.
}
```

An open module does not declare any open packages, since its declaration implies that all its packages are open. However, note that making a module open also means that it would be possible to use reflection on *all types and their members in all packages* in the module. Making a module open should be done judiciously, and in a way that does not weaken the principle of encapsulation.

The declaration of module `controller` can be modified so that the module is open for other modules, as shown below. The application `adviceOpen` can be compiled and run as before.

Click here to view code image

```
open module controller {              // An open module
  requires model;
  requires view;
}
```

19.9 Services

Programming to interfaces is a powerful design paradigm that advocates writing code using interfaces and abstract classes—that is, using abstract types and not concrete types. This allows new implementations of abstract types to be used by the code if and when necessary. This strategy results in the code being loosely coupled.

The `requires` directives in module declarations create explicit dependencies between concrete modules, requiring explicit naming of modules with heavy reliance on concrete types, thus making the modules tightly coupled. As we shall see, services allow programming to interfaces, but at a higher level of abstraction, resulting in the modules being loosely coupled, as direct dependencies between them are removed.

The main advantage of services is application *extensibility*: adding functionality by providing new services without having to recompile the whole application. This is feasible as services are discoverable at runtime. This design strategy is typically employed in implementing *plugins* which can extend the capabilities of an application.

A *service* is a specific abstract type, typically an interface, but it can be an abstract class as well, that specifies some specific functionality. The following artifacts are typically required to create a service:

- *The service interface*
  The service interface specifies the abstract type that represents the service. It is also known as a *service provider interface*.
- *The service provider*
  A service provider implements the service. There can be zero or more service providers of a service. A service provider advertises the implementation of a specific service with the `provides-with` directive.
- *The service locator*
  The service locator discovers service providers at runtime through a *service loader* which is an instance of the `java.util.ServiceLoader<S>` class, where type parameter `S` designates the type of service to be loaded. A service loader locates and loads service providers at runtime. A service locator announces the services it is interested in with the `uses` directive.
- *The service consumer*

A service consumer utilizes the service. The service consumer accesses a particular implementation of the service via the *service locator*, and does not communicate directly with a service provider. It can choose which implementation of the service to consume from the ones provided by the service locator.

We illustrate the setup necessary to use services by implementing a variant of the `adviceApp` we have seen earlier (**Example 19.1**, **Figure 19.12**). Each artifact required to set up a service will be implemented as a module in our example below, although other configurations are possible, typically where the service locator is merged with either the service interface or the service consumer. The service providers, being autonomous modules, can be added or removed as necessary.

Figure 19.15, **p. 1198** shows the module diagram and the module declarations of the `adviceService` application, and **Example 19.3** to **Example 19.6** show the source code in each module of the application.

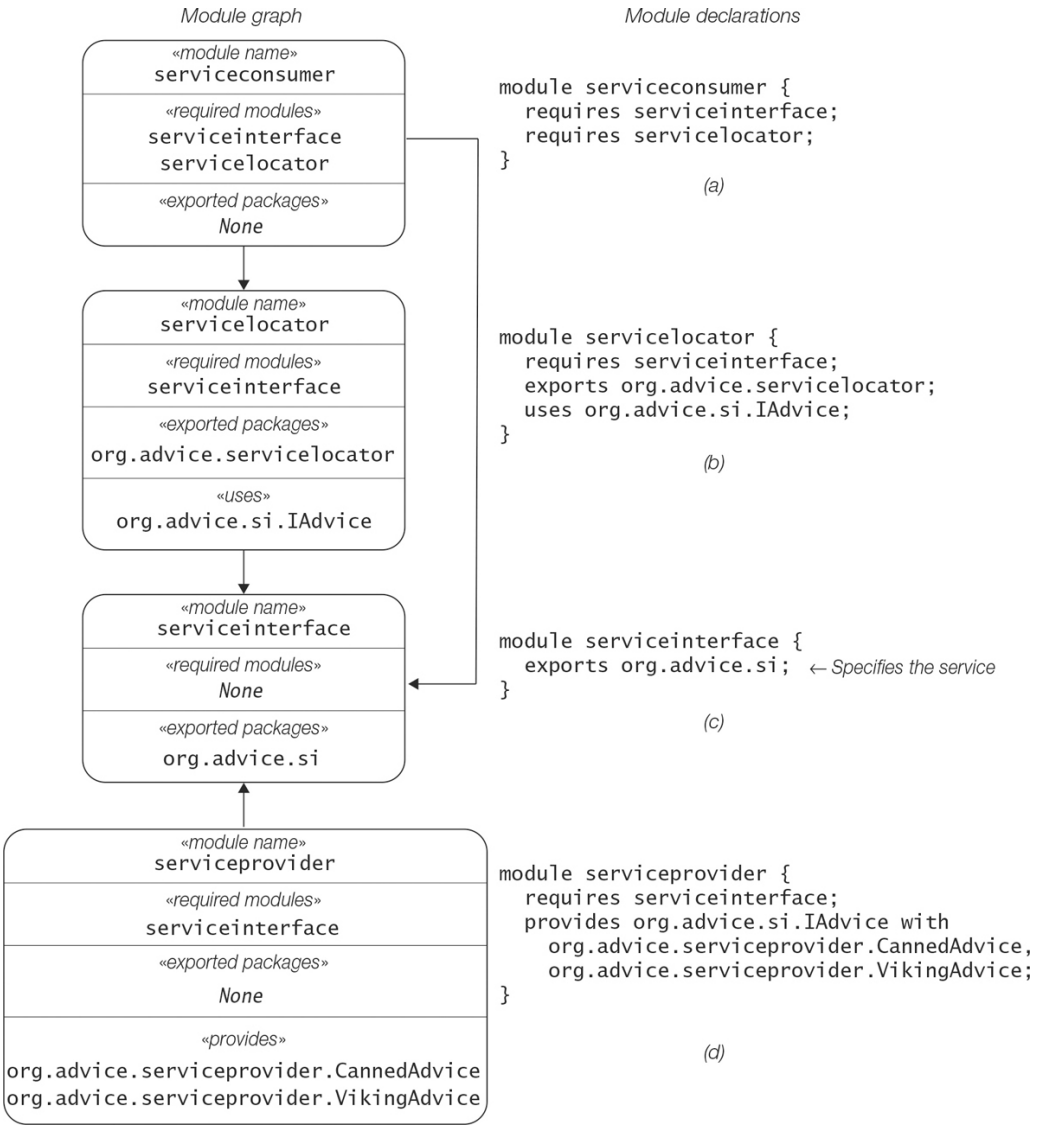


Figure 19.15 *Module Graph and Module Declarations for Services*

Specifying the Service Interface

The service is represented by the interface `IAdvice` that is declared in package `org.advice.si` of the `serviceinterface` module (**Example 19.3**). It specifies the functionality that the service will provide: two abstract methods `getContent()` and `getLocale()` that return the advice (as a string) and the associated locale at (1) and (2), respectively. In other words, the advice service is locale-specific. Since the two methods have no accessibility modifier in their declaration, they are implicitly `public`, and as they do not provide an implementation, they are implicitly `abstract`.

Example 19.3 *Specifying the* `Advice` *Service Interface*

Click here to view code image

```
// File: ./src/serviceinterface/org/advice/si/IAdvice.java
package org.advice.si;
import java.util.Locale;

// Service interface
public interface IAdvice {
  String getContent();                  // (1) Returns the content of the advice.
  Locale getLocale();                   // (2) Returns the associated locale.
}
```

The declaration of the `serviceinterface` module in **Figure 19.15(c)** exports the package `org.advice.si` containing the service interface `IAdvice` so that service providers can implement it.

Click here to view code image

```
// File: ./src/serviceinterface/module-info.java
module serviceinterface {
   exports org.advice.si;
}
```

Implementing Service Providers

Two service providers are implemented in the `serviceprovider` module (**Example 19.4**). The package `org.advice.serviceprovider` contains the class `Canned-Advice` that implements the service interface `IAdvice`. The `getContent()` and the `getLocale()` methods return the canned advice and the associated UK locale at (1) and (2), respectively.

Example 19.4 *Implementing the Advice Service Providers*

Click here to view code image

```
// File:./src/serviceprovider/org/advice/serviceprovider/CannedAdvice.java
package org.advice.serviceprovider;

import org.advice.si.IAdvice;
import java.util.Locale;

public class CannedAdvice implements IAdvice {
  public String getContent() {                          // (1)
    return "Keep calm and service on!";
  }
  public Locale getLocale() { return Locale.UK; }       // (2)
}
```

Click here to view code image

```
// File:./src/serviceprovider/org/advice/serviceprovider/VikingAdvice.java
package org.advice.serviceprovider;

import org.advice.si.IAdvice;
import java.util.Locale;

public class VikingAdvice implements IAdvice {
```

```
  public String getContent() {                                    // (3)
    return "Gi aldri opp!";    // Never give up!
  }
  public Locale getLocale() { return new Locale("no", "Norway"); }  // (4)
}
```

The package `org.advice.serviceprovider` also contains the class `VikingAdvice` that implements the service interface `IAdvice`. The `getContent()` and the `getLocale()` methods return the Viking advice and the associated Norwegian locale at (3) and (4), respectively.

The `serviceprovider` module in **Figure 19.15(d)** requires the module `serviceinterface` in order to implement the service. The module advertizes that it implements the service in a `provides-with` directive at (1) below. In the directive, the service interface that is implemented is specified first, and the `with` clause specifies a comma-separated list of *classes* in this module that implement the service. This information is used by the service loader at runtime to locate and load the classes that implement this service in this module. Note that the package implementing the service is not exported, avoiding any service consumers having explicit dependency on service providers.

Click here to view code image

```
// File: ./src/serviceprovider/module-info.java
module serviceprovider {
  requires serviceinterface;

  provides org.advice.si.IAdvice with              // (1)
    org.advice.serviceprovider.CannedAdvice,
    org.advice.serviceprovider.VikingAdvice;
}
```

Implementing the Service Locator

The class `java.util.ServiceLoader<S>`, where the type parameter `S` designates the type of service, is at the core of finding and loading service providers for the service `S`. A service loader for a specific service is created by calling the static method `load()` of the `ServiceLoader` class, passing the runtime `Class` object of the service interface, as shown below. Typically, a stream is then created by calling the `stream()` method on the service loader. This stream can be used to lazily load the available service providers for the specified service `S`. The type of an element in this stream is the static member interface `ServiceProvider.Provider<S>`.

Click here to view code image

```
ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);
Stream<ServiceLoader.Provider<IAdvice>> spStream = loader.stream();
```

The procedure above allows processing of stream elements as type `Provider<S>` without instantiating the service provider. The `get()` method of the `Provider<S>` interface must be invoked on a stream element of type `Provider<S>` to obtain an instance of the service provider —that is, the class that implements the service interface `S`. Given the service provider, the service implemented by the provider can be utilized.

Click here to view code image

```
static <S> ServiceLoader<S> load(Class<S> service)
```

This static method creates a service loader for the given service type specified by its runtime object, using the current thread's context class loader.

Click here to view code image

```
Stream<ServiceLoader.Provider<S>> stream()
```

Returns a stream that lazily loads available providers of this loader's service (designated by type parameter `S`). The stream elements are of type `Service-Loader.Provider<S>`.

The interface `Provider<S>` is a `static` member interface of the class `Service-Loader<S>` that extends the `java.util.function.Supplier<S>` functional interface. The `get()` method of the `Provider<S>` interface must be invoked to get or instantiate the service provider for the service `S`.

A service locator for the advice service is implemented by the class `org.advice.servicelocator.AdviceLocator` in the `servicelocator` module (**Example 19.5**). The class provides two static methods, `getAdvice()` and `getAllAdvice()`, which return a service provider that implements the `IAdvice` service interface for a particular locale or *all* service providers that implement the `IAdvice` service interface, as shown at (1) and (8), respectively. In other words, the former returns a service provider for a particular locale and the latter returns all service providers that implement the `IAdvice` service interface.

The method `getAdvice()` at (1) is passed the desired locale. Its return type is `Optional<IAdvice>`, as there might not be any service provider for this service for the desired locale. At (2), a service loader is created for the `IAdvice` service interface. A stream is built with the service loader as the source at (3). This stream of element type `Provider<IAdvice>` is processed to find a service provider for the desired locale. Note how at (5), the stream is converted from type `Stream<ServiceLoader.Provider<IAdvice>>` to `Stream<IAdvice>` by the `map()` intermediate operation that applies the `get()` method of the `Provider<S>` interface to each element. The resulting stream is filtered at (6) for a service provider with the desired locale. The first service provider that has the desired locale is selected by the `findFirst()` terminal operation at (7). This operation returns an `Optional<IAdvice>` as there might not be such a service provider in the stream.

Analogously, the method `getAllAdvice()` at (8) collects all service providers for the `IAdvice` service interface in a list and returns them. In this case, an empty list will indicate that no service providers were found.

Example 19.5 *Implementing the Advice Service Locator*

Click here to view code image

```
// File:./src/servicelocator/org/advice/servicelocator/AdviceLocator.java
package org.advice.servicelocator;

import java.util.*;
import java.util.stream.*;
import org.advice.si.*;

public class AdviceLocator {

  /** Get advice for a particular locale. */
  public static Optional<IAdvice> getAdvice(Locale desiredLocale) {     // (1)
    ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);  // (2)
    Stream<ServiceLoader.Provider<IAdvice>> spStream = loader.stream(); // (3)
```

```
    Optional<IAdvice> optAdvice = spStream                              // (4)
      .map(ServiceLoader.Provider::get)                  // (5) Stream<IAdvice>
      .filter(a -> desiredLocale.equals(a.getLocale()))  // (6)
      .findFirst();                                      // (7)
    return optAdvice;
  }

  /** Get all advice implemented by all providers. */
  public static List<IAdvice> getAllAdvice() {           // (8)
    ServiceLoader<IAdvice> loader = ServiceLoader.load(IAdvice.class);
    List<IAdvice> allAdvice = loader
      .stream()
      .map(p -> p.get())                 // (9) map(ServiceLoader.Provider::get)
      .collect(Collectors.toList());     // (10) List<IAdvice>
    return allAdvice;
  }
}
```

At (1) below, the `servicelocator` module (**Figure 19.15(b)**) requires the `serviceinterface` module in order to locate its service providers, and it exports the package `org.advice.servicelocator` at (2) so that service consumers can locate service providers. The `uses` directive at (3) declares which service (`org.advice.si.IAdvice`) is used by this module, allowing it to look up this particular service at runtime.

Click here to view code image

```
// File: ./src/servicelocator/module-info.java
module servicelocator {
  requires serviceinterface;          // (1)
  exports org.advice.servicelocator;  // (2)
  uses org.advice.si.IAdvice;         // (3) In module serviceinterface.
}
```

Implementing the Service Consumer

The class `AdviceConsumer` implements a simple service consumer for the `IAdvice` service in the package `org.advice.serviceconsumer` of the `serviceconsumer` module (**Example 19.6**).

The code at (1) shows how advice for the UK locale is obtained via the `org.advice.servicelocator.AdviceLocator` class in the `servicelocator` module. The static method `getAdvice()` of the `AdviceLocator` class at (2) returns an `Optional<IAdvice>` object. This object can be queried to extract the advice at (3) with the `getContent()` method, if there is one. Otherwise, the `orElse()` method at (4) provides an appropriate message.

The code at (5) shows how all advice providers for the `IAdvice` service can be obtained via the `AdviceLocator` class. The static method `getAllAdvice()` of the `AdviceLocator` class at (6) returns a `List<IAdvice>` object with all providers for this service. A stream on this list is used at (7) to print the advice from each provider by extracting the advice with the `getContent()` method.

Example 19.6 *Implementing of the Advice Service Consumer*

Click here to view code image

```
// File: ./src/serviceconsumer/org/advice/serviceconsumer/AdviceConsumer.java
package org.advice.serviceconsumer;

import java.util.*;
```

```
import org.advice.servicelocator.*;
import org.advice.si.*;

public class AdviceConsumer {
  public static void main(String[] args) {

    // Get advice for the UK locale.                                    // (1)
    Optional<IAdvice> optAdvice = AdviceLocator.getAdvice(Locale.UK);   // (2)
    String adviceStr = optAdvice.map(IAdvice::getContent)               // (3)
                                .orElse("Sorry. No Advice!");           // (4)
    System.out.println("Advice for UK locale: " + adviceStr);

    // Get all implemented advice.                                      // (5)
    System.out.println("Printing all advice:");
    List<IAdvice> allAdvice = AdviceLocator.getAllAdvice();             // (6)
    allAdvice.stream()                                                  // (7)
      .forEach(a -> System.out.println(a.getContent()));
  }
}
```

The `serviceconsumer` module (**Figure 19.15(a)**) only requires the module that declares the service, as shown at (1), and the module that can locate service providers for this service as at (2). Note that the consumer module does not directly depend on any service providers.

Click here to view code image

```
// File: ./src/serviceconsumer/module-info.java
module serviceconsumer {
  requires serviceinterface;     // (1)
  requires servicelocator;       // (2)
}
```

Compiling and Running a Service

The directory structure of the `adviceService` application is analogous to that of the `adviceApp` (**Figure 19.9**, **p. 1181**).

We can compile all modules in the `adviceService` application as before. However, note that compilation of the `serviceprovider` module does not require any other modules except the `serviceinterface` module.

Click here to view code image

```
>javac -d mods \
   --module-source-path src \
   --module serviceprovider,serviceconsumer,servicelocator,serviceinterface
```

We can run the application as before:

Click here to view code image

```
>java --module-path mods \
      --module serviceconsumer/org.advice.serviceconsumer.AdviceConsumer
Advice for UK locale: Keep calm and service on!
Printing all advice:
```

```
Keep calm and service on!
Gi aldri opp!
```

We leave it as an exercise to experiment other configurations of implementing services. In particular, merging the service locator with the service consumer is highly recommended.

19.10 Creating Runtime Images

The *Java Linker* tool, `jlink`, can be used to assemble a customized *runtime image* of a modular application. This image only includes the modules in the application and their dependencies, together with a minimal JRE (Java Runtime Environment) to run the application.

We only present the basic use of the `jlink` tool, and strongly encourage consulting the documentation for the JDK tools for more in-depth coverage.

At a minimum, the `jlink` tool requires the following information to create a runtime image:

- The *module path* given by the `--module-path` option (*short form*: `-p`) to find modules that are to be included in the runtime image.
- The *names of the modules* that should be included in the runtime image. These modules are specified using the `--add-modules` option, and are looked up in the module path. If any standard or JDK modules required by the application are not specified, they are automatically included by the tool. The dependencies between the modules are automatically resolved.
- The *output directory* in which to create the runtime image, specified with the `--output` option.

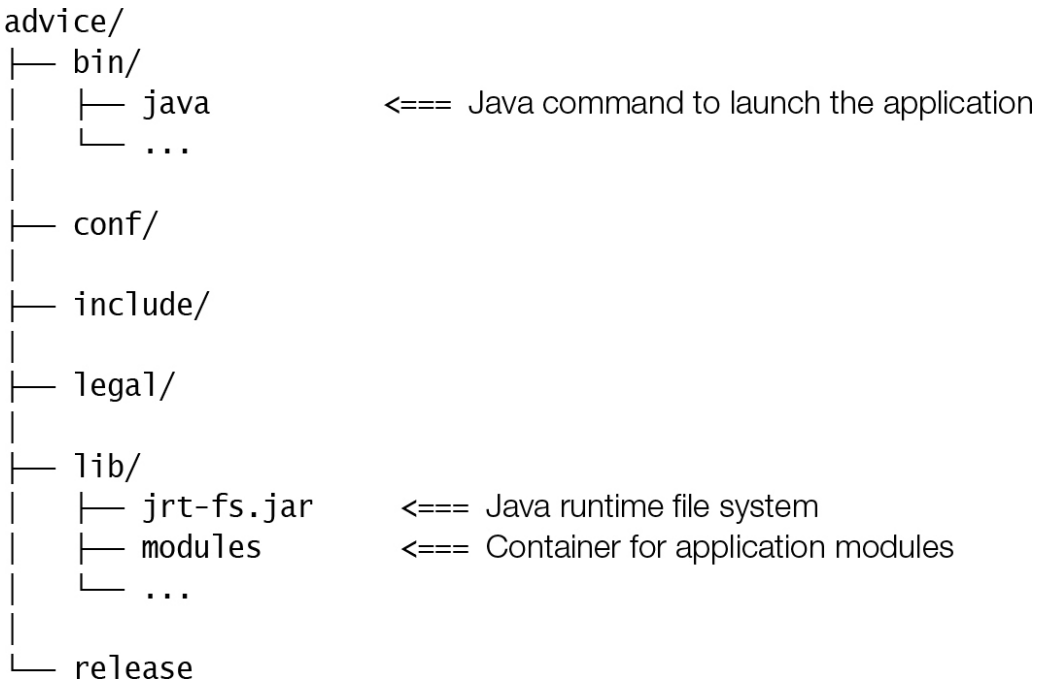
The following command creates a runtime image of the `adviceApp` in the `advice` directory, assuming that the JAR files for the application modules can be found in the directory `mlib`:

Click here to view code image

```
jlink --module-path mlib --add-modules model,view,controller,main,java.base \
      --output advice
```

The standard module `java.base` is explicitly specified. If omitted, it will be automatically included by the `jlink` tool.

The output directory `advice` containing the runtime image has the following structure, shown below with some of the files in the runtime image. The size of this particular runtime image is approximately 39 MB. Note the `java` command to launch the application under the directory `advice/bin`.



The modules in the runtime image `advice` can be listed by the following command:

Click here to view code image

```
>advice/bin/java --list-modules
controller
java.base@17.0.2
main
model
view
```

The application can be run from the runtime image by the following command, as the entry point of the application is specified in the `main` modular JAR included in the runtime image:

Click here to view code image

```
>advice/bin/java --module main
See no evil.
Speak no evil.
Hear no evil.
No advice.
```

A runtime image is platform specific, and thus meant to be run on the platform it was created for. As the runtime image contains platform-specific files, it will most likely not run on other platforms.

It is not possible to automatically update a runtime image with a newer version of Java. The runtime image must be rebuilt with the appropriate version of Java.

Runtime images are beneficial in many ways: They have a smaller memory footprint, can boost performance, are more secure, and are easier to maintain. Not surprisingly, runtime images are ideal for running on smaller devices.

19.11 Categories of Modules

The Java Module System is designed to allow both non-modular and modular code to work together. Types are usually bundled in JAR files. Regardless of whether it a plain JAR or a modular JAR, how its content is handled by the module system depends on the path on which it is

placed when using the JDK tools: *the class path* (§6.4, p. 337) or *the module path* (p. 1186). Many JDK tools, like `javac` and `java` commands, allow both paths to be specified on the command line in order to mix non-modular and modular code. Interpretation of JARs placed on either of these two paths is shown in Table 19.4, and is the subject of this section. Understanding this distinction is also essential for migrating code into modules (p. 1209).

Table 19.4 *How JARs Are Handled Depends on the Path*

Path	Plain JAR	Modular JAR
On the class path: `-cp`	Included in the *unnamed module*	Included in the *unnamed module*
On the module path: `-p`	Treated as an *automatic module*	Treated as an *explicit module*

Searching for types on the class path is different from searching for types on the module path. Class path search is a linear search on the entire class path, finding the first occurrence of a type and terminating if the type is found. Module path search is according to the module dependencies and therefore more efficient.

The Unnamed Module

Types from any JAR that is placed on the class path are included in *the unnamed module*, regardless of whether it is a plain or a modular JAR. If it is a modular JAR, its module descriptor is ignored. However, note that a unique unnamed module is associated with each class loader. Classes loaded by a class loader via the class-path are members of the unnamed module associated with that class loader. Colloquially we refer to *the unnamed module* as the one associated with the application class loader.

The unnamed module is a catchall solution to capture code that is on the class path. It is analogous to code not declared in packages being part of the unnamed package. It is a compatibility feature that allows code on the class path to work with modules on the module path.

Types in the unnamed module can access code from both the class path and the module path—that is, the unnamed module can read *all* modules.

All packages in the unnamed module are exported and open for reflection. Types in the unnamed module are accessible to other types in the unnamed module, and also accessible to *automatic* modules, but only by reflection to explicit modules. An explicit module on the module path cannot access code in the unnamed module, as the unnamed module cannot be specified in a `requires` directive because it has no name. Access to types in the unnamed module is governed by accessibility rules for types in packages—that is, only `public` types in a package are accessible to other packages in the unnamed module and in any automatic modules.

Automatic Modules

A plain JAR—that is, a JAR that does not have a `module-info.class` file in its top-level directory—defines an *automatic module* when placed on the module path. An automatic module has a module name which is determined according to the scheme described below.

An automatic module can read all other modules, including the unnamed module. This means that if an explicit module reads a single automatic module, all other automatic modules can be implicitly read by the explicit module. This avoids having a `requires` directive to every automatic module required by a named module.

All packages in the automatic module are exported and open for reflection. Access to its types is solely determined by the accessibility rules for types in packages. Explicit modules can read automatic modules by specifying them in the `requires` directives, as they have a name.

An automatic module is a migration feature. Resorting to an automatic module should be a temporary measure. As such, a module should be migrated to an explicit module by defining an appropriate module descriptor.

Scheme for Naming Automatic Modules

If the JAR file has the attribute `Automatic-Module-Name` defined in its `META-INF/MANIFEST.MF` file (p. 1212), its value is used as the module name. This is the recommended practice, as it avoids issues like name clashes.

If the JAR file does not have the attribute `Automatic-Module-Name` defined in its `META-INF/MANIFEST.MF` file, its name is derived from the *file name* of the JAR file according to the following algorithm:

- Remove the `.jar` suffix from the name.
- Remove any version information at the end of the name, which is usually specified in the version number format after a hyphen (`-`); for example, `-1.2.3-ERC`.
- All non-alphanumeric characters are replaced with a dot (`.`), any sequence of dots is replaced with a single dot, and all leading and trailing dots are removed. This step avoids constructing a module name with illegal characters.

As an example, the JAR file name `ying-yang.jar` of an automatic module will derive the module name `ying.yang`, as will the JAR file name `ying-yang-1.2.3-ERC.jar`. File names for plain JARs acting as automatic modules should be chosen with care so as to avoid name clashes when such JARs do not specify a unique module name in their manifest file.

Explicit Modules

As the name implies, an explicit module is described by its module descriptor that explicitly specifies its name and the modules it reads (i.e., its dependencies), including any packages it exports or opens, or any services it provides or uses. A modular JAR is treated as an explicit module when specified on the module path.

Access to types in an explicit module is determined by the accessibility rules for types in packages *exported* or *opened* by the explicit module.

An explicit module *cannot* access code in the unnamed module as it cannot refer to the unnamed module in its module declaration.

Named Modules

Explicit modules and automatic modules are characterized as *named modules*. In the case of the explicit module, the name is specified in the module declaration. For the automatic modules, the name is either in the manifest file of its JAR or generated from its file name. A plain JAR or a modular JAR is treated as a named module when specified on the module path. Comparison of modules is shown in **Table 19.5**.

Table 19.5 *Module Comparison*

		Named modules	
Criteria	**Unnamed module**	**Automatic module**	**Explicit module**
JAR	Plain or modular JAR	Plain JAR	Modular JAR
Path	On class path	On module path	On module path
Name	Not applicable	Either in manifest file or from file name	Specified in module descriptor
Module descriptor	Not applicable	Not applicable	Mandatory
Reads	All modules	All modules	Only named modules that are required
Exports/Opens	All packages	All packages	Only exported/open packages

19.12 Migrating to Modules

The module system allows incremental migration of non-modular code to modules. Knowledge of dependencies between types in the plain JARs is crucial to drive this migration. This is probably easier with an application whose code one has control over than it is with libraries and frameworks from third parties. Regardless, having a better handle on dependencies makes the process easier. The `jar` tool (**p. 1211**) is useful to obtain information about the contents of a JAR and the `jdeps` tool (**p. 1214**) is a great tool to discover dependencies in the code.

The scenarios presented below are very much idealized, and deal with code in plain and modular JARs. The basic idea is to move code from the class path to the module path, either converting plain JARs to explicit modules directly or via automatic modules.

Bottom-Up Strategy for Code Migration

If all *direct* dependencies of a plain JAR are known to be modules, the plain JAR can be directly converted to an explicit module by declaring their dependencies and exports in a module declaration. This idea is embodied in the following algorithm, based on the graph of dependencies between the plain JARs (see **Figure 19.16**).

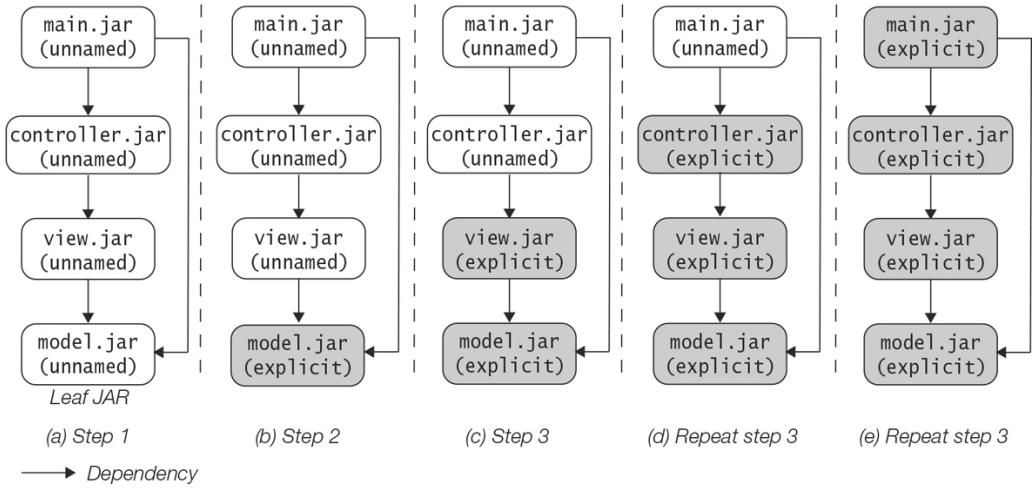


Figure 19.16 *Bottom-Up Strategy for Code Migration*

- (1) Place all plain JARs on the *class path* such that they are in the unnamed module.
- (2) Choose *leaf* JARs from the class path—that is, JARs that do not have any dependencies. Turn them into explicit modules by adding appropriate `exports` directives in their module descriptors and move them from the class path to the module path.
- (3) From the class path, choose plain JARs from the next higher level, that have direct module dependencies. Turn them into explicit modules. Their module declarations must specify `requires` directives on their dependencies and appropriate `exports` directives for packages they export. Move the newly created explicit modules from the class path to the *module path*.
- (4) Repeat step (3) until the unnamed module is empty.

There are variations on the bottom-up strategy, but the gist is represented by the algorithm above. Note that when applying the bottom-up strategy for migration, the explicit module on the module path never accesses code in any plain JAR on the class path.

Figure 19.16 shows the bottom-up strategy applied to the plain JARs: `main.jar`, `controller.jar`, `view.jar`, and `model.jar`, that are converted to modular JARs—that is, explicit modules—as they are moved from the class path to the module path.

Top-Down Strategy for Code Migration

Migrating to modular code via *automatic* modules is convenient when it will take too long to wait for all dependencies to be available. The algorithm below migrates the code into explicit modules via automatic modules, based on the graph of known dependencies between the plain JARs (see **Figure 19.17**).

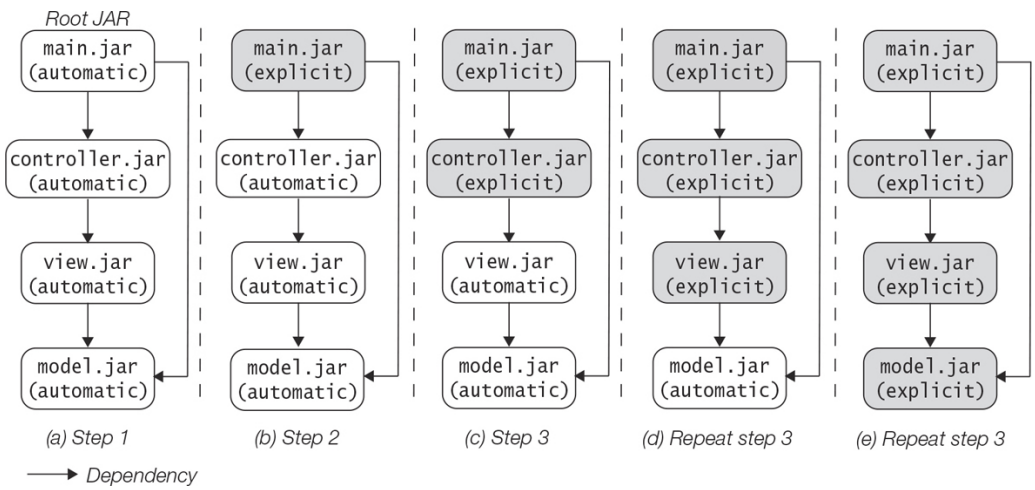


Figure 19.17 *Top-Down Strategy for Code Migration*

(1) Place all plain JARs on the module path. All plain JARs are now automatic modules.

(2) Choose *root* JARs that are automatic modules. A root JAR is one that no other JAR depends on. Turn these automatic modules into explicit modules by adding module declarations. Add `requires` directives using names of the automatic modules that a root JAR depends on.
(3) Choose automatic modules at the next lower level that are directly required by their parent modular JARs. Turn these automatic modules into explicit modules. Add `requires` directives using names of the automatic modules that these JARs depend on. Add appropriate `exports` directives for any packages exported.
(4) Repeat step (3) until all automatic modules have been converted to explicit modules.

Again, there are variations to the top-down strategy, but the algorithm embodies the main idea of this strategy. Note that in real life, both strategies will most likely be applied to complete the migration.

Figure 19.17 shows the top-down strategy applied to the plain JARs: `main.jar`, `controller.jar`, `view.jar`, and `model.jar`, which are first placed on the module path as automatic modules and then are converted to modular JARs—that is, explicit modules.

19.13 Exploring Modules

This section provides an introduction to using the JDK tools to discover, explore, and analyze modular applications.

Given the JARs `main.jar`, `control.jar`, `view.jar`, and `model.jar` for the `adviceApp` application in the `mlib` directory in **Figure 19.14(c)**, **p. 1188**, we will use JDK tools to explore these archives for miscellaneous information about this application.

Listing the Contents of a JAR

We can list the contents of a JAR using the `--list` option (*short form*: `-t`) of the `jar` tool. The JAR file is specified with the `--file` option (*short form*: `-f`). The contents of `main.jar` are printed by the following `jar` command:

Click here to view code image

```
>jar --list --file mlib/main.jar
META-INF/
META-INF/MANIFEST.MF
module-info.class
com/
com/passion/
com/passion/main/
com/passion/main/Main.class
```

From the output on the terminal, we can see that there is a `META-INF` directory listed, in addition to the module descriptor file (`module-info.class`), the directory hierarchy of the package `com.passion.main`, and the file `Main.class` contained in it.

The `META-INF` directory is used internally to record any metadata in the JAR archive. In the listing above, the `META-INF` directory contains the file `MANIFEST.MF`. This manifest file is used to store any necessary information (specified as attribute-value pairs)—for example, the `Main-Class` attribute to designate the entry point of the application.

Extracting the Contents of a JAR

The `jar` tool can be used to extract the contents of a JAR into a desired directory. We illustrate the extract operation on `main.jar`. The following commands create the `main-extracted` di-

rectory under the current directory and change the working directory to it, respectively.

```
>mkdir main-extracted
>cd main-extracted
```

The `jar` command below with the `--extract` option (*short form*: `-x`) will extract the contents of `main.jar` into the `main-extracted` directory, which is now the current working directory. The structure of the `main-extracted` directory is also shown below after the extraction (printed using the `tree` command), showing the exploded module structure and the `META-INF` directory from `main.jar`.

Click here to view code image

```
>jar --extract --file ../mlib/main.jar
>tree -F --dirsfirst --noreport main-extracted
main-extracted
├── META-INF/
│   └── MANIFEST.MF
├── com/
│   └── passion/
│       └── main/
│           └── Main.class
└── module-info.class
```

The contents of the `MANIFEST.MF` file, listed below using the `more` command, show that the entry point of the application is given by the value of `Main-Class`—that is, the class `com.passion.main.Main`.

Click here to view code image

```
>more main-extracted/META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: 17.0.2 (Oracle Corporation)
Main-Class: com.passion.main.Main
```

Listing All Observable Modules

The `java` command with the `--list-modules` option (*no short form*) lists the *system modules* that are installed in the JDK, and then exits. These modules are available to every application. This lengthy list gives an idea of how the JDK has been modularized. The `java` command lists each module with its version; in this case, it is Java 17.0.2. Module names starting with the prefix `java` implement the Java SE Language Specification, whereas those starting with `jdk` are JDK specific. The reader will no doubt recognize some names, specially `java.base`.

The system modules are found in the `jmods` directory under the installation directory of the JDK. These are *JMOD files* having a module name with the extension `".jmod"`. JMOD files have a special non-executable format that allows native binary libraries and other configuration files to be packaged with bytecode artifacts that can then be linked to create runtime images with the `jlink` tool (p. 1222).

```
>java --list-modules
java.base@17.0.2
...
java.se@17.0.2
...
jdk.compiler@17.0.2
...
```

```
jdk.jartool@17.0.2
...
```

Specifying a module path in the `java` command below not only lists the system modules, but also the modular JARs found in the specified module path—in other words, all *observable modules*. In the `java` command below, the absolute path of all JARs found in the `mlib` directory is listed last in the output.

Click here to view code image

```
>java --module-path mlib --list-modules
java.base@17.0.2
...
jdk.javadoc@17.0.2
...
jdk.jdeps@17.0.2
...
controller file:.../adviceApp/mlib/controller.jar
main file:.../adviceApp/mlib/main.jar
model file:.../adviceApp/mlib/model.jar
view file:.../adviceApp/mlib/view.jar
```

Describing the Module Descriptor of a JAR

Both the `java` tool and the `jar` tool have the `--describe-module` option (*short form*: `-d`) to show the information contained in the module descriptor of a JAR. Both commands are used respectively below.

Click here to view code image

```
>java --module-path mlib --describe-module main
main file:.../adviceApp/mlib/main.jar
requires java.base mandated
requires controller
contains com.passion.main

>jar --file mlib/main.jar --describe-module
main jar:file:.../adviceApp/mlib/main.jar/!module-info.class
requires controller
requires java.base mandated
contains com.passion.main
main-class com.passion.main.Main
```

Note that in the `java` command, the `--describe-module` option requires a module name, whereas that is not the case in the `jar` command, where the `--file` option specifies the modular JAR.

Since the module descriptor is a Java bytecode class file, it must be disassembled to display its information. In both cases, first the module name is printed, followed by the path of the JAR. In the case of the `jar` command, the name `module-info.class` is appended to the path of the JAR. In both cases, the names of modules required (`java.base` and `controller`) are reported. The `main` module also *contains* an *internal* package (`com.passion.main`). Not surprisingly, the `java.base` module is mandated. However, only the `jar` command reports that the main-class is `com.passion.main.Main`. The `jar` command is useful to find the entry point of an application.

The `--describe-module` option (*short form*: `-d`) can also be used on a system module. The following `java` command describes the module descriptor of the `java.base` system module. The

command lists all modules the `java.base` module *exports*, *uses*, *exports qualified*, and *contains* (p. 1177). As can be expected from its status as a mandated module for all other modules, it does *not require* any module.

Click here to view code image

```
>java --describe-module java.base
java.base@17.0.2
exports java.io
exports java.lang
...
uses java.util.spi.CurrencyNameProvider
uses java.util.spi.TimeZoneNameProvider
...
qualified exports jdk.internal to jdk.jfr
qualified exports sun.net.www to java.desktop java.net.http jdk.jartool
...
contains com.sun.crypto.provider
contains com.sun.java.util.jar.pack
...
```

Viewing Dependencies

The *Java Dependency Analysis tool*, `jdeps`, is a versatile command-line tool for static analysis of dependencies between Java artifacts like class files and JARs. It can analyze dependencies at all levels: module, package, and class. It allows the results to be filtered and aggregated in various ways, even generating various graphs to illustrate its findings. It is an indispensable tool when migrating non-modular code to make use of the module system.

In this section, we confine our discussion to modular code—that is, either an exploded module directory with the compiled module code (e.g., `mods/main`) or a modular JAR (e.g., `mlib/main.jar`).

In this section, the results from some of the `jdeps` commands have been edited to fit the width of the page without any loss of information, or elided to shorten repetitious outputs.

Viewing Package-Level Dependencies

The `jdeps` command with no options, shown below, illustrates the *default behavior* of the tool. Line numbers have been added for illustrative purposes.

When presented with the root module directory `mods/model` (1), the default behavior of `jdeps` is to print the name of the module (2), the path to its location (3), its module descriptor (4), followed by its module dependencies (5), and lastly the *package-level* dependencies (6). The package-level dependency at (6) shows that the package `com.passion.model` in the `model` module depends on the `java.lang` package in the proverbial `java.base` module.

Click here to view code image

```
(1)  >jdeps mods/model
(2)  model
(3)  [file:.../adviceApp/mods/model/]
(4)     requires mandated java.base (@17.0.2)
(5)  model -> java.base
(6)     com.passion.model           -> java.lang           java.base
```

The following `jdeps` command if let loose on the `model.jar` archive will print the same information for the `model` module, barring the difference in the file location:

```
>jdeps mlib/model.jar
```

However, the following `jdeps` command with the `main.jar` archive gives an error:

Click here to view code image

```
>jdeps mlib/main.jar
Exception in thread "main" java.lang.module.FindException: Module controller
not found, required by main
       at java.base/...
```

Dependency analysis cannot be performed on the `main` module by the `jdeps` tool because the modules the `main` module depends on cannot be found. In the case of the `model` module, which does not depend on any other user-defined module, the dependency analysis can readily be performed.

The two options `--module-path` (*no short form*) and `--module` (*short form*: `-m`) in the `jdeps` command below unambiguously specify the location of other modular JARs and the module to analyze, respectively. The format of the output is the same as before, and can be verified easily.

Click here to view code image

```
>jdeps --module-path mlib --module main
main
 [file:.../adviceApp/mlib/main.jar]
   requires controller
   requires mandated java.base (@17.0.2)
main -> controller
main -> java.base
   com.passion.main     -> com.passion.controller     controller
   com.passion.main     -> java.lang                  java.base
```

However, if one wishes to analyze all modules that the specified module depends on recursively, the `--recursive` option (*short form*: `-R`) can be specified. The output will be in the same format as before, showing the package-level dependencies for each module. The output from the `jdeps` command for each module is elided below, but has the same format we have seen previously.

Click here to view code image

```
>jdeps --module-path mlib --module main --recursive
controller
...
main
...
model
...
view
...
```

Viewing Module Dependencies

If the default output from the `jdeps` command is overwhelming, it can be filtered. The `-summary` option (*short form*: `-s`) will only print the *module dependencies*, as shown by the `jdeps` command below. Only the module dependencies of the `main` module will be shown in the output.

Click here to view code image

```
>jdeps --module-path mlib --module main -summary
main -> controller
main -> java.base
```

If we use the `--recursive` option (*short form*: `-R`) with the `-summary` option (*short form*: `-s`), then the module dependencies of each module will be printed recursively, starting with the specified module.

Click here to view code image

```
>jdeps --module-path mlib --module main -summary --recursive              (1)
controller -> java.base
controller -> model
controller -> view
main -> controller
main -> java.base
model -> java.base
view -> java.base
view -> model
```

Finally, we illustrate the graph-generating capabilities of the `jdeps` tool. The `jdeps` command takes all JARs of the `adviceApp` application from the `mlib` directory and creates a module graph (options `-summary` and `--recursive`) in the `DOT` format (option `-dotoutput`) under the current directory. The `DOT` file `summary.dot` containing the graph will be created. Using the `dot` command, this graph can be converted to a `pdf` file (`summary.pdf`) as shown in **Figure 19.18**.

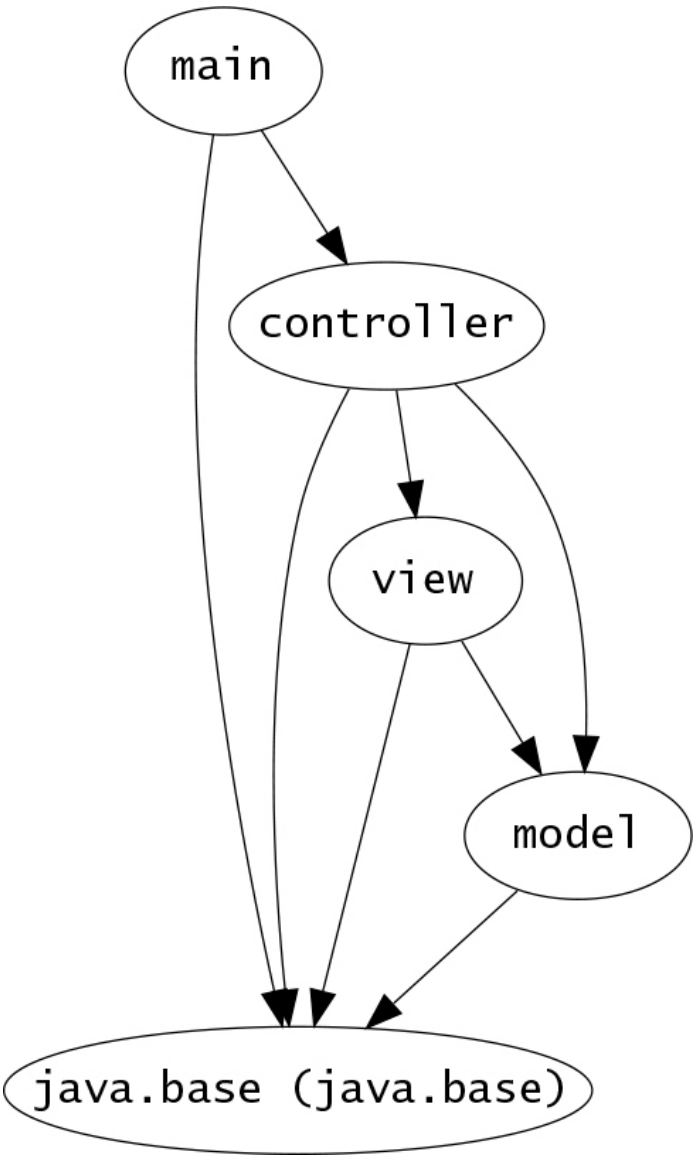


Figure 19.18 *Module Graph Using the* `jdeps` *Tool*

Click here to view code image

```
>jdeps -dotoutput . -summary --recursive  mlib/*
>dot -Tpdf summary.dot >summary.pdf
```

The module graph in **Figure 19.18** shows the same module dependencies printed by the `jdeps` command above at (1). Comparing the module graph in **Figure 19.18** with the one in **Figure 19.8**, we see that `jdeps` has added the *implicit dependency* of each module on the `java.base` module.

Viewing Class-Level Dependencies

It is possible to dive deeper into dependencies with the `jdeps` tool. The `-verbose` option (*short form*: `-v`) will elicit the *class dependencies* of the specified module. Instead of package dependencies to round off the output, class dependencies are listed. The last line in the output shows that the class `com.passion.main.Main` in the `main` module depends on the `java.lang.String` class in the `java.base` module.

Click here to view code image

```
>jdeps --module-path mlib --module main -verbose
main
 [file:.../adviceApp/mlib/main.jar]
   requires controller
```

```
    requires mandated java.base (@17.0.2)
main -> controller
main -> java.base
   com.passion.main.Main  -> com.passion.controller.AdviceController  controller
   com.passion.main.Main  -> java.lang.Object                         java.base
   com.passion.main.Main  -> java.lang.String                         java.base
```

If we use the `--recursive` option (*short form*: `-R`) with the `-verbose` option (*short form*: `-v`), then the class dependencies of each module will be printed recursively, starting with the specified module.

Click here to view code image

```
>jdeps --module-path mlib --module main -verbose --recursive
controller
...
main
...
model
...
view
...
```

19.14 Summary of Selected Operations with the JDK Tools

Table 19.6 provides a ready reference for commands to perform miscellaneous operations using the JDK tools introduced in this chapter. Short forms for the options are shown where appropriate. Use the indicated reference for each operation to learn more about it. The only way to become familiar with these tools is to use them.

Table 19.6 *Selected Operations with the JDK Tools*

Operation	Command

Compiling modular code (p. 1186)	Click here to view code image
	`javac --module-path modulepath -d directory sourceAndModuleInfoFiles`
	`javac  -p  modulepath -d directory sourceAndModuleInfoFiles`
	`javac --module-source-path modulepath -d directory \`
	`  --module moduleName1,...`
	`javac --module-source-path modulepath -d directory \`
	`  -m moduleName1,...`

Launching modular application (p. 1189)	Click here to view code image
	`java --module-path modulepath --module moduleName/qualifiedClassName`
	`java  -p  modulepath  -m  moduleName/qualifiedClassName`
	`java --module-path modulepath  -module moduleName`
	`java  -p  modulepath  -m  moduleName`

Creating and listing a modular JAR ar-	Click here to view code image
	`jar --verbose --create --file jarfile -C directory files`
	`jar -vcf jarfile -C directory .`
	`jar -vcf jarfile --main-class qualifiedMainClassName -C directory .`

Operation	Command

chive (p. 1189)	`jar -vcfe jarfile qualifiedMainClassName -C directory .`
	
	`jar --list --file jarfile`
	`jar -tf jarfile`

Listing available modules (p. 1212)	Click here to view code image
	
	`java --list-modules`
	`java --module-path modulepath --list-modules`
	`java  -p           modulepath --list-modules`

Describing a module—that is, printing the module descriptor (p. 1213)	Click here to view code image
	
	`java --module-path modulepath --describe-module moduleName`
	`java  -p           modulepath  -d               moduleName`
	
	`jar --file jarFile --describe-module`
	`jar  -f    jarFile  -d`

Viewing package-level dependencies (p. 1214)	Click here to view code image
	
	`jdeps --module-path modulepath -m moduleName`
	`jdeps --module-path modulepath -m moduleName --recursive`
	`jdeps --module-path modulepath -m moduleName -R`

Viewing module dependencies (p. 1216)	Click here to view code image
	
	`jdeps --module-path modulepath -m moduleName -summary`
	`jdeps --module-path modulepath -m moduleName -summary --recursive`
	`jdeps --module-path modulepath -m moduleName -s -R`

Viewing class-level dependencies (p. 1216)	Click here to view code image
	
	`jdeps --module-path modulepath -m moduleName -verbose`
	`jdeps --module-path modulepath -m moduleName -verbose --recursive`
	`jdeps --module-path modulepath -m moduleName -v -R`

Selected Options for the `javac` Tool

The Java language compiler, `javac`, compiles Java source code into Java bytecode. The general form of the `javac` command is:

```
javac [options] [sourcefiles]
```

Table 19.7 shows some selected options that can be used when compiling modules. The optional *sourcefiles* is an itemized list of source files, often omitted in favor of using module-re-

lated options.

Table 19.7 *Selected Options for the* `javac` *Tool*

Option	Description
`--module-source-path` *moduleSourcePath*	The *moduleSourcePath* specifies the *source directory* where the exploded modules with the *source code files* can be found.
`--module-path` *modulepath* `-p` *modulepath*	The *modulepath* specifies where the modules needed by the application can be found. This can be a root directory of the exploded modules with the *class files* or a root directory where the modular JARs can be found. Multiple directories of modules can be specified, separated by a colon (`:`) on a Unix-based platform and semicolon (`;`) on the Windows platform.
`--module` *moduleName* `-m` *moduleName*	Specifies the module(s) to be compiled. Can be a single module name or a comma-separated (`,`) list of module names. For each module specified, all modules it depends on are also compiled, according to the module dependencies.
`-d` *classesDirectory*	Specifies the *destination directory* for the class files. Mandatory when compiling modules. For classes in a package, their class files are put in a directory hierarchy that reflects the package name, with directories being created as needed. Without the `-d` option, the class files are put in the same directory as their respective source files. Specifying the directory path is platform dependent: slash (`/`) on Unix-based platforms and backslash (`\`) on Windows platforms being used when specifying the directory path.
`--add-modules` *module,...*	Specifies root modules to resolve in addition to the initial modules. These modules can be modular JAR files, JMOD files, or even exploded modules.

Selected Options for the `java` Tool

The `java` tool launches an application—that is, it creates an instance of the JVM in which to run the application. A typical command to launch a modular application is by specifying the location of its modules (*path*) and the entry point of the application (as *module* or *module/mainclass*):

Click here to view code image

```
java --module-path path --module module[/mainclass]
```

Table 19.8 shows some selected options that can be used for launching and exploring modular applications.

Table 19.8 *Selected Options for the* `java` *Tool*

Option	Description
`--module-path` *modulepath...* `-p` *modulepath*	The *modulepath* specifies the location where the modules needed to run the application can be found. This can be a root directory for the exploded modules with the *class files* or a directory where the modular JARs can be found. Multiple directories of modules can be specified, separated by a colon (`:`) on a Unix-based platform and semicolon (`;`) on Windows platforms.
`--module` *module*`[/`*mainclass*`]` `-m` *module*`[/`*mainclass*`]`	When *module/mainclass* is specified, it explicitly states the *module name* and the *fully qualified name of the class* with the `main()` method, thereby overriding any other entry point in the application. Without the *mainclass*, the entry point of the application is given by the *module* which must necessarily contain the main-class.
`--add-modules` *module,...*	Specifies root modules to resolve in addition to the initial modules.
`--list-modules`	Only lists the observable modules, and does not launch the application. That is, it lists the modules that the JVM can use when the application is run.
`--describe-module` *moduleName* `-d` *moduleName*	Describes a specified module, in particular its module descriptor, but does not launch the application.
`--validate-modules`	Validates all modules on the module path to find conflicts and errors within modules.

Selected Options for the `jar` Tool

The `jar` tool is an archiving and compression tool that can be used to bundle Java artifacts and any other resources that comprise the application. The archive file names have the `.jar` extension. A typical command to create a modular JAR (*jarfile*) with an application entry point (*qualifiedMainClassName*), based on the contents of a specific directory (*DIR*), is shown below. Note the obligatory dot (`.`) at the end of the command.

Click here to view code image

```
jar --create --file jarfile --main-class qualifiedMainClassName -C DIR .
```

Table 19.9 gives an overview of some selected options that can be used for working with JARs.

Table 19.9 *Selected Options for the* `jar` *Tool*

Option	Description
`--create` *or* `-c`	Creates a new archive.
`--extract` *or* `-x`	Extracts specified or all files in the archive.
`--list` *or* `-t`	Lists the contents of the archive.
`--update` *or* `-u`	Updates an existing archive with specified files.
`--describe-module` *or* `-d`	Prints the module descriptor of the archive and the main-class, if one is specified in the manifest.
`--verbose` *or* `-v`	Prints extra information about the operation.

`--file jarfile`
`--file=jarfile`
`-f jarfile`
`-f=jarfile`
	Specifies the name of the archive.

`-C` *DIR files*	Changes to the specified directory and includes the contents of the specified *files* from this directory. If *files* is a dot (`.`), the contents under the specified directory *DIR* are included.

Click here to view code image
`--main-class qualifiedMainClassName`
`--main-class=qualifiedMainClassName`
`-e qualifiedMainClassName`
`-e=qualifiedMainClassName`
	Specifies the entry point of the application.

`--manifest TXTFILE`
`--manifest=TXTFILE`
`-m TXTFILE`
`-m=TXTFILE`
	Reads the manifest information for the archive from the specified `TXTFILE` and incorporates it in the archive—for example, the value of the `Main-Class` attribute that specifies the entry point of the application.

Option	Description
`--module-path` *modulepath* `-p` *modulepath*	Specifies the location of the modules for recording hashes.

Selected Options for the `jdeps` Tool

The Java Class Dependency Analyzer, `jdeps`, is the tool of choice when working with modules, as it is module savvy and highly versatile. Among its extensive module analyzing capabilities, it can be used to explore dependencies at different levels: module level, package level, and class level.

Table 19.10 gives an overview of some selected options for the `jdeps` tool that can be used for analyzing modules.

Table 19.10 *Selected Options for the* `jdeps` *Tool*

Option	Description
`--module-path` *modulepath*	Specifies where to find the module JARs needed by the application. No short form, as `-p` is already reserved for `--package`.
`--module-name` *moduleName* `-m` *moduleName*	Specifies the root module for module dependency analysis.
`-summary` *or* `-s`	Presents only a summary of the module dependencies.
`--recursive` *or* `-R`	Forces `jdeps` to recursively iterate over the module dependencies. When used alone, also prints the package-level dependencies.
`-verbose` *or* `-v`	Also includes all class-level dependencies in the printout.
`-verbose:package`	Includes package-level dependencies in the printout, excluding, by default, dependencies within the same package.
`-verbose:class`	Includes class-level dependencies in the printout, excluding, by default, dependencies within the same JAR.

Selected Options for the `jlink` Tool

The `jlink` tool creates a runtime image of an application. A typical command to create a runtime image of an application requires the location of its modules (*path*), names of modules to

include (*module_names*), and output directory to store the runtime image (*output_dir*):

Click here to view code image

```
jlink --module-path path --add-modules module_names --output output_dir
```

The runtime image can be executed by the `output_dir/java` command.

Selected options for the `jlink` tool are summarized in **Table 19.11**.

Table 19.11 *Selected Options for the* `jlink` *Tool*

Option	Description
`--module-path modulepath...` `-p modulepath`	Specifies the location where the modules for the application can be found. This can be a root directory for the exploded modules with the *class files* or a directory where the modular JARs can be found. Multiple directories of modules can be specified, separated by a colon (`:`) on Unix-based platforms and semicolon (`;`) on Windows platforms.
`--add-modules module,...`	Specifies the modules to include in the generated runtime image. All application modules must be listed. Any standard or JDK modules needed will be automatically included.
`--output path`	Specifies the location of the generated runtime image.

Final Remarks on Options for the JDK Tools

It is worth taking a note of how the command options having the short form `-p`, `-m`, and `-d` are specified in different JDK tools. **Table 19.12** gives an overview of which long form they represent in which tool and how they are specified.

Table 19.12 *Selected Common Shorthand Options for JDK Tools*

`javac`	`java`	`jar`	`jdeps`
`--module-path path` `-p path`	`--module-path path` `-p path`	`--module-path path` `-p path`	`--module-pat` *(no short form reserved for* `-package` *)*
`--module module` `-m module`	`--module module[/mainclass]` `-m module[/mainclass]`	`--manifest TXTFILE` `-m TXTFILE`	`--module m` `-m module`

javac	java	jar	jdeps
(no long form) `-d` *classesDirectory*	`--describe-module` *module* `-d` *module*	`--describe-module` `-d`	*(no* `-d` *option,*

Review Questions

19.1 Given the following code:

Click here to view code image

```
module store {
  requires transitive product;
  exports store.frontend to ui;
  exports store.backend;
}
module product {
  exports product.data;
}
module ui {
  requires store;
}
module customer {
  requires ui;
  exports customer.data;
}
```

Which statement is true?

Select the one correct answer.

a. The code in module `ui` can access public types from packages `store.frontend`, `store.backend`, `customer.data`, and `product.data`.

b. The code in module `ui` can access public types from packages `store.frontend`, `store.backend`, and `product.data`.

c. The code in module `customer` can access public types from package `store.frontend`, `store.backend`, and `product.data`.

d. The code in module `customer` can access public types from package `prod-uct.data`.

e. The code in module `product` can access public types from package `customer.data`.

19.2 Which statement is true about the `requires` directive?

Select the one correct answer.

a. It allows access to any public types in the module specified in the `requires` directive.

b. It allows access to non-public types using reflection in the module specified in the `requires` directive.

c. It allows access to protected types in the module specified in the `requires` directive.

d. It allows access to all types except protected types in the module specified in the `requires` directive.

e. It allows access to public types in the exported packages of the module specified in the `requires` directive.

19.3 Place the following modules in order, based on the number of dependencies they have, starting from the module that has the most dependencies.

Select the one correct answer.

a. `java.base java.se java.logging`

b. `java.logging java.se java.base`

c. `java.base java.logging java.se`

d. `java.logging java.base java.se`

e. `java.se java.logging java.base`

f. `java.se java.base java.logging`

19.4 Given the following code:

Click here to view code image

```
module zoo {
  requires animals;
}
module animals {
  exports animals.primates;
}
```

Which type declarations from the module `animals` can be accessed by code in the module `zoo`?

Select the two correct answers.

a.

```
package animals.primates;
class Ape { }
```

b.

```
package animals.primates;
protected class Gorilla { }
```

c.

```
package animals.primates;
public interface Toolmaker { }
```

d.

```
package animals.primates.fossil;
public class JavaMan { }
```

e.

```
package animals;
public class Hybrid { }
```

f.

Click here to view code image

```
package animals.primates;
public class Chimpanzee extends Ape { }
```

19.5 The code in the `music` module needs to access types from package `production.company` contained in the `production` module.

Which module definitions implement this requirement?

Select the two correct answers.

a.

```
module music {
  requires production;
}
```

b.

```
module music {
  requires production.company;
}
```

c.

```
module music {
  requires production.company.*;
}
```

d.

```
module production {
  exports production.company;
}
```

e.

```
module production {
  exports production.*;
}
```

f.

```
module production {
  exports production.company.*;
}
```

19.6 Which of the following statements are true? Select the two correct answers.

a. An automatic module is a plain JAR loaded from the class path.

b. Plain JARs loaded from the module path are included in the unnamed module.

c. An automatic module is a plain JAR loaded from the module path.

d. Plain JARs loaded from the class path are included in the unnamed module.

19.7 Given the following code:

Click here to view code image

```
module store {
  requires product;
  exports store.frontend;
}

module marketing {
  requires store;
  opens marketing.offers;
}

module product {
  exports product.data;
  exports product.pricing to marketing;
}
```

Which statement is true?

Select the one correct answer.

a. The code in module `store` can access types defined in package `product.data`, but only by reflection in package `marketing.offers`.

b. The code in module `marketing` can access types defined in packages `product.data` and `product.pricing`.

c. The code in module `marketing` can access types defined in packages `product.data` and `store.frontend`.

d. The code in module `store` can access types defined in packages `product.data` and `product.pricing`.

e. The code in module `product` can access types defined in package `store.front-end`, and those in package `marketing.offers` by reflection.

19.8 The code contained in module `music` needs to access types from the `production.recording` package through reflection. It also needs to access public types from the packages `production.mix` and `artist.recording`.

Which module definitions implement these requirements?

Select the one correct answer.

a.

Click here to view code image

```
module music {
  requires artist.recording;
  requires production.mix;
}
module production {
  opens production.mix;
  exports production.recording;
}
module artist {
  exports artist.recording;
}
```

b.

```
module music {
  requires production;
}
module production {
  requires transitive artist;

  opens production.recording;
  exports production.mix;
}
module artist {
  exports artist.recording;
}
```

c.

Click here to view code image

```
module music {
  requires production;
}
module production {
  opens production.mix;

  exports recording;
}
module artist {
  requires transitive production;

  exports artist.recording;
}
```

d.

Click here to view code image

```
module music {
  requires production.recording;
}
module production {
```

```
  opens music;

  exports production.mix;
}
module artist {
  requires transitive production.recording;

  exports artist.recording;
}
```

19.9 Given the following code:

Click here to view code image

```
module music {
  exports music.sound;
}

module brass {
  requires music;
  provides music.sound.Instrument with brass.sound.Trumpet;
}
```

What is the correct definition for module `player` that wants to utilize the `music.sound.Instrument` service?

Select the one correct answer.

a.

Click here to view code image

```
module player {
  requires brass;
  uses music.sound.Instrument;
}
```

b.

Click here to view code image

```
module player {
  requires music;
  uses music.sound.Instrument;
}
```

c.

```
module player {
  requires brass;
  uses brass.sound.Trumpet;
}
```

d.

```
module player {
  requires music;
```

```
  uses brass.sound.Trumpet;
}
```

19.10 Which of the following statements are true about the `jlink` tool and runtime images?

Select the two correct answers.

a. The `jlink` tool can create platform-independent runtime images.

b. The `jlink` tool can create platform-specific runtime images.

c. Runtime images only contain application code.

d. Runtime images are automatically updated when a new Java version is installed.

e. The runtime image of an application can be executed on a system that has no JVM installed.

19.11 Given the following code:

Click here to view code image

```
module music {
  requires artist;
  requires instrument;
  exports preferences.style to catalog;
}

module instrument {
  requires transitive music;
  opens instrument.data;
}

module artist {
  exports preferences.style;
}

module catalog {
}
```

What are the reasons why the module declarations will fail to compile?

Select the two correct answers.

a. There is a cyclic dependency between modules.

b. There is a split package deployment.

c. There is an `opens` directive in a non-open module.

d. There is a qualified export referring to a module that does not require this export.

19.12 Which statement is true?

Select the one correct answer.

a. A modular JAR is not backward compatible and thus cannot be used in the context of a class path.

b. A plain JAR is not forward compatible and thus cannot be used in the context of a module path.

c. The `--list-modules` option of the `java` command includes the unnamed module.

d. The `--list-modules` option of the `java` command does not include automatic modules.

e. By default, an automatic module is referred to by its JAR file name.

f. By default, the unnamed module is referred to by its JAR file name.

19.13 Which of the following statements are true? Select the two correct answers.

a. An automatic module implicitly requires all explicit modules, but not other automatic modules or the unnamed module.

b. An automatic module implicitly requires all explicit modules and other automatic modules, but not the unnamed module.

c. An automatic module implicitly requires all other modules, including explicit and automatic modules, and the unnamed module.

d. An explicit module must use the `requires` directive to access types in an automatic module.

e. An explicit module must use the `requires` directive to access types in the unnamed module.