Chapter 12
Modules

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Packaging and deploying Java code and use the Java Platform Module System**
  - Define modules and their dependencies, expose module content including for reflection. Define services, producers, and consumers
  - Compile Java code, produce modular and non-modular jars, runtime images, and implement migration using unnamed and automatic modules

Packages can be grouped into modules. In this chapter, we explain the purpose of modules and how to build your own. We also show how to run them and how to discover existing modules. Next, we cover strategies for migrating an application to use modules, running a partially modularized application, and dealing with dependencies. We then move on to discuss services and service locators. Finally, we show how to create a runtime image.

We've made the code in this chapter available online. Since it can be tedious to create the directory structure, this will save you some time. Additionally, the commands need to be exactly right, so we've included those online so you can copy and paste them and compare them with what you typed. Both are available in our GitHub repo, linked to from

www.selikoff.net/ocp-17/

Introducing Modules

When writing code for the exam, you generally see small classes. After all, exam questions have to fit on a single screen! When you work on real programs, they are much bigger. A real project will consist of hundreds or thousands of classes grouped into packages. These packages are grouped into Java archive (JAR) files. A JAR is a ZIP file with some extra information, and the extension is .jar.

In addition to code written by your team, most applications also use code written by others. *Open source* is software with the code supplied and is often free to use. Java has a vibrant open source software (OSS) community, and those libraries are also supplied as JAR files. For example, there are libraries to read files, connect to a database, and much more.

Some open source projects even depend on functionality in other open source projects. For example, Spring is a commonly used framework, and JUnit is

a commonly used testing library. To use either, you need to make sure you have compatible versions of all the relevant JARs available at runtime. This complex chain of dependencies and minimum versions is often referred to by the community as *JAR hell*. Hell is an excellent way of describing the wrong version of a class being loaded or even a ClassNotFoundException at runtime.

The *Java Platform Module System* (JPMS) groups code at a higher level. The main purpose of a module is to provide groups of related packages that offer developers a particular set of functionality. It's like a JAR file, except a developer chooses which packages are accessible outside the module. Let's look at what modules are and what problems they are designed to solve.

The Java Platform Module System includes the following:

- A format for module JAR files
- Partitioning of the JDK into modules
- Additional command-line options for Java tools

Exploring a Module

In Chapter 1, "Building Blocks," we had a small Zoo application. It had only one class and just printed out one thing. Now imagine that we had a whole staff of programmers and were automating the operations of the zoo. Many things need to be coded, including the interactions with the animals, visitors, the public website, and outreach.

A *module* is a group of one or more packages plus a special file called module-info.java. The contents of this file are the *module declaration*. Figure 12.1 lists just a few of the modules a zoo might need. We decided to focus on the animal interactions in our example. The full zoo could easily have a dozen modules. In Figure 12.1, notice that there are arrows between many of the modules. These represent *dependencies*, where one module relies on code in another. The staff needs to feed the animals to keep their jobs. The line from zoo.staff to zoo.animal.feeding shows that the former depends on the latter.

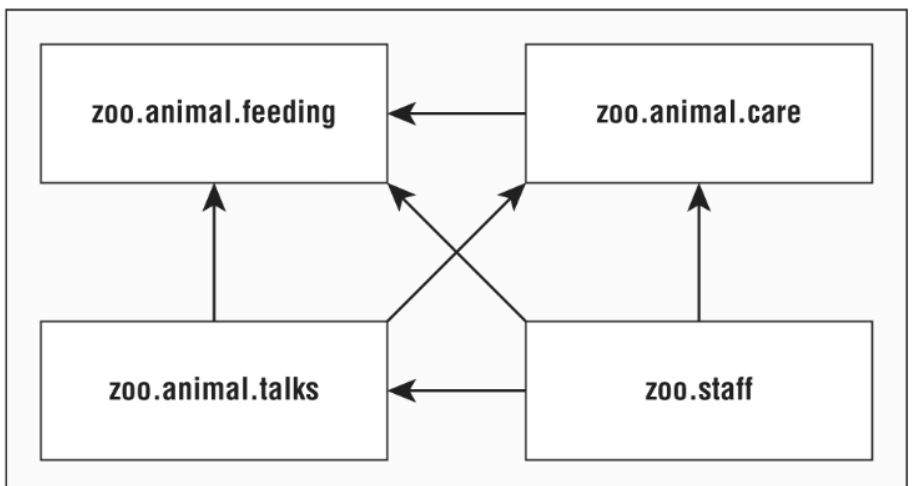


FIGURE 12.1 Design of a modular system

Now let's drill down into one of these modules. Figure 12.2 shows what is inside the zoo.animal.talks module. There are three packages with two classes each. (It's a small zoo.) There is also a strange file called module-info.java. This file is required to be inside all modules. We explain this in more detail later in the chapter.

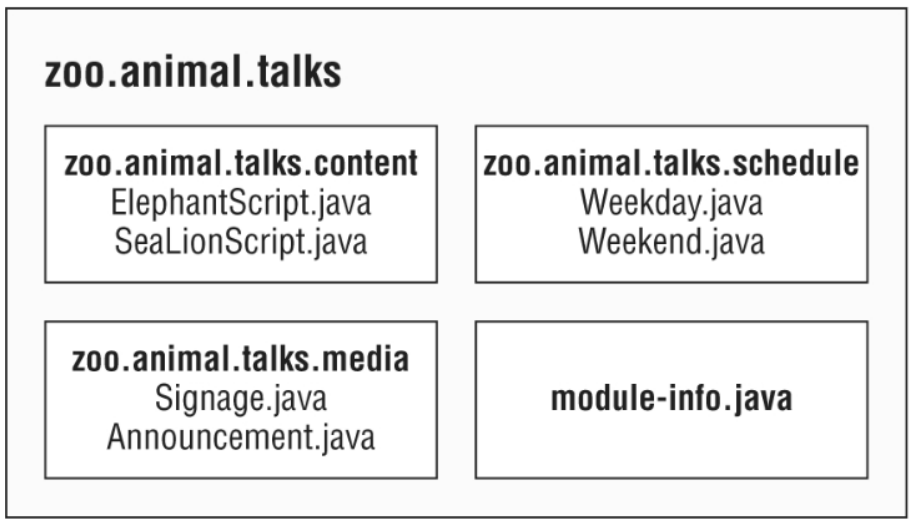


FIGURE 12.2 Looking inside a module

Benefits of Modules

Modules look like another layer of things you need to know in order to program. While using modules is optional, it is important to understand the problems they are designed to solve:

- **Better access control**: In addition to the levels of access control covered in Chapter 5, "Methods," you can have packages that are only accessible to other packages in the module.
- **Clearer dependency management:** Since modules specify what they rely on, Java can complain about a missing JAR when starting up the program rather than when it is first accessed at runtime.
- **Custom Java builds:** You can create a Java runtime that has only the parts of the JDK that your program needs rather than the full one at over 150 MB.
- **Improved security:** Since you can omit parts of the JDK from your custom build, you don't have to worry about vulnerabilities discovered in a part you don't use.
- **Improved performance**: Another benefit of a smaller Java package is improved startup time and a lower memory requirement.
- **Unique package enforcement**: Since modules specify exposed packages, Java can ensure that each package comes from only one module and avoid confusion about what is being run.

Creating and Running a Modular Program

In this section, we create, build, and run the zoo.animal.feeding module. We chose this one to start with because all the other modules depend on it. Figure 12.3 shows the design of this module. In addition to the module-info.java file, it has one package with one class inside.

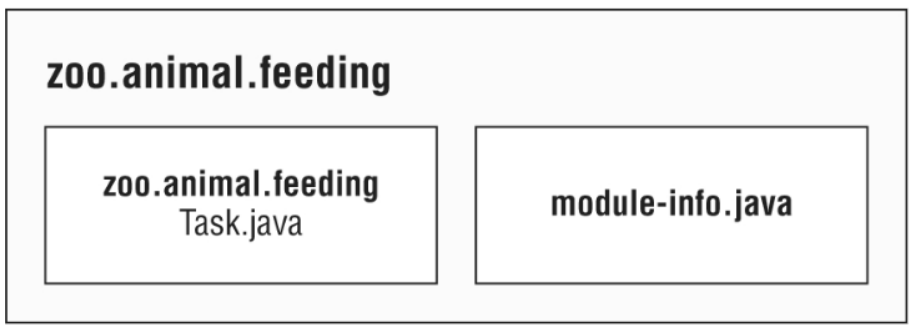


FIGURE 12.3 Contents of zoo.animal.feeding

In the next sections, we create, compile, run, and package the zoo.animal.feeding module.

Creating the Files

First we have a really simple class that prints one line in a main() method. We know, that's not much of an implementation. All those programmers we hired can fill it in with business logic. In this book, we focus on what you need to know for the exam. So, let's create a simple class.

```
package zoo.animal.feeding;

public class Task {
  public static void main(String… args) {
    System.out.println("All fed!");
  }
}
```

Next comes the module-info.java file. This is the simplest possible one:

```
module zoo.animal.feeding {
}
```

There are a few key differences between a module declaration and a regular Java class declaration:

- The module-info.java file must be in the root directory of your module. Regular Java classes should be in packages.
- The module declaration must use the keyword module instead of class, interface, or enum.

- The module name follows the naming rules for package names. It often includes periods (.) in its name. Regular class and package names are not allowed to have dashes (-). Module names follow the same rule.

That's a lot of rules for the simplest possible file. There will be many more rules when we flesh out this file later in the chapter.

The next step is to make sure the files are in the right directory structure. Figure 12.4 shows the expected directory structure.

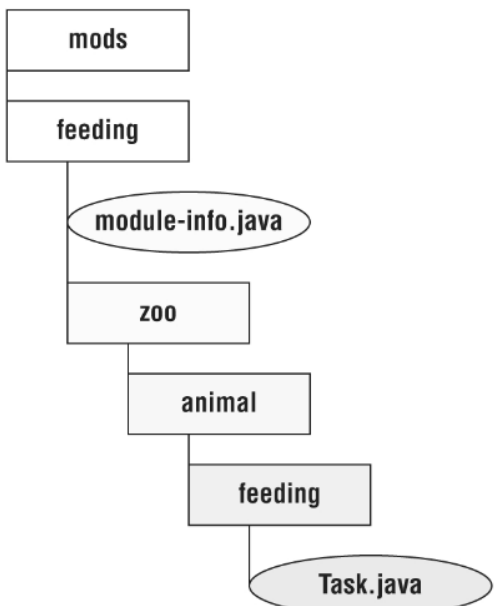


FIGURE 12.4 Module zoo.animal.feeding directory structure

In particular, feeding is the module directory, and the module-info.java file is directly under it. Just as with a regular JAR file, we also have the zoo.animal.feeding package with one subfolder per portion of the name. The Task class is in the appropriate subfolder for its package.

Also, note that we created a directory called mods at the same level as the module. We use it to store the module artifacts a little later in the chapter. This directory can be named anything, but mods is a common name. If you are following along with the online code example, note that the mods directory is not included, because it is empty.

Compiling Our First Module

Before we can run modular code, we need to compile it. Other than the module-path option, this code should look familiar from Chapter 1:

```
javac --module-path mods
  -d feeding
  feeding/zoo/animal/feeding/*.java feeding/module-info.java
```

When you're entering commands at the command line, they should be typed all on one line. We use line breaks in the book to make the commands easier to read and study. If you want to use multiple lines at the command prompt, the approach varies by operating system. Linux uses a backslash (\) to escape the line break.

As a review, the -d option specifies the directory to place the class files in. The end of the command is a list of the .java files to compile. You can list the files individually or use a wildcard for all .java files in a subdirectory.

The new part is module-path. This option indicates the location of any custom module files. In this example, module-path could have been omitted since there are no dependencies. You can think of module-path as replacing the classpath option when you are working on a modular program.

What about the *classpath?*

The classpath option has three possible forms: -cp, --class-path, and -classpath. You can still use these options. In fact, it is common to do so when writing nonmodular programs.

Just like classpath, you can use an abbreviation in the command. The syntax --module-path and -p are equivalent. That means we could have written many other commands in place of the previous command. The following four commands show the -p option:

```
javac -p mods -d feeding
  feeding/zoo/animal/feeding/*.java feeding/*.java

javac -p mods -d feeding
  feeding/zoo/animal/feeding/*.java feeding/module-info.java

javac -p mods -d feeding
  feeding/zoo/animal/feeding/Task.java feeding/module-info.java
```

```
javac -p mods -d feeding
  feeding/zoo/animal/feeding/Task.java feeding/*.java
```

While you can use whichever you like best, be sure that you can recognize all valid forms for the exam. Table 12.1 lists the options you need to know well when compiling modules. There are many more options you can pass to the javac command, but these are the ones you can expect to be tested on.

TABLE 12.1 Options you need to know for using modules with javac

Use for	Abbreviation	Long form
Directory for class files	-d <dir>	n/a
Module path	-p <path>	--module-path <path>

Real World Scenario

Building Modules

Even without modules, it is rare to run javac and java commands manually on a real project. They get long and complicated very quickly. Most developers use a build tool such as Maven or Gradle. These build tools suggest directories in which to place the class files, like target/classes.

It is likely that the only time you need to know the syntax of these commands is when you take the exam. The concepts themselves are useful, regardless.

Be sure to memorize the module command syntax. You will be tested on it on the exam. We give you lots of practice questions on the syntax to reinforce it.

Running Our First Module

Before we package our module, we should make sure it works by running it. To do that, we need to learn the full syntax. Suppose there is a module named book.module. Inside that module is a package named com.sybex, which has a class named OCP with a main() method. Figure 12.5 shows the syntax for running a module. Pay special attention to the book.module/com.sybex.OCP part. It is important to remember that you specify the module name followed by a slash (/) followed by the fully qualified class name.

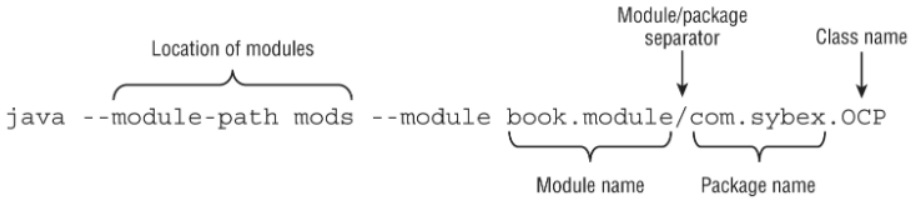


FIGURE 12.5 Running a module using java

Now that we've seen the syntax, we can write the command to run the Task class in the zoo.animal.feeding package. In the following example, the package name and module name are the same. It is common for the module name to match either the full package name or the beginning of it.

```
java --module-path feeding
  --module zoo.animal.feeding/zoo.animal.feeding.Task
```

Since you already saw that --module-path uses the short form of -p, we bet you won't be surprised to learn there is a short form of --module as well. The short option is -m. That means the following command is equivalent:

```
java -p feeding
  -m zoo.animal.feeding/zoo.animal.feeding.Task
```

In these examples, we used feeding as the module path because that's where we compiled the code. This will change once we package the module and run that.

Table 12.2 lists the options you need to know for the java command.

TABLE 12.2 Options you need to know for using modules with java

Use for	Abbreviation	Long form
Module name	-m <name>	--module <name>
Module path	-p <path>	--module-path <path>

Packaging Our First Module

A module isn't much use if we can run it only in the folder it was created in. Our next step is to package it. Be sure to create a mods directory before running this command:

```
jar -cvf mods/zoo.animal.feeding.jar -C feeding/ .
```

There's nothing module-specific here. We are packaging everything under the feeding directory and storing it in a JAR file named zoo.animal.feeding.jar under the mods folder. This represents how the module JAR will look to other code that wants to use it.

Now let's run the program again, but this time using the mods directory instead of the loose classes:

```
java -p mods
  -m zoo.animal.feeding/zoo.animal.feeding.Task
```

You might notice that this command looks identical to the one in the previous section except for the directory. In the previous example, it was feeding. In this one, it is the module path of mods. Since the module path is used, a module JAR is being run.

Updating Our Example for Multiple Modules

Now that our zoo.animal.feeding module is solid, we can start thinking about our other modules. As you can see in Figure 12.6, all three of the other modules in our system depend on the zoo.animal.feeding module.

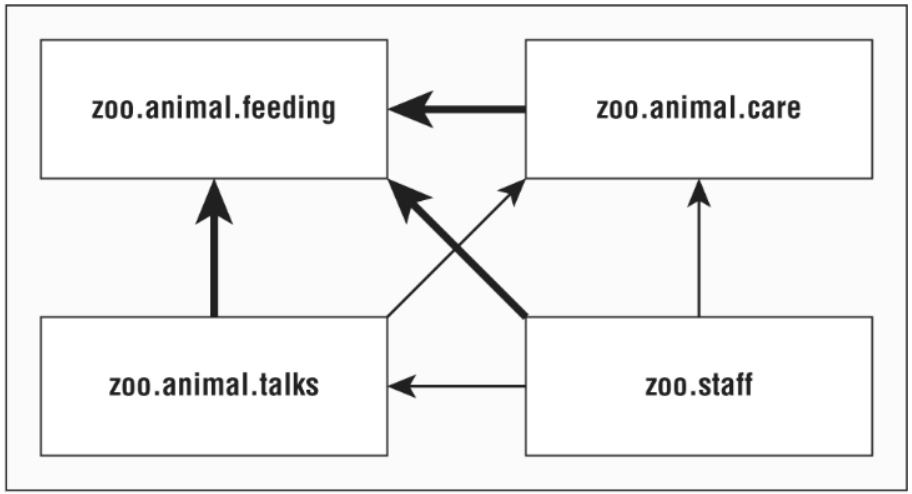


FIGURE 12.6 Modules depending on zoo.animal.feeding

Updating the Feeding Module

Since we will be having our other modules call code in the zoo.animal.feeding package, we need to declare this intent in the module declaration.

The exports directive is used to indicate that a module intends for those packages to be used by Java code outside the module. As you might expect, without an exports directive, the module is only available to be run from the command line on its own. In the following example, we export one package:

```
module zoo.animal.feeding {
  exports zoo.animal.feeding;
}
```

Recompiling and repackaging the module will update the module-info.class inside our zoo.animal.feeding.jar file. These are the same javac and jar commands you ran previously:

```
javac -p mods
  -d feeding
  feeding/zoo/animal/feeding/*.java feeding/module-info.java

jar -cvf mods/zoo.animal.feeding.jar -C feeding/ .
```

Creating a Care Module

Next, let's create the zoo.animal.care module. This time, we are going to have two packages. The zoo.animal.care.medical package will have the classes and methods that are intended for use by other modules. The zoo.animal.care.details package is only going to be used by this module. It will not be exported from the module. Think of it as healthcare privacy for the animals.

Figure 12.7 shows the contents of this module. Remember that all modules must have a module-info.java file.

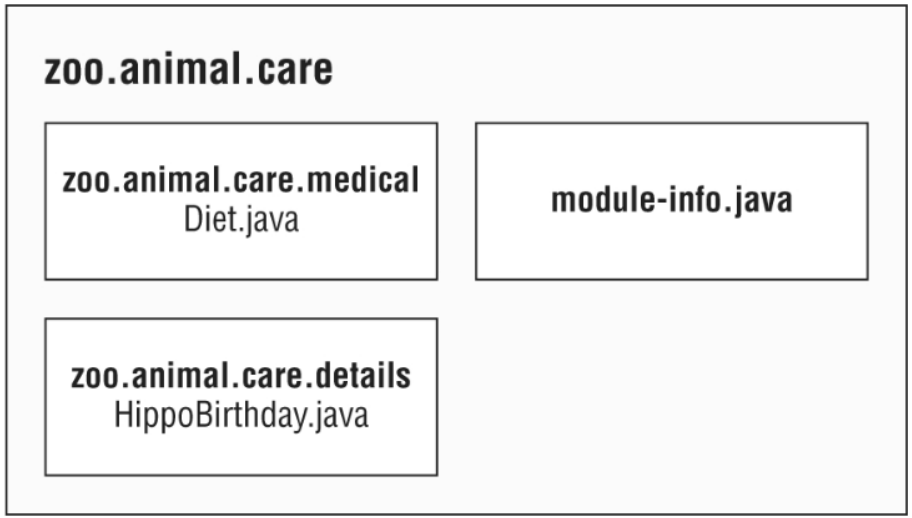


FIGURE 12.7 Contents of zoo.animal.care

The module contains two basic packages and classes in addition to the module-info.java file:

```
// HippoBirthday.java
package zoo.animal.care.details;
import zoo.animal.feeding.*;
public class HippoBirthday {
  private Task task;
```

```
}

// Diet.java
package zoo.animal.care.medical;
public class Diet { }
```

This time the module-info.java file specifies three things:

```
1: module zoo.animal.care {
2:   exports zoo.animal.care.medical;
3:   requires zoo.animal.feeding;
4: }
```

Line 1 specifies the name of the module. Line 2 lists the package we are exporting so it can be used by other modules. So far, this is similar to the zoo.animal.feeding module.

On line 3, we see a new directive. The requires statement specifies that a module is needed. The zoo.animal.care module depends on the zoo.animal.feeding module.

Next, we need to figure out the directory structure. We will create two packages. The first is zoo.animal.care.details and contains one class named HippoBirthday. The second is zoo.animal.care.medical, which contains one class named Diet. Try to draw the directory structure on paper or create it on your computer. If you are trying to run these examples without using the online code, just create classes without variables or methods for everything except the module-info.java files.

You might have noticed that the packages begin with the same prefix as the module name. This is intentional. You can think of it as if the module name "claims" the matching package and all subpackages.

To review, we now compile and package the module:

```
javac -p mods
  -d care
  care/zoo/animal/care/details/*.java
  care/zoo/animal/care/medical/*.java
  care/module-info.java
```

We compile both packages and the module-info.java file. In the real world, you'll use a build tool rather than doing this by hand. For the exam, you just list all the packages and/or files you want to compile.

Now that we have compiled code, it's time to create the module JAR:

```
jar -cvf mods/zoo.animal.care.jar -C care/ .
```

Creating the Talks Module

So far, we've used only one exports and requires statement in a module. Now you'll learn how to handle exporting multiple packages or requiring multiple modules. In Figure 12.8, observe that the zoo.animal.talks module depends on two modules: zoo.animal.feeding and zoo.animal.care. This means that there must be two requires statements in the module-info.java file.

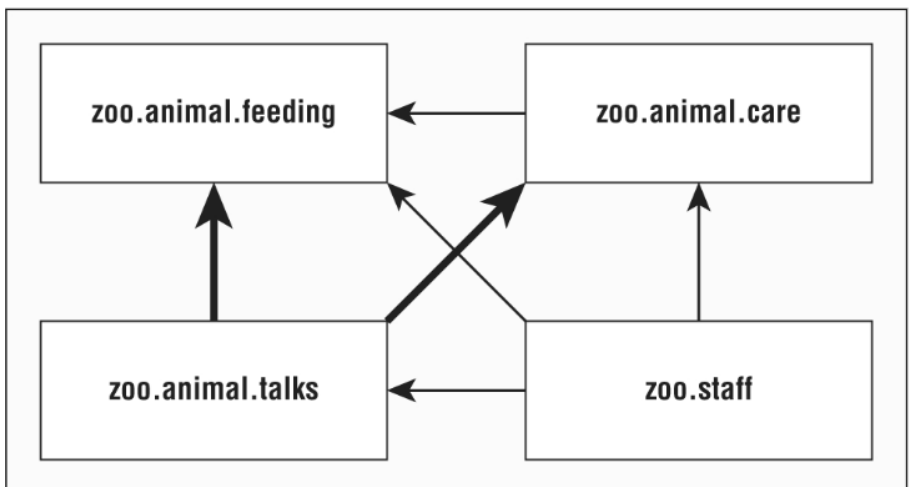


FIGURE 12.8 Dependencies for zoo.animal.talks

Figure 12.9 shows the contents of this module. We are going to export all three packages in this module.

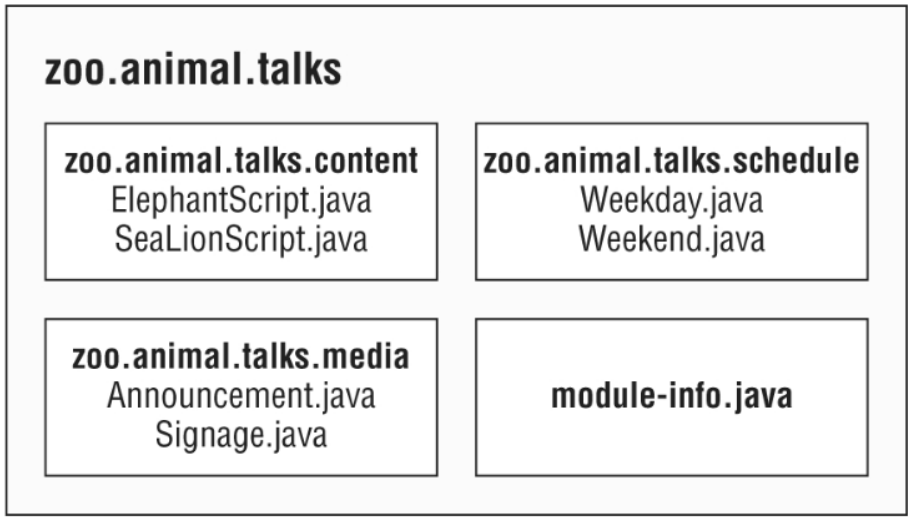


FIGURE 12.9 Contents of zoo.animal.talks

First let's look at the module-info.java file for zoo.animal.talks:

```
1: module zoo.animal.talks {
2:   exports zoo.animal.talks.content;
3:   exports zoo.animal.talks.media;
4:   exports zoo.animal.talks.schedule;
```

```
5:
6:   requires zoo.animal.feeding;
7:   requires zoo.animal.care;
8: }
```

Line 1 shows the module name. Lines 2–4 allow other modules to reference all three packages. Lines 6 and 7 specify the two modules that this module depends on.

Then we have the six classes, as shown here:

```
// ElephantScript.java
package zoo.animal.talks.content;
public class ElephantScript { }

// SeaLionScript.java
package zoo.animal.talks.content;
public class SeaLionScript { }

// Announcement.java
package zoo.animal.talks.media;
public class Announcement {
  public static void main(String[] args) {
    System.out.println("We will be having talks");
  }
}

// Signage.java
package zoo.animal.talks.media;
public class Signage { }

// Weekday.java
package zoo.animal.talks.schedule;
public class Weekday { }

// Weekend.java
package zoo.animal.talks.schedule;
public class Weekend {}
```

If you are still following along on your computer, create these classes in the packages. The following are the commands to compile and build the module:

```
javac -p mods
  -d talks
  talks/zoo/animal/talks/content/*.java talks/zoo/animal/talks/media/*.java
  talks/zoo/animal/talks/schedule/*.java talks/module-info.java
```

jar -cvf mods/zoo.animal.talks.jar -C talks/ .

Creating the Staff Module

Our final module is zoo.staff. Figure 12.10 shows that there is only one package inside. We will not be exposing this package outside the module.

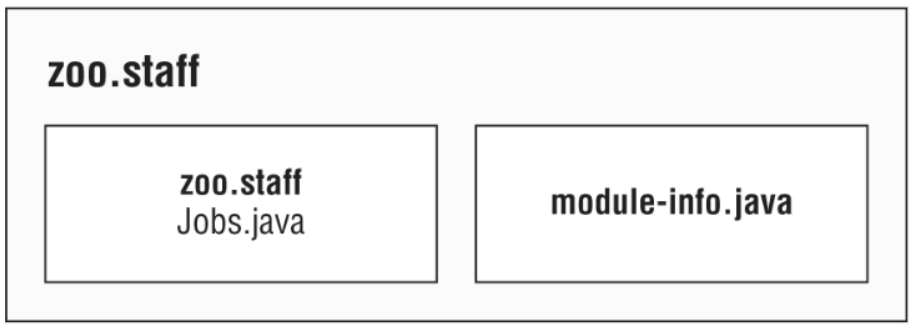


FIGURE 12.10 Contents of zoo.staff

Based on Figure 12.11, do you know what should go in the module-info?

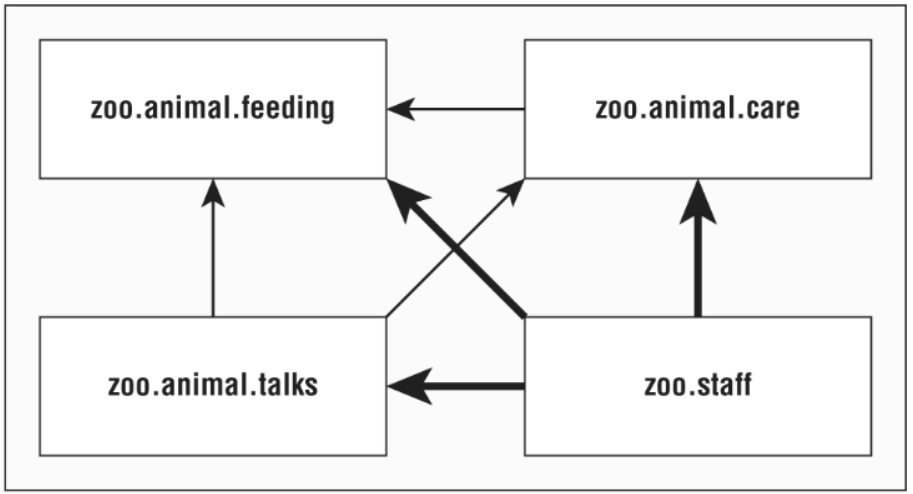


FIGURE 12.11 Dependencies for zoo.staff

There are three arrows in Figure 12.11 pointing from zoo.staff to other modules. These represent the three modules that are required. Since no packages are to be exposed from zoo.staff, there are no exports statements. This gives us:

```
module zoo.staff {
  requires zoo.animal.feeding;
  requires zoo.animal.care;
```

```
  requires zoo.animal.talks;
}
```

In this module, we have a single class in the Jobs.java file:

```
package zoo.staff;
public class Jobs { }
```

For those of you following along on your computer, create a class in the package. The following are the commands to compile and build the module:

```
javac -p mods
  -d staff
  staff/zoo/staff/*.java staff/module-info.java

jar -cvf mods/zoo.staff.jar -C staff/ .
```

Diving into the Module Declaration

Now that we've successfully created modules, we can learn more about the module declaration. In these sections, we look at exports, requires, and opens. In the following section on services, we explore provides and uses. Now would be a good time to mention that these directives can appear in any order in the module declaration.

Exporting a Package

We've already seen how exports *packageName* exports a package to other modules. It's also possible to export a package to a specific module. Suppose the zoo decides that only staff members should have access to the talks. We could update the module declaration as follows:

```
module zoo.animal.talks {
  exports zoo.animal.talks.content to zoo.staff;
  exports zoo.animal.talks.media;
  exports zoo.animal.talks.schedule;

  requires zoo.animal.feeding;
  requires zoo.animal.care;
}
```

From the zoo.staff module, nothing has changed. However, no other modules would be allowed to access that package.

You might have noticed that none of our other modules requires zoo.animal.talks in the first place. However, we don't know what other modules will exist in the future. It is important to consider future use when designing modules. Since we want only the one module to have access, we only allow access for that module.

Exported Types

We've been talking about exporting a package. But what does that mean, exactly? All public classes, interfaces, enums, and records are exported. Further, any public and protected fields and methods in those files are visible.

Fields and methods that are private are not visible because they are not accessible outside the class. Similarly, package fields and methods are not visible because they are not accessible outside the package.

The exports directive essentially gives us more levels of access control. Table 12.3 lists the full access control options.

TABLE 12.3 Access control with modules

Level	Within module code	Outside module
private	Available only within class	No access
Package	Available only within package	No access
protected	Available only within package or to subclasses	Accessible to subclasses only if package is exported
public	Available to all classes	Accessible only if package is exported

Requiring a Module Transitively

As you saw earlier in this chapter, requires *moduleName* specifies that the current module depends on moduleName. There's also a requires transitive *moduleName*, which means that any module that requires this module will also depend on moduleName.

Well, that was a mouthful. Let's look at an example. Figure 12.12 shows the modules with dashed lines for the redundant relationships and solid lines for relationships specified in the module-info. This shows how the module relationships would look if we were to only use transitive dependencies.

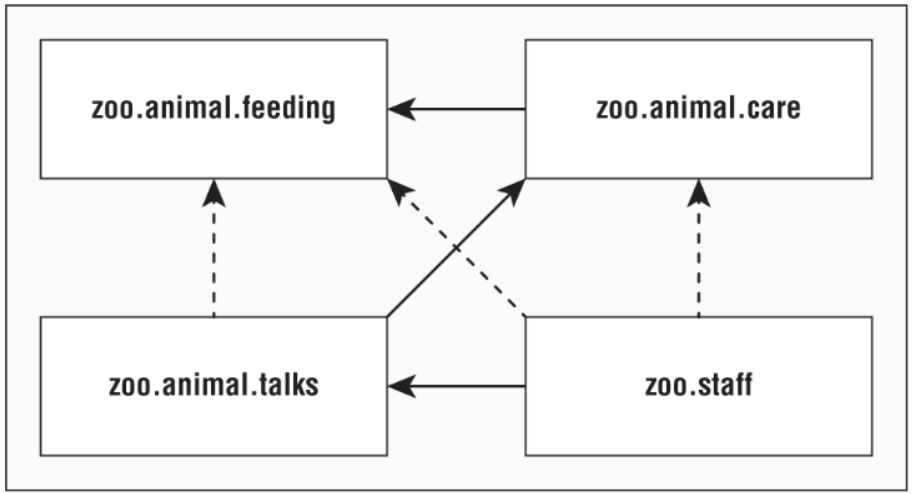


FIGURE 12.12 Transitive dependency version of our modules

For example, zoo.animal.talks depends on zoo.animal.care, which depends on zoo.animal.feeding. That means the arrow between zoo.animal.talks and zoo.animal.feeding no longer appears in Figure 12.12.

Now let's look at the four module declarations. The first module remains unchanged. We are exporting one package to any packages that use the module.

```
module zoo.animal.feeding {
  exports zoo.animal.feeding;
}
```

The zoo.animal.care module is the first opportunity to improve things. Rather than forcing all remaining modules to explicitly specify zoo.animal.feeding, the code uses requires transitive.

```
module zoo.animal.care {
  exports zoo.animal.care.medical;
  requires transitive zoo.animal.feeding;
}
```

In the zoo.animal.talks module, we make a similar change and don't force other modules to specify zoo.animal.care. We also no longer need to specify zoo.animal.feeding, so that line is commented out.

```
module zoo.animal.talks {
  exports zoo.animal.talks.content to zoo.staff;
  exports zoo.animal.talks.media;
  exports zoo.animal.talks.schedule;
  // no longer needed requires zoo.animal.feeding;
  // no longer needed requires zoo.animal.care;
```

```
  requires transitive zoo.animal.care;
}
```

Finally, in the zoo.staff module, we can get rid of two requires statements.

```
module zoo.staff {
  // no longer needed requires zoo.animal.feeding;
  // no longer needed requires zoo.animal.care;
  requires zoo.animal.talks;
}
```

The more modules you have, the greater the benefits of the requires transitive compound. It is also more convenient for the caller. If you were trying to work with this zoo, you could just require zoo.staff and have the remaining dependencies automatically inferred.

Effects of *requires transitive*

Given our new module declarations, and using Figure 12.12, what is the effect of applying the transitive modifier to the requires statement in our zoo.animal.care module? Applying the transitive modifiers has the following effects:

- Module zoo.animal.talks can optionally declare that it requires the zoo.animal.feeding module, but it is not required.
- Module zoo.animal.care cannot be compiled or executed without access to the zoo.animal.feeding module.
- Module zoo.animal.talks cannot be compiled or executed without access to the zoo.animal.feeding module.

These rules hold even if the zoo.animal.care and zoo.animal.talks modules do not explicitly reference any packages in the zoo.animal.feeding module. On the other hand, without the transitive modifier in our module declaration of zoo.animal.care, the other modules would have to explicitly use requires in order to reference any packages in the zoo.animal.feeding module.

Duplicate *requires* Statements

One place the exam might try to trick you is mixing requires and requires transitive. Can you think of a reason this code doesn't compile?

```
module bad.module {
  requires zoo.animal.talks;
  requires transitive zoo.animal.talks;
}
```

Java doesn't allow you to repeat the same module in a requires clause. It is redundant and most likely an error in coding. Keep in mind that requires transitive is like requires plus some extra behavior.

Opening a Package

Java allows callers to inspect and call code at runtime with a technique called *reflection*. This is a powerful approach that allows calling code that might not be available at compile time. It can even be used to subvert access control! Don't worry—you don't need to know how to write code using reflection for the exam.

The opens directive is used to enable reflection of a package within a module. You only need to be aware that the opens directive exists rather than understanding it in detail for the exam.

Since reflection can be dangerous, the module system requires developers to explicitly allow reflection in the module declaration if they want calling modules to be allowed to use it. The following shows how to enable reflection for two packages in the zoo.animal.talks module:

```
module zoo.animal.talks {
  opens zoo.animal.talks.schedule;
  opens zoo.animal.talks.media to zoo.staff;
}
```

The first example allows any module using this one to use reflection. The second example only gives that privilege to the zoo.staff module. There are two more directives you need to know for the exam—provides and uses—which are covered in the following section.

Real World Scenaria

Opening an Entire Module

In the previous example, we opened two packages in the zoo.animal.talks module, but suppose we instead wanted to open all packages for reflection. No problem. We can use the open module modifier, rather than the opens directive (notice the s difference):

```
  open module zoo.animal.talks {
}
```

With this module modifier, Java knows we want all the packages in the module to be open. What happens if you apply both together?

```
  open module zoo.animal.talks {
  opens zoo.animal.talks.schedule; // DOES NOT COMPILE
}
```

This does not compile because a modifier that uses the open modifier is not permitted to use the opens directive. After all, the packages are already open!

Creating a Service

In this section, you learn how to create a service. A *service* is composed of an interface, any classes the interface references, and a way of looking up implementations of the interface. The implementations are not part of the service.

We will be using a tour application in the services section. It has four modules shown in Figure 12.13. In this example, the zoo.tours.api and zoo.tours.reservations modules make up the service since they consist of the interface and lookup functionality.

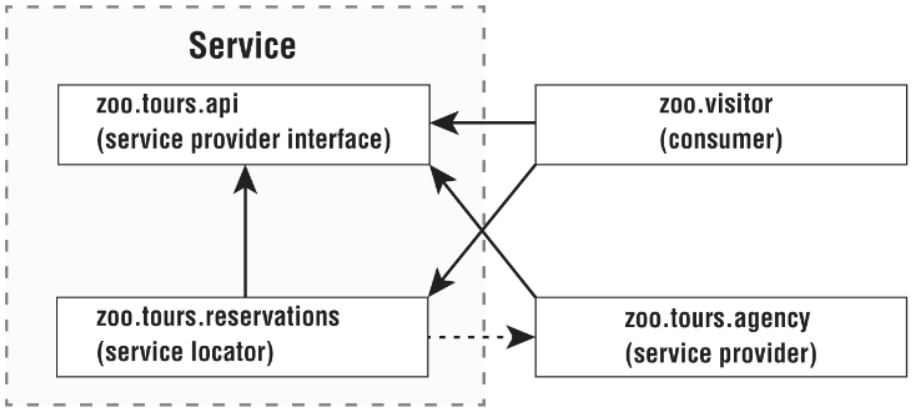


FIGURE 12.13 Modules in the tour application

You aren't required to have four separate modules. We do so to illustrate the concepts. For example, the service provider interface and service locator could be in the same module.

Declaring the Service Provider Interface

First, the zoo.tours.api module defines a Java object called Souvenir. It is considered part of the service because it will be referenced by the interface.

```
// Souvenir.java
package zoo.tours.api;

public record Souvenir(String description) { }
```

Next, the module contains a Java interface type. This interface is called the *service provider interface* because it specifies what behavior our service will have. In this case, it is a simple API with three methods.

```
// Tour.java
package zoo.tours.api;

public interface Tour {
  String name();
  int length();
  Souvenir getSouvenir();
}
```

All three methods use the implicit public modifier. Since we are working with modules, we also need to create a module-info.java file so our module definition exports the package containing the interface.

```
// module-info.java
module zoo.tours.api {
```

```
  exports zoo.tours.api;
}
```

Now that we have both files, we can compile and package this module.

```
javac -d serviceProviderInterfaceModule
  serviceProviderInterfaceModule/zoo/tours/api/*.java
  serviceProviderInterfaceModule/module-info.java

jar -cvf mods/zoo.tours.api.jar -C serviceProviderInterfaceModule/ .
```

A service provider "interface" can be an abstract class rather than an actual interface. Since you will only see it as an interface on the exam, we use that term in the book.

To review, the service includes the service provider interface and supporting classes it references. The service also includes the lookup functionality, which we define next.

Creating a Service Locator

To complete our service, we need a service locator. A *service locator* can find any classes that implement a service provider interface.

Luckily, Java provides a ServiceLoader class to help with this task. You pass the service provider interface type to its load() method, and Java will return any implementation services it can find. The following class shows it in action:

```
// TourFinder.java
package zoo.tours.reservations;

import java.util.*;
import zoo.tours.api.*;

public class TourFinder {

  public static Tour findSingleTour() {
    ServiceLoader<Tour> loader = ServiceLoader.load(Tour.class);
    for (Tour tour : loader)
      return tour;
    return null;
```

```
  }
  public static List<Tour> findAllTours() {
    List<Tour> tours = new ArrayList<>();
    ServiceLoader<Tour> loader = ServiceLoader.load(Tour.class);
    for (Tour tour : loader)
      tours.add(tour);
    return tours;
  }
}
```

As you can see, we provided two lookup methods. The first is a convenience method if you are expecting exactly one Tour to be returned. The other returns a List, which accommodates any number of service providers. At runtime, there may be many service providers (or none) that are found by the service locator.

The ServiceLoader call is relatively expensive. If you are writing a real application, it is best to cache the result.

Our module definition exports the package with the lookup class TourFinder. It requires the service provider interface package. It also has the uses directive since it will be looking up a service.

```
// module-info.java
module zoo.tours.reservations {
  exports zoo.tours.reservations;
  requires zoo.tours.api;
  uses zoo.tours.api.Tour;
}
```

Remember that both requires and uses are needed, one for compilation and one for lookup. Finally, we compile and package the module.

```
javac -p mods -d serviceLocatorModule
  serviceLocatorModule/zoo/tours/reservations/*.java
  serviceLocatorModule/module-info.java

jar -cvf mods/zoo.tours.reservations.jar -C serviceLocatorModule/ .
```

Now that we have the interface and lookup logic, we have completed our service.

Using *ServiceLoader*

There are two methods in ServiceLoader that you need to know for the exam. The declaration is as follows, sans the full implementation:

```
public final class ServiceLoader<S> implements Iterable<S> {

public static <S> ServiceLoader<S> load(Class<S> service) { ... }

public Stream<Provider<S>> stream() { ... }

// Additional methods
}
```

As we already saw, calling ServiceLoader.load() returns an object that you can loop through normally. However, requesting a Stream gives you a different type. The reason for this is that a Stream controls when elements are evaluated. Therefore, a ServiceLoader returns a Stream of Provider objects. You have to call get() to retrieve the value you wanted out of each Provider, such as in this example:

```
  ServiceLoader.load(Tour.class)
.stream()
.map(Provider::get)
.mapToInt(Tour::length)
.max()
.ifPresent(System.out::println);
```

Invoking from a Consumer

Next up is to call the service locator by a consumer. A *consumer* (or *client*) refers to a module that obtains and uses a service. Once the consumer has acquired a service via the service locator, it is able to invoke the methods provided by the service provider interface.

```
// Tourist.java
package zoo.visitor;

import java.util.*;
import zoo.tours.api.*;
import zoo.tours.reservations.*;

public class Tourist {
```

```
  public static void main(String[] args) {
    Tour tour = TourFinder.findSingleTour();
    System.out.println("Single tour: " + tour);

    List<Tour> tours = TourFinder.findAllTours();
    System.out.println("# tours: " + tours.size());
  }
}
```

Our module definition doesn't need to know anything about the implementations since the zoo.tours.reservations module is handling the lookup.

```
// module-info.java
module zoo.visitor {
  requires zoo.tours.api;
  requires zoo.tours.reservations;
}
```

This time, we get to run a program after compiling and packaging.

```
javac -p mods -d consumerModule
  consumerModule/zoo/visitor/*.java consumerModule/module-info.java

jar -cvf mods/zoo.visitor.jar -C consumerModule/ .

java -p mods -m zoo.visitor/zoo.visitor.Tourist
```

The program outputs the following:

```
Single tour: null
# tours: 0
```

Well, that makes sense. We haven't written a class that implements the interface yet.

Adding a Service Provider

A *service provider* is the implementation of a service provider interface. As we said earlier, at runtime it is possible to have multiple implementation classes or modules. We will stick to one here for simplicity.

Our service provider is the zoo.tours.agency package because we've outsourced the running of tours to a third party.

```
// TourImpl.java
package zoo.tours.agency;

import zoo.tours.api.*;
```

```
public class TourImpl implements Tour {
  public String name() {
    return "Behind the Scenes";
  }
  public int length() {
    return 120;
  }
  public Souvenir getSouvenir() {
    return new Souvenir("stuffed animal");
  }
}
```

Again, we need a module-info.java file to create a module.

```
// module-info.java
module zoo.tours.agency {
  requires zoo.tours.api;
  provides zoo.tours.api.Tour with zoo.tours.agency.TourImpl;
}
```

The module declaration requires the module containing the interface as a dependency. We don't export the package that implements the interface since we don't want callers referring to it directly. Instead, we use the provides directive. This allows us to specify that we provide an implementation of the interface with a specific implementation class. The syntax looks like this:

provides *interfaceName* with *className*;

We have not exported the package containing the implementation. Instead, we have made the implementation available to a service provider using the interface.

Finally, we compile it and package it up.

```
javac -p mods -d serviceProviderModule
  serviceProviderModule/zoo/tours/agency/*.java
  serviceProviderModule/module-info.java
jar -cvf mods/zoo.tours.agency.jar -C serviceProviderModule/ .
```

Now comes the cool part. We can run the Java program again.

```
java -p mods -m zoo.visitor/zoo.visitor.Tourist
```

This time, we see the following output:

```
Single tour: zoo.tours.agency.TourImpl@1936f0f5
# tours: 1
```

Notice how we didn't recompile the zoo.tours.reservations or zoo.visitor package. The service locator was able to observe that there was now a service provider implementation available and find it for us.

This is useful when you have functionality that changes independently of the rest of the code base. For example, you might have custom reports or logging.

In software development, the concept of separating different components into stand-alone pieces is referred to as *loose coupling*. One advantage of loosely coupled code is that it can be easily swapped out or replaced with minimal (or zero) changes to code that uses it. Relying on a loosely coupled structure allows service modules to be easily extensible at runtime.

Reviewing Directives and Services

Table 12.4 summarizes what we've covered in the section about services. We recommend learning really well what is needed when each artifact is in a separate module. That is most likely what you will see on the exam and will ensure that you understand the concepts. Table 12.5 lists all the directives you need to know for the exam.

TABLE 12.4 Reviewing services

Artifact	Part of the service	Directives required
Service provider interface	Yes	exports
Service provider	No	requires provides
Service locator	Yes	exports requires uses
Consumer	No	requires

TABLE 12.5 Reviewing directives

Directive	Description
exports *package*; **exports** *package***to** *module*;	Makes package available outside module
requires *module*; **requires transitive** *module*;	Specifies another module as dependency
opens *package*; **opens** *package***to** *module*;	Allows package to be used with reflection
provides *serviceInterface***with** *implName*;	Makes service available
uses *serviceInterface*;	References service

Discovering Modules

So far, we've been working with modules that we wrote. Even the classes built into the JDK are modularized. In this section, we show you how to use commands to learn about modules.

You do not need to know the output of the commands in this section. You do, however, need to know the syntax of the commands and what they do. We include the output where it facilitates remembering what is going on. But you don't need to memorize that (which frees up more space in your head to memorize command-line options).

Identifying Built-in Modules

The most important module to know is java.base. It contains most of the packages you have been learning about for the exam. In fact, it is so important that you don't even have to use the requires directive; it is available to all modular applications. Your module-info.java file will still compile if you explicitly require java.base. However, it is redundant, so it's better to omit it. Table 12.6 lists some common modules and what they contain.

TABLE 12.6 Common modules

Module name	What it contains	Coverage in book
java.base	Collections, math, IO, NIO.2, concurrency, etc.	Most of this book
java.desktop	Abstract Windows Toolkit (AWT) and Swing	Not on exam beyond module name
java.logging	Logging	Not on exam beyond module name
java.sql	JDBC	Chapter 15, “JDBC”
java.xml	Extensible Markup Language (XML)	Not on exam beyond module name

The exam creators feel it is important to recognize the names of modules supplied by the JDK. While you don't need to know the names by heart, you do need to be able to pick them out of a lineup.

For the exam, you need to know that module names begin with java for APIs you are likely to use and with jdk for APIs that are specific to the JDK. Table 12.7 lists all the modules that begin with java.

TABLE 12.7 Java modules prefixed with java

java.base	java.naming	java.smartcardio
java.compiler	java.net .http	java.sql
java.datatransfer	java.prefs	java.sql.rowset
java.desktop	java.rmi	java.transaction.xa
java.instrument	java.scripting	java.xml
java.logging	java.se	java.xml.crypto
java.management	java.security.jgss	
java.management.rmi	java.security.sasl	

Table 12.8 lists all the modules that begin with jdk. We recommend reviewing this right before the exam to increase the chances of them sounding familiar. Remember that you don't have to memorize them.

TABLE 12.8 Java modules prefixed with jdk

jdk.accessiblity	jdk.javadoc	jdk.management.agent
jdk.attach	jdk.jcmd	jdk.management.jfr
jdk.charsets	jdk.jconsole	jdk.naming.dns
jdk.compiler	jdk.jdeps	jdk.naming.rmi
jdk.crypto.cryptoki	jdk.jdi	jdk.net
jdk.crypto.ec	jdk.jdwp.agent	jdk.nio.mapmode
jdk.dynalink	jdk.jfr	jdk.sctp
jdk.editpad	jdk.jlink	jdk.security.auth
jdk.hotspot.agent	jdk.jshell	jdk.security.jgss
jdk.httpserver	jdk.jsobject	jdk.xml.dom
jdk.incubator.foreign	jdk.jstatd	jdk.zipfs
jdk.incubator.vector	jdk.localedata	
jdk.jartool	jdk.management	

Getting Details with *java*

The java command has three module-related options. One describes a

module, another lists the available modules, and the third shows the module resolution logic.

It is also possible to add modules, exports, and more at the command line. But please don't. It's confusing and hard to maintain. Note that these flags are available on java but not all commands.

Describing a Module

Suppose you are given the zoo.animal.feeding module JAR file and want to know about its module structure. You could "unjar" it and open the module-info.java file. This would show you that the module exports one package and doesn't explicitly require any modules.

```
module zoo.animal.feeding {
  exports zoo.animal.feeding;
}
```

However, there is an easier way. The java command has an option to describe a module. The following two commands are equivalent:

```
java -p mods
  -d zoo.animal.feeding

java -p mods
  --describe-module zoo.animal.feeding
```

Each prints information about the module. For example, it might print this:

```
zoo.animal.feeding file:///absolutePath/mods/zoo.animal.feeding.jar
exports zoo.animal.feeding
requires java.base mandated
```

The first line is the module we asked about: zoo.animal.feeding. The second line starts with information about the module. In our case, it is the same package exports statement we had in the module declaration file.

On the third line, we see requires java.base mandated. Now, wait a minute. The module declaration very clearly does not specify any modules that zoo.animal.feeding has as dependencies.

Remember, the java.base module is special. It is automatically added as a dependency to all modules. This module has frequently used packages like java.util. That's what the mandated is about. You get java.base regardless of whether you asked for it.

In classes, the java.lang package is automatically imported whether you type it or not. The java.base module works the same way. It is automatically available to all other modules.

More about Describing Modules

You only need to know how to run --describe-module for the exam rather than interpret the output. However, you might encounter some surprises when experimenting with this feature, so we describe them in a bit more detail here.

Assume the following are the contents of module-info.java in zoo.animal.care:

```
module zoo.animal.care {
exports zoo.animal.care.medical to zoo.staff;
requires transitive zoo.animal.feeding;
}
```

Now we have the command to describe the module and the output.

```
java -p mods -d zoo.animal.care
```

```
zoo.animal.care file:///absolutePath/mods/zoo.animal.care.jar
requires zoo.animal.feeding transitive
requires java.base mandated
qualified exports zoo.animal.care.medical to zoo.staff
contains zoo.animal.care.details
```

The first line of the output is the absolute path of the module file. The two requires lines should look familiar as well. The first is in the module-info, and the other is added to all modules. Next comes something new. The qualified exports is the full name of the package we are exporting to a specific module.

Finally, the contains means that there is a package in the module that is not exported at all. This is true. Our module has two packages, and one is available only to code inside the module.

Listing Available Modules

In addition to describing modules, you can use the java command to list the modules that are available. The simplest form lists the modules that are part of the JDK.

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

When we ran it, the output went on for 70 lines and looked like this:

```
java.base@17
java.compiler@17
java.datatransfer@17
```

This is a listing of all the modules that come with Java and their version numbers. You can tell that we were using Java 17 when testing this example.

More interestingly, you can use this command with custom code. Let's try again with the directory containing our zoo modules.

```
java -p mods --list-modules
```

How many lines do you expect to be in the output this time? There are 78 lines now: the 70 built-in modules plus the 8 we've created in this chapter. Two of the custom lines look like this:

```
zoo.animal.care file:///absolutePath/mods/zoo.animal.care.jar
zoo.animal.feeding file:///absolutePath/mods/zoo.animal.feeding.jar
```

Since these are custom modules, we get a location on the file system. If the project had a module version number, it would have both the version number and the file system path.

Note that --list-modules exits as soon as it prints the observable modules. It does not run the program.

Showing Module Resolution

If listing the modules doesn't give you enough output, you can also use the --show-module-resolution option. You can think of it as a way of debugging modules. It spits out a lot of output when the program starts up. Then it runs the program.

```
java --show-module-resolution
  -p feeding
  -m zoo.animal.feeding/zoo.animal.feeding.Task
```

Luckily, you don't need to understand this output. That said, having seen it will make it easier to remember. Here's a snippet of the output:

```
root zoo.animal.feeding file:///absolutePath/feeding/
java.base binds java.desktop jrt:/java.desktop
java.base binds jdk.jartool jrt:/jdk.jartool
```

…
jdk.security.auth requires java.naming jrt:/java.naming
jdk.security.auth requires java.security.jgss jrt:/java.security.jgss
…
All fed!

It starts by listing the root module. That's the one we are running: zoo.animal.feeding. Then it lists many lines of packages included by the mandatory java.base module. After a while, it lists modules that have dependencies. Finally, it outputs the result of the program: All fed!.

Describing with *jar*

Like the java command, the jar command can describe a module. These commands are equivalent:

jar -f mods/zoo.animal.feeding.jar -d
jar --file mods/zoo.animal.feeding.jar --describe-module

The output is slightly different from when we used the java command to describe the module. With jar, it outputs the following:

zoo.animal.feeding jar:file:///*absolutePath*/mods/zoo.animal.feeding.jar
/!module-info.class
exports zoo.animal.feeding
requires java.base mandated

The JAR version includes the module-info.class in the filename, which is not a particularly significant difference in the scheme of things. You don't need to know this difference. You do need to know that both commands can describe a module.

Learning about Dependencies with *jdeps*

The jdeps command gives you information about dependencies within a module. Unlike describing a module, it looks at the code in addition to the module declaration. This tells you what dependencies are actually used rather than simply declared. Luckily, you are not expected to memorize all the options for the exam.

You are expected to understand how to use jdeps with projects that have not yet been modularized to assist in identifying dependencies and problems. First, we will create a JAR file from this class. If you are following along, feel free to copy the class from the online examples referenced at the beginning of the chapter rather than typing it in.

// Animatronic.java
package zoo.dinos;

```
import java.time.*;
import java.util.*;
import sun.misc.Unsafe;

public class Animatronic {
  private List<String> names;
  private LocalDate visitDate;

  public Animatronic(List<String> names, LocalDate visitDate) {
    this.names = names;
    this.visitDate = visitDate;
  }
  public void unsafeMethod() {
    Unsafe unsafe = Unsafe.getUnsafe();
  }
}
```

This example is silly. It uses a number of unrelated classes. The Bronx Zoo really did have electronic moving dinosaurs for a while, so at least the idea of having dinosaurs in a zoo isn't beyond the realm of possibility.

Now we can compile this file. You might have noticed that there is no module-info.java file. That is because we aren't creating a module. We are looking into what dependencies we will need when we do modularize this JAR.

```
javac zoo/dinos/*.java
```

Compiling works, but it gives you some warnings about Unsafe being an internal API. Don't worry about those for now—we discuss that shortly. (Maybe the dinosaurs went extinct because they did something unsafe.)

Next, we create a JAR file.

```
jar -cvf zoo.dino.jar .
```

We can run the jdeps command against this JAR to learn about its dependencies. First, let's run the command without any options. On the first two lines, the command prints the modules that we would need to add with a requires directive to migrate to the module system. It also prints a table showing what packages are used and what modules they correspond to.

```
jdeps zoo.dino.jar
```

```
zoo.dino.jar -> java.base
zoo.dino.jar -> jdk.unsupported
  zoo.dinos   -> java.lang     java.base
  zoo.dinos   -> java.time     java.base
  zoo.dinos   -> java.util     java.base
  zoo.dinos   -> sun.misc      JDK internal API (jdk.unsupported)
```

Note that java.base is always included. It also says which modules contain classes used by the JAR. If we run in summary mode, we only see just the first part where jdeps lists the modules. There are two formats for the summary flag:

jdeps **-s** zoo.dino.jar
jdeps **-summary** zoo.dino.jar

zoo.dino.jar -> java.base
zoo.dino.jar -> jdk.unsupported

For a real project, the dependency list could include dozens or even hundreds of packages. It's useful to see the summary of just the modules. This approach also makes it easier to see whether jdk.unsupported is in the list.

There is also a --module-path option that you can use if you want to look for modules outside the JDK. Unlike other commands, there is no short form for this option on jdeps.

You might have noticed that jdk.unsupported is not in the list of modules you saw in Table 12.8. It's special because it contains internal libraries that developers in previous versions of Java were discouraged from using, although many people ignored this warning. You should not reference it, as it may disappear in future versions of Java.

Using the *--jdk-internals* Flag

The jdeps command has an option to provide details about these unsupported APIs. The output looks something like this:

jdeps **--jdk-internals** zoo.dino.jar

zoo.dino.jar -> jdk.unsupported
  zoo.dinos.Animatronic -> sun.misc.Unsafe
    JDK internal API (jdk.unsupported)

Warning: <omitted warning>

JDK Internal API	Suggested Replacement
sun.misc.Unsafe	See http://openjdk.java.net/jeps/260

The --jdk-internals option lists any classes you are using that call an internal API along with which API. At the end, it provides a table suggesting what you should do about it. If you wrote the code calling the internal API, this message

is useful. If not, the message would be useful to the team that did write the code. You, on the other hand, might need to update or replace that JAR file entirely with one that fixes the issue. Note that -jdkinternals is equivalent to --jdk-internals.

Real World Scenaria

About *sun.misc.Unsafe*

Prior to the Java Platform Module System, classes had to be public if you wanted them to be used outside the package. It was reasonable to use the class in JDK code since that is low-level code that is already tightly coupled to the JDK. Since it was needed in multiple packages, the class was made public. Sun even named it Unsafe, figuring that would prevent anyone from using it outside the JDK.

However, developers are clever and used the class since it was available. A number of widely used open source libraries started using Unsafe. While it is quite unlikely that you are using this class in your project directly, you probably use an open source library that is using it.

The jdeps command allows you to look at these JARs to see whether you will have any problems when Oracle finally prevents the usage of this class. If you find any uses, you can look at whether there is a later version of the JAR that you can upgrade to.

Using Module Files with *jmod*

The final command you need to know for the exam is jmod. You might think a JMOD file is a Java module file. Not quite. Oracle recommends using JAR files for most modules. JMOD files are recommended only when you have native libraries or something that can't go inside a JAR file. This is unlikely to affect you in the real world.

The most important thing to remember is that jmod is only for working with the JMOD files. Conveniently, you don't have to memorize the syntax for jmod. Table 12.9 lists the common modes.

TABLE 12.9 Modes using jmod

Operation	Description
create	Creates JMOD file.
extract	Extracts all files from JMOD. Works like unzipping.
describe	Prints module details such as requires.
list	Lists all files in JMOD file.

hash	Prints or records hashes.

Creating Java Runtimes with *jlink*

One of the benefits of modules is being able to supply just the parts of Java you need. Our zoo example from the beginning of the chapter doesn't have many dependencies. If the user already doesn't have Java or is on a device without much memory, downloading a JDK that is over 150 MB is a big ask. Let's see how big the package actually needs to be! This command creates our smaller distribution:

jlink **--module-path** mods **--add-modules** zoo.animal.talks **--output** zooApp

First we specify where to find the custom modules with -p or --module-path. Then we specify our module names with --add-modules. This will include the dependencies it requires as long as they can be found. Finally, we specify the folder name of our smaller JDK with --output.

The output directory contains the bin, conf, include, legal, lib, and man directories along with a release file. These should look familiar as you find them in the full JDK as well.

When we run this command and zip up the zooApp directory, the file is only 15 MB. This is an order of magnitude smaller than the full JDK. Where did this space savings come from? There are many modules in the JDK we don't need. Additionally, development tools like javac don't need to be in a runtime distribution.

There are a lot more items to customize this process that you don't need to know for the exam. For example, you can skip generating the help documentation and save even more space.

Reviewing Command-Line Options

This section presents a number of tables that cover what you need to know about running command-line options for the exam.

Table 12.10 shows the command-line operations you should expect to encounter on the exam. There are many more options in the documentation. For example, there is a --module option on javac that limits compilation to that module. Luckily, you don't need to know those for the exam.

TABLE 12.10 Comparing command-line operations

Description	Syntax
Compile nonmodular code	**javac** -cp *classpath* -d *directory classesToCompile* **javac** --class-path *classpath* -d *directory classesToCompile*

	javac -classpath *classpath* -d *directory* *classesToCompile*
Run nonmodular code	**java** -cp *classpath package.className* **java** -classpath *classpath package.className* **java** --class-path *classpath package.className*
Compile module	**javac** -p *moduleFolderName* -d *directory* *classesToCompileIncludingModuleInfo* **javac** --module-path *moduleFolderName* -d *directory classesToCompileIncludingModuleInfo*
Run module	**java** -p *moduleFolderName* -m *moduleName/package.className* **java** --module-path *moduleFolderName* --module *moduleName/package.className*
Describe module	**java** -p *moduleFolderName* -d *moduleName* **java** --module-path *moduleFolderName* --describe-module *moduleName* **jar** --file *jarName* --describe-module **jar** -f *jarName* -d
List available modules	**java** --module-path *moduleFolderName* --list-modules
	java -p *moduleFolderName* --list-modules **java** --list-modules
View dependencies	**jdeps** -summary --module-path *moduleFolderName jarName* **jdeps** -s --module-path *moduleFolderName jarName* **jdeps** --jdk-internals *jarName* **jdeps** -jdkinternals *jarName*
Show module resolution	**java** --show-module-resolution -p *moduleFolderName* -m *moduleName* **java** --show-module-resolution --module-path *moduleFolderName* --module *moduleName*
Create runtime JAR	**jlink** -p *moduleFolderName* --add-modules *moduleName* --output *zooApp* **jlink** --module-path *moduleFolderName* --add-modules *moduleName* --output *zooApp*

Table 12.11 shows the options for javac, Table 12.12 shows the options for java, Table 12.13 shows the options for jar, and Table 12.14 shows the options for jdeps. Finally, Table 12.15 shows the options for jlink.

TABLE 12.11 Options you need to know for the exam: javac

Option	Description
-cp <classpath> -classpath <classpath> --class-path <classpath>	Location of JARs in nonmodular program
-d <dir>	Directory in which to place generated class files
-p <path> --module-path <path>	Location of JARs in modular program

TABLE 12.12 Options you need to know for the exam: java

Option	Description
-p <path> --module-path <path>	Location of JARs in modular program
-m <name> --module <name>	Module name to run
-d --describe-module	Describes details of module
--list-modules	Lists observable modules without running program
--show-module-resolution	Shows modules when running program

TABLE 12.13 Options you need to know for the exam: jar

Option	Description
-c --create	Creates new JAR file
-v --verbose	Prints details when working with JAR files
-f --file	JAR filename
-C	Directory containing files to be used to create JAR
-d --describe-module	Describes details of module

TABLE 12.14 Options you need to know for the exam: jdeps

Option	Description
--module-path <path>	Location of JARs in modular program

-s -summary	Summarizes output
--jdk-internals -jdkinternals	Lists uses of internal APIs

TABLE 12.15 Options you need to know for the exam: jlink

Option	Description
-p --module-path <path>	Location of JARs in modular program
--add-modules	List of modules to package
--output	Name of output directory

Comparing Types of Modules

All the modules we've used so far in this chapter are called named modules. There are two other types of modules: automatic modules and unnamed modules. In this section, we describe these three types of modules. On the exam, you will need to be able to compare them.

Named Modules

A *named module* is one containing a module-info.java file. To review, this file appears in the root of the JAR alongside one or more packages. Unless otherwise specified, a module is a named module. Named modules appear on the module path rather than the classpath. Later, you learn what happens if a JAR containing a module-info.java file is on the classpath. For now, just know it is not considered a named module because it is not on the module path.

As a way of remembering this, a named module has the *name* inside the module-info.java file and is on the module path.

Remember from Chapter 7, "Beyond Classes," that the only way for subclasses of sealed classes to be in a different package is to be within the same-named module.

Automatic Modules

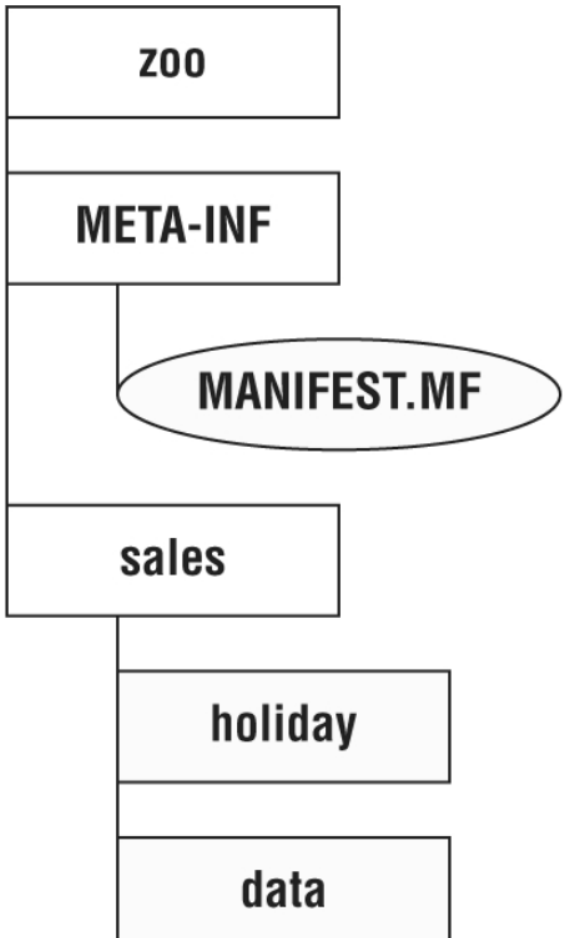
An *automatic module* appears on the module path but does not contain a module-info.java file. It is simply a regular JAR file that is placed on the module path and gets treated as a module.

As a way of remembering this, Java *automatically* determines the module name. The code referencing an automatic module treats it as if there is a module-info.java file present. It automatically exports all packages. It also determines the module name. How does it determine the module name, you ask? Excellent question.

To answer this, we need to provide a bit of history on JAR files and module adoption. Every JAR file contains a special folder called META-INF and, within it, a text file called MANIFEST.MF. It can be created automatically when the JAR is created or by hand by the JAR's author. Getting back to modules, many Java libraries weren't quite ready to modularize when the feature was introduced. The authors were encouraged to declare the name they intended to use for the module by adding a property named Automatic-Module-Name into their MANIFEST.MF file.

About the *MANIFEST.MF* File

A JAR file contains a special text file called META-INF/MANIFEST.MF that contains information about the JAR. It's been around significantly longer than modules—since the early days of Java and JARs, to be exact. The figure shows how the manifest fits into the directory structure of a JAR file.



The manifest contains extra information about the JAR file. For example, it often contains the version of Java used to build the JAR file. For command-line programs, the class with the main() method is commonly specified.

Each line in the manifest is a key/value pair separated by a colon. You can think of the manifest as a map of property names and values. The default manifest in Java 17 looks like this:

```
Manifest-Version: 1.0
Created-By: 17 (Oracle Corporation)
```

Specifying a single property in the manifest allowed library providers to make things easier for applications that wanted to use their library in a modular application. You can think of it as a promise that when the library becomes a named module, it will use the specified module name.

If the JAR file does not specify an automatic module name, Java will still allow you to use it in the module path. In this case, Java will determine the module name for you. We'd say that this happens automatically, but the joke is probably wearing thin by now.

Java determines the automatic module name by basing it on the filename of the JAR file. Let's go over the rules by starting with an example. Suppose we have a JAR file named holiday-calendar-1.0.0.jar.

First Java will remove the extension .jar from the name. Then Java will remove the version from the end of the JAR filename. This is important because we want module names to be consistent. Having a different automatic module name every time you upgraded to a new version would not be good! After all, this would force you to change the module declaration of your nice, clean, modularized application every time you pulled in a later version of the holiday calendar JAR.

Removing the version and extension gives us holiday-calendar. This leaves us with a problem. Dashes (-) are not allowed in module names. Java solves this problem by converting any special characters in the name to dots (.). As a result, the module name is holiday.calendar. Any characters other than letters and numbers are considered special characters in this replacement. Finally, any adjacent dots or leading/trailing dots are removed.

Since that's a number of rules, let's review the algorithm in a list for determining the name of an automatic module:

- If the MANIFEST.MF specifies an Automatic-Module-Name, use that. Otherwise, proceed with the remaining rules.
- Remove the file extension from the JAR name.

- Remove any version information from the end of the name. A version is digits and dots with possible extra information at the end: for example, -1.0.0 or -1.0-RC.
- Replace any remaining characters other than letters and numbers with dots.
- Replace any sequences of dots with a single dot.
- Remove the dot if it is the first or last character of the result.

Table 12.16 shows how to apply these rules to two examples where there is no automatic module name specified in the manifest.

TABLE 12.16 Practicing with automatic module names

#	Description	Example 1	Example 2
1	Beginning JAR name	commons2-x-1.0.0-SNAPSHOT.jar	mod_$-1.0.jar
2	Remove file extension	commons2-x-1.0.0-SNAPSHOT	mod_$-1.0
3	Remove version information	commons2-x	mod_$
4	Replace special characters	commons2.x	mod..
5	Replace sequence of dots	commons2.x	mod.
6	Remove leading/ trailing dots (results in the automatic module name)	commons2.x	mod

While the algorithm for creating automatic module names does its best, it can't always come up with a good name. For example, 1.2.0-calendar-1.2.2-good-1.jar isn't conducive. Luckily, such names are rare and out of scope for the exam.

Unnamed Modules

An *unnamed module* appears on the classpath. Like an automatic module, it is a regular JAR. Unlike an automatic module, it is on the classpath rather than the module path. This means an unnamed module is treated like old code and a second-class citizen to modules.

An unnamed module does not usually contain a module-info.java file. If it happens to contain one, that file will be ignored since it is on the classpath.

Unnamed modules do not export any packages to named or automatic modules. The unnamed module can read from any JARs on the classpath or module path. You can think of an unnamed module as code that works the way Java worked before modules. Yes, we know it is confusing for something that isn't really a module to have the word *module* in its name.

Reviewing Module Types

You can expect to get questions on the exam comparing the three types of modules. Please study Table 12.17 thoroughly and be prepared to answer questions about these items in any combination. A key point to remember is that code on the classpath can access the module path. By contrast, code on the module path is unable to read from the classpath.

TABLE 12.17 Properties of module types

Property	Named	Automatic	Unnamed
Does a _____ module contain a module-info.java file?	Yes	No	Ignored if present
Which packages does a _____ module export to other modules?	Those in module-info.java file	All packages	No packages
Is a _____ module readable by other modules on the module path?	Yes	Yes	No
Is a _____ module readable by other JARs on the classpath?	Yes	Yes	Yes

Migrating an Application

Many applications were not designed to use the Java Platform Module System because they were written before it was created or chose not to use it. Ideally, they were at least designed with projects instead of as a big ball of mud. This section gives you an overview of strategies for migrating an existing application to use modules. We cover ordering modules, bottom-up migration, top-down migration, and how to split up an existing project.

Real World Scenaria

Migrating Your Applications at Work

The exam exists in a pretend universe where there are no open source dependencies and applications are very small. These scenarios make learning and discussing migration far easier. In the real world, applications have libraries that haven't been updated in 10 or more years, complex dependency graphs, and all sorts of surprises.

Note that you can use all the features of Java 17 without converting your application to modules (except the features in this module chapter, of course!). Please make sure you have a reason for migration and don't think it is required.

This chapter does a great job teaching you what you need to know for the exam. However, it does not adequately prepare you to convert real applications to use modules. If you find yourself in that situation, consider reading *The Java Module System* by Nicolai Parlog (Manning Publications, 2019).

Determining the Order

Before we can migrate our application to use modules, we need to know how the packages and libraries in the existing application are structured. Suppose we have a simple application with three JAR files, as shown in Figure 12.14. The dependencies between projects form a graph. Both of the representations in Figure 12.14 are equivalent. The arrows show the dependencies by pointing from the project that will require the dependency to the one that makes it available. In the language of modules, the arrow will go from requires to exports.

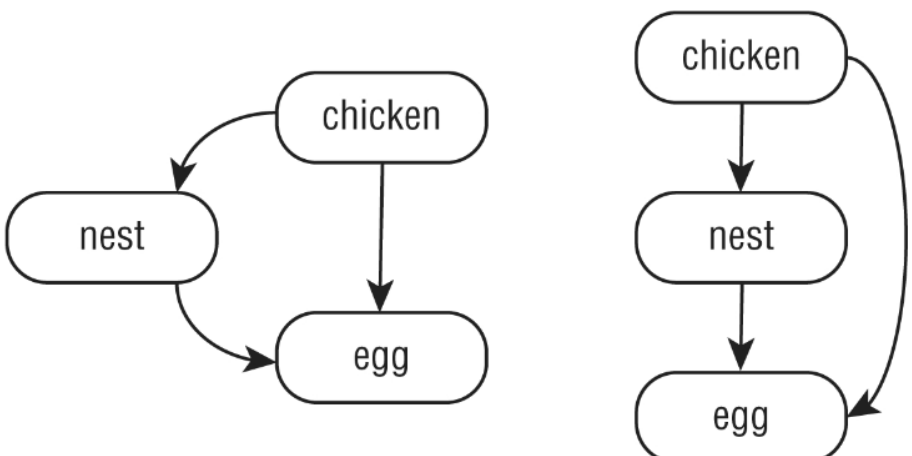


FIGURE 12.14 Determining the order

The right side of the diagram makes it easier to identify the top and bottom that top-down and bottom-up migration refer to. Projects that do not have any dependencies are at the bottom. Projects that do have dependencies are at the top.

In this example, there is only one order from top to bottom that honors all the dependencies. Figure 12.15 shows that the order is not always unique. Since two of the projects do not have an arrow between them, either order is allowed when deciding migration order.

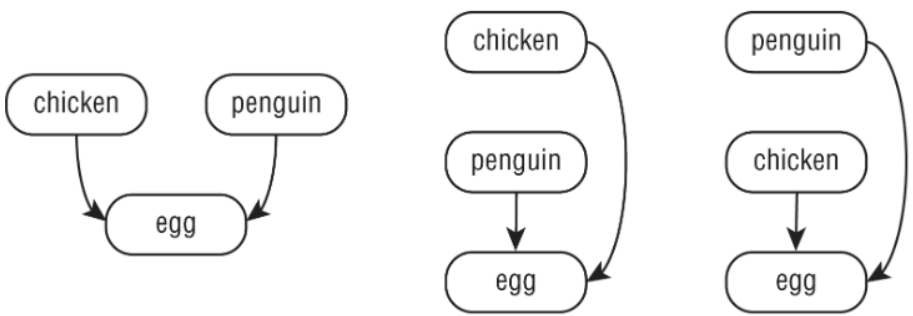


FIGURE 12.15 Determining the order when not unique

Exploring a Bottom-Up Migration Strategy

The easiest approach to migration is a bottom-up migration. This approach works best when you have the power to convert any JAR files that aren't already modules. For a bottom-up migration, you follow these steps:

1. Pick the lowest-level project that has not yet been migrated. (Remember the way we ordered them by dependencies in the previous section?)
2. Add a module-info.java file to that project. Be sure to add any exports to expose any package used by higher-level JAR files. Also, add a requires directive for any modules this module depends on.
3. Move this newly migrated named module from the classpath to the module path.
4. Ensure that any projects that have not yet been migrated stay as unnamed modules on the classpath.
5. Repeat with the next-lowest-level project until you are done.

You can see this procedure applied to migrate three projects in Figure 12.16. Notice that each project is converted to a module in turn.

With a bottom-up migration, you are getting the lower-level projects in good shape. This makes it easier to migrate the top-level projects at the end. It also encourages care in what is exposed.

During migration, you have a mix of named modules and unnamed modules. The named modules are the lower-level ones that have been migrated. They are on the module path and not allowed to access any unnamed modules.

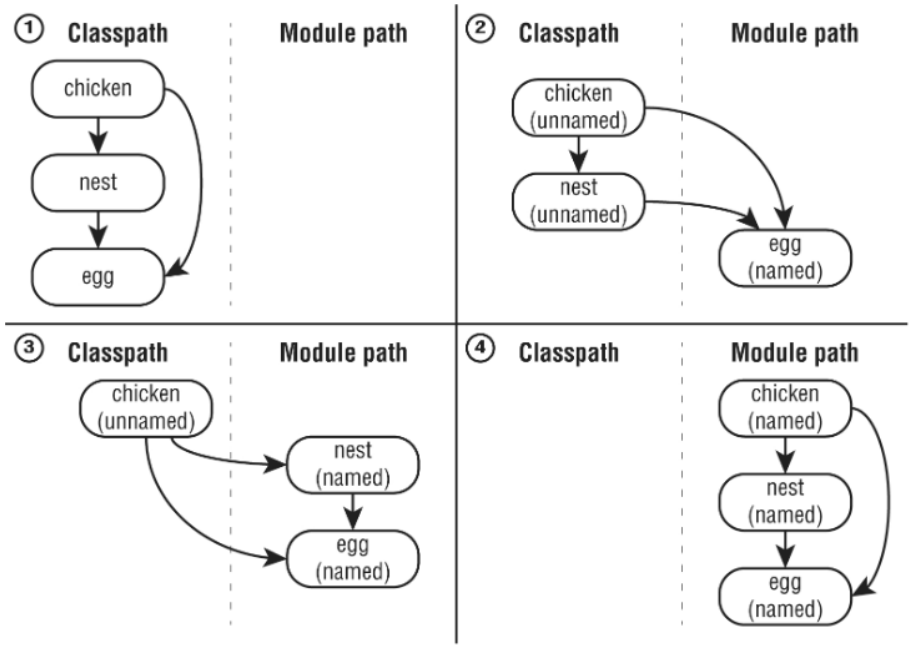


FIGURE 12.16 Bottom-up migration

The unnamed modules are on the classpath. They can access JAR files on both the classpath and the module path.

Exploring a Top-Down Migration Strategy

A top-down migration strategy is most useful when you don't have control of every JAR file used by your application. For example, suppose another team owns one project. They are just too busy to migrate. You wouldn't want this situation to hold up your entire migration.

For a top-down migration, you follow these steps:

1. Place all projects on the module path.
2. Pick the highest-level project that has not yet been migrated.
3. Add a module-info.java file to that project to convert the automatic module into a named module. Again, remember to add any exports or requires directives. You can use the automatic module name of other modules when writing the requires directive since most of the projects on the module path do not have names yet.
4. Repeat with the next-highest-level project until you are done.

You can see this procedure applied in order to migrate three projects in Figure 12.17. Notice that each project is converted to a module in turn.

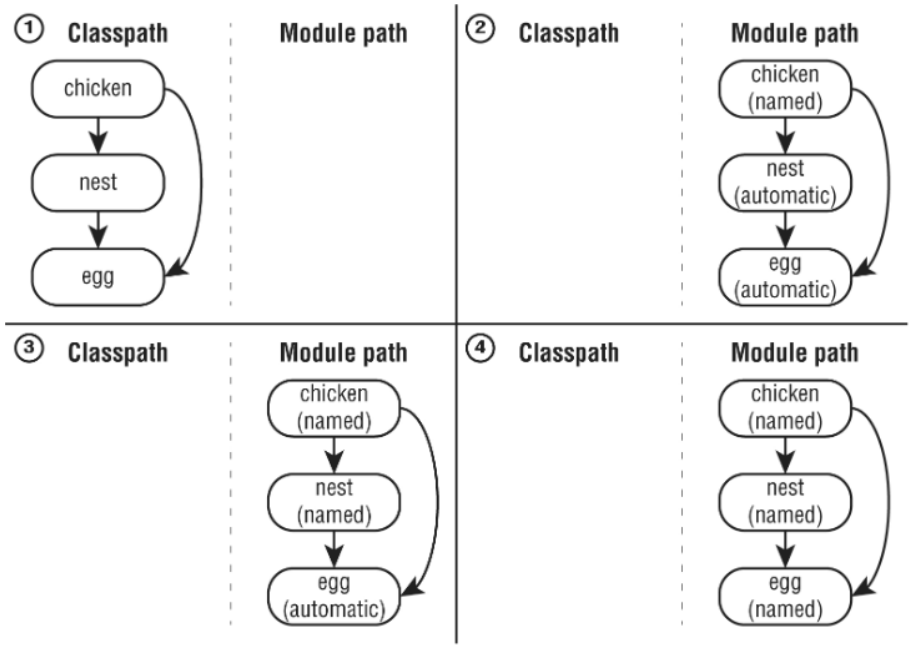


FIGURE 12.17 Top-down migration

With a top-down migration, you are conceding that all of the lower-level dependencies are not ready but that you want to make the application itself a module.

During migration, you have a mix of named modules and automatic modules. The named modules are the higher-level ones that have been migrated. They are on the module path and have access to the automatic modules. The automatic modules are also on the module path.

Table 12.18 reviews what you need to know about the two main migration strategies. Make sure you know it well.

TABLE 12.18 Comparing migration strategies

Category	Bottom-Up	Top-Down
Project that depends on all others	Unnamed module on classpath	Named module on module path
Project that has no dependencies	Named module on module path	Automatic module on module path

Splitting a Big Project into Modules

For the exam, you need to understand the basic process of splitting a big project into modules. You won't be given a big project, of course. After all, there is only so much space to ask a question. Luckily, the process is the same for a small project.

Suppose you start with an application that has a number of packages. The first step is to break them into logical groupings and draw the dependencies between them. Figure 12.18 shows an imaginary system's decomposition. Notice that there are seven packages on both the left and right sides. There are fewer modules because some packages share a module.

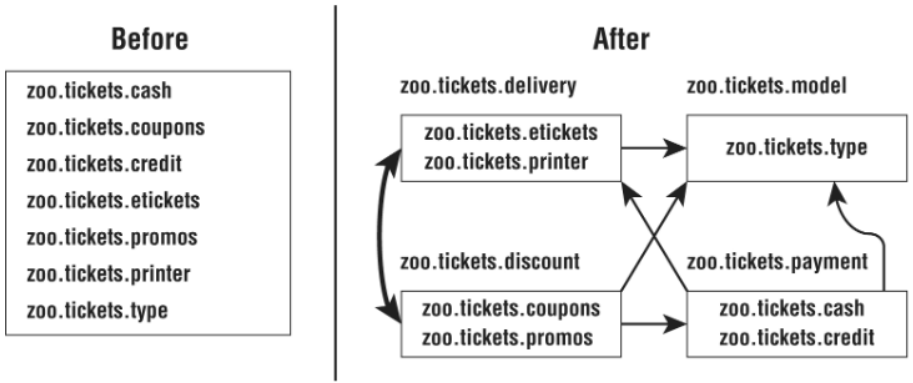


FIGURE 12.18 First attempt at decomposition

There's a problem with this decomposition. Do you see it? The Java Platform Module System does not allow for *cyclic dependencies*. A cyclic dependency, or *circular dependency*, is when two things directly or indirectly depend on each other. If the zoo.tickets.delivery module requires the zoo.tickets.discount module, zoo.tickets.discount is not allowed to require the zoo.tickets.delivery module.

Now that we know that the decomposition in Figure 12.18 won't work, what can we do about it? A common technique is to introduce another module. That module contains the code that the other two modules share. Figure 12.19 shows the new modules without any cyclic dependencies. Notice the new module zoo.tickets.etech. We created new packages to put in that module. This allows the developers to put the common code in there and break the dependency. No more cyclic dependencies!

Failing to Compile with a Cyclic Dependency

It is extremely important to understand that Java will not allow you to compile modules that have circular dependencies. In this section, we look at an example leading to that compiler error.

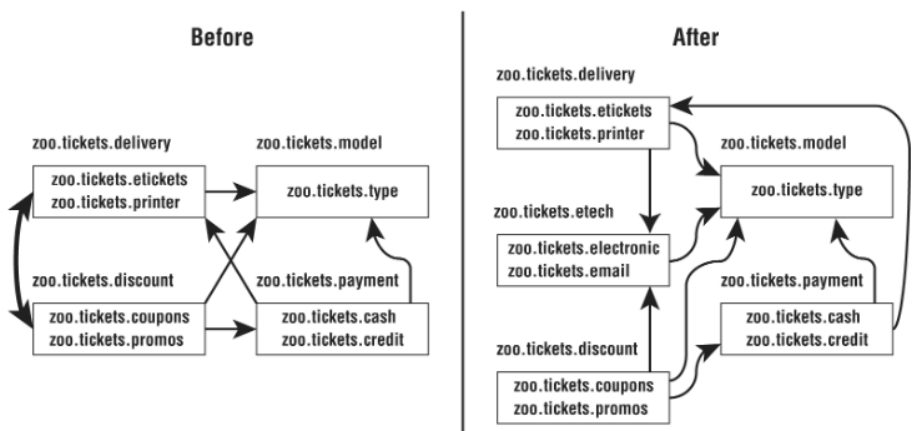


FIGURE 12.19 Removing the cyclic dependencies

Consider the zoo.butterfly module described here:

```
// Butterfly.java
package zoo.butterfly;
public class Butterfly {
 private Caterpillar caterpillar;
}

// module-info.java
module zoo.butterfly {
  exports zoo.butterfly;
  requires zoo.caterpillar;
}
```

We can't compile this yet as we need to build zoo.caterpillar first. After all, our butterfly requires it. Now we look at zoo.caterpillar:

```
// Caterpillar.java
package zoo.caterpillar;
public class Caterpillar {
  Butterfly emergeCocoon() {
    // logic omitted
  }
}

// module-info.java
module zoo.caterpillar {
  exports zoo.caterpillar;
  requires zoo.butterfly;
}
```

We can't compile this yet as we need to build zoo.butterfly first. Uh oh! Now we have a stalemate. Neither module can be compiled. This is our circular dependency problem at work.

This is one of the advantages of the module system. It prevents you from writing code that has a cyclic dependency. Such code won't even compile!

You might be wondering what happens if three modules are involved. Suppose module ballA requires module ballB and ballB requires module ballC. Can module ballC require module ballA? No. This would create a cyclic dependency. Don't believe us? Try drawing it. You can follow your pencil around the circle from ballA to ballB to ballC to ballA to … well, you get the idea. There are just too many balls in the air!

Java will still allow you to have a cyclic dependency between packages within a module. It enforces that you do not have a cyclic dependency between modules.

Summary

The Java Platform Module System organizes code at a higher level than packages. Each module contains one or more packages and a module-info.java file. The java.base module is most common and is automatically supplied to all modules as a dependency.

The process of compiling and running modules uses the --module-path, also known as -p. Running a module uses the --module option, also known as -m. The class to run is specified in the format moduleName/className.

The module declaration file supports a number of directives. The exports directive specifies that a package should be accessible outside the module. It can optionally restrict that export to a specific package. The requires directive is used when a module depends on code in another module. Additionally, requires transitive can be used when all modules that require one module should always require another. The provides and uses directives are used when sharing and consuming a service. Finally, the opens directive is used to allow access via reflection.

Both the java and jar commands can be used to describe the contents of a module. The java command can additionally list available modules and show module resolution. The jdeps command prints information about packages used in addition to module-level information. The jmod command is used when dealing with files that don't meet the requirements for a JAR. The jlink command creates a smaller Java runtime image.

There are three types of modules. Named modules contain a module-info.java file and are on the module path. They can read only from the module path. Automatic modules are also on the module path but have not yet been modularized. They might have an automatic module name set in the manifest. Unnamed modules are on the classpath.

The two most common migration strategies are top-down and bottom-up migration. Top-down migration starts migrating the module with the most dependencies and places all other modules on the module path. Bottom-up migration starts migrating a module with no dependencies and moves one module to the module path at a time. Both of these strategies require ensuring that you do not have any cyclic dependencies since the Java Platform Module System will not allow cyclic dependencies to compile.