Chapter 17 Modular Applications

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Migration to a Modular Application**
  - Migrate the application developed using a Java version prior to SE 9 to SE 11 including top-down and bottom-up migration, splitting a Java SE 8 application into modules for migration
  - Use jdeps to determine dependencies and identify ways to address the cyclic dependencies
- **Services in a Modular Application**
  - Describe the components of Services including directives
  - Design a service type, load services using ServiceLoader, check for dependencies of the services including consumer and provider modules

If you took the 1Z0-815 exam, you learned the basics of modules. If not, read Chapter 11, “Modules,” before reading this chapter. That will bring you up to speed. Luckily, you don't have to memorize as many command-line options for the 1Z0-816 exam.

This chapter covers 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.

You aren't required to create modules by hand or memorize as many commands to compile and run modules on the 1Z0-816 exam. We still include them in the chapter so you can study and follow along. Feel free to use the files we've already set up in the GitHub repo linked to from www.selikoff.net/ocp11-2.

Reviewing Module Directives

Since you are expected to know highlights of the `module-info.java` file for the 1Z0-816, we have included the relevant parts here as a reference. If anything in Table 17.1 is unclear or unfamiliar, please stop and read Chapter 11 before continuing in this chapter.

TABLE 17.1 Common module directives

Derivative	Description
`exports` *`<package>`*	Allows all modules to access the package
`exports` *`<package>`* `to` *`<module>`*	Allows a specific module to access the package
`requires` *`<module>`*	Indicates module is dependent on another module
`requires transitive` *`<module>`*	Indicates the module and that all modules that use this module are dependent on another module
`uses` *`<interface>`*	Indicates that a module uses a service
`provides` *`<interface>`* `with` *`<class>`*	Indicates that a module provides an implementation of a service

If you don't have any experience with `uses` or `provides`, don't worry—they will be covered in this chapter.

Comparing Types of Modules

The modules you learned about in Chapter 11 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.

CLASSPATH VS. MODULE PATH

Before we get started, a brief reminder that the Java runtime is capable of using class and interface types from both the classpath and the module path, although the rules for each are a bit different. An application can access any type in the classpath that is exposed via standard Java access modifiers, such as a `public` class.

On the other hand, `public` types in the module path are not automatically available. While Java access modifiers must still be used, the type must also be in a package that is exported by the module in which it is defined. In addition, the module making use of the type must contain a dependency on the module.

Named Modules

A *named module* is one containing a `module-info` 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. You'll learn what happens if a JAR containing a `module-info` 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` file and is on the module path. Figure 17.1 shows the contents of a JAR file for a named module. It contains two packages in addition to the `module-info.class`.

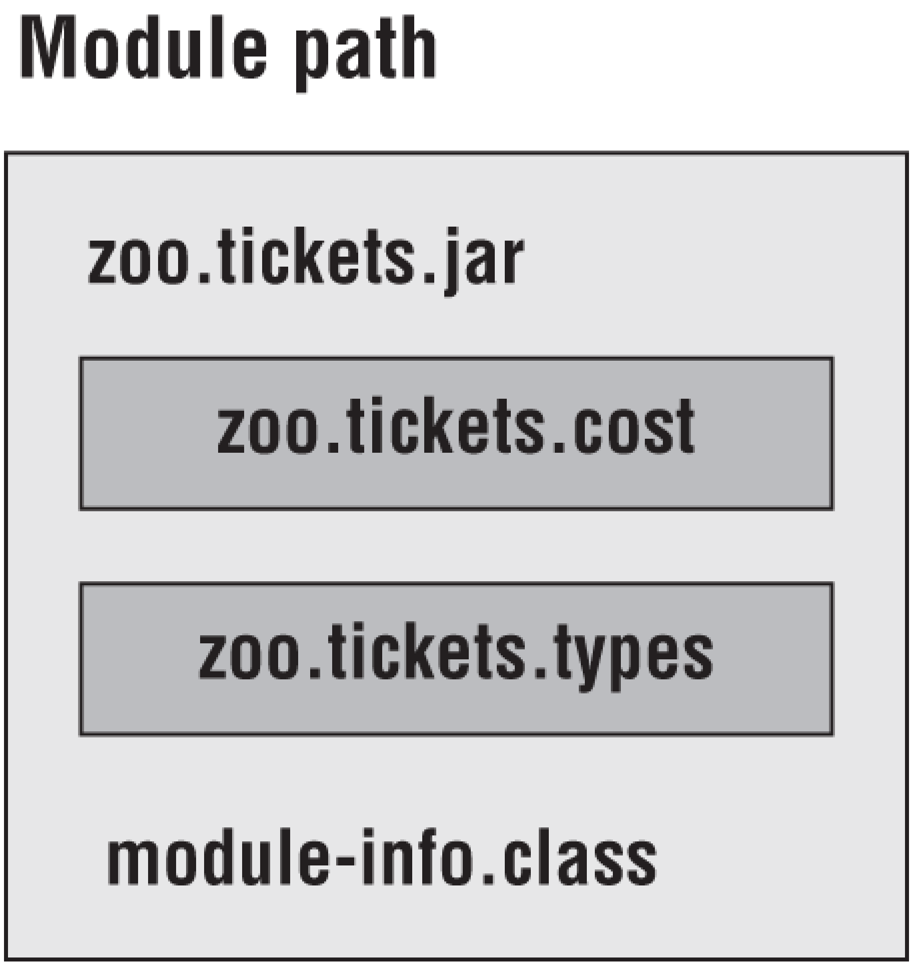


FIGURE 17.1 A named module

Automatic Modules

An *automatic module* appears on the module path but does not contain a `module-info` 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. Figure 17.2 shows an automatic module with two packages.

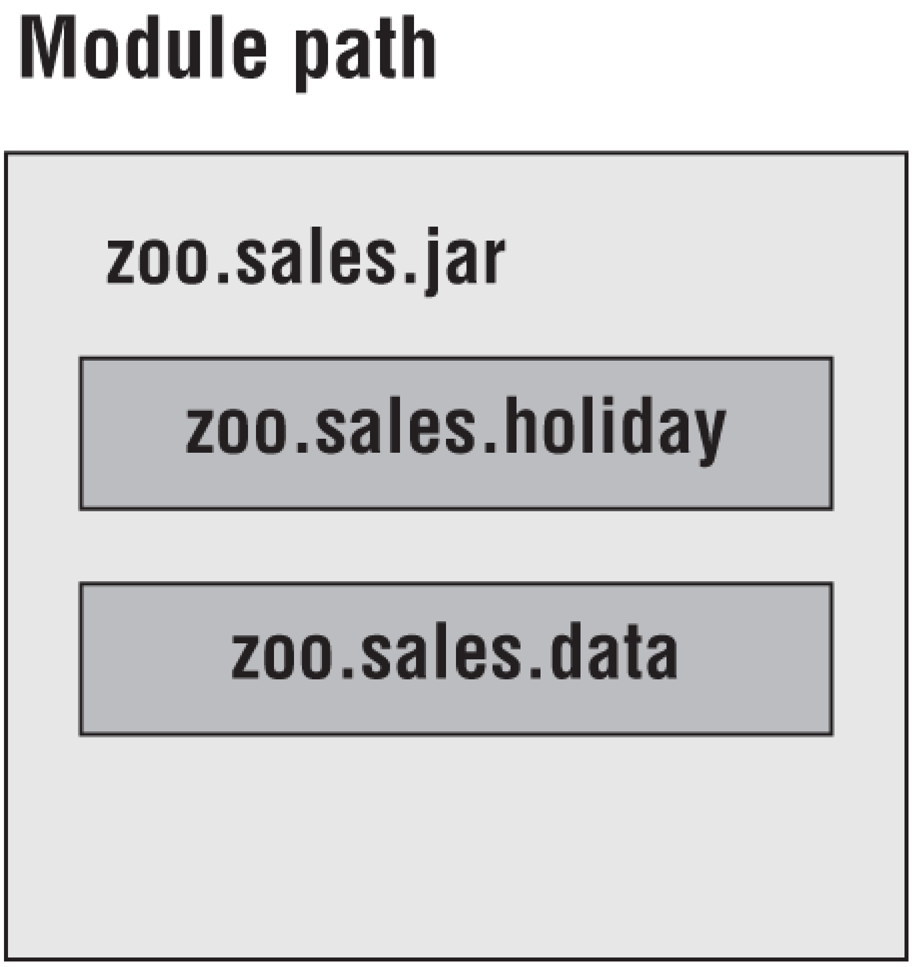
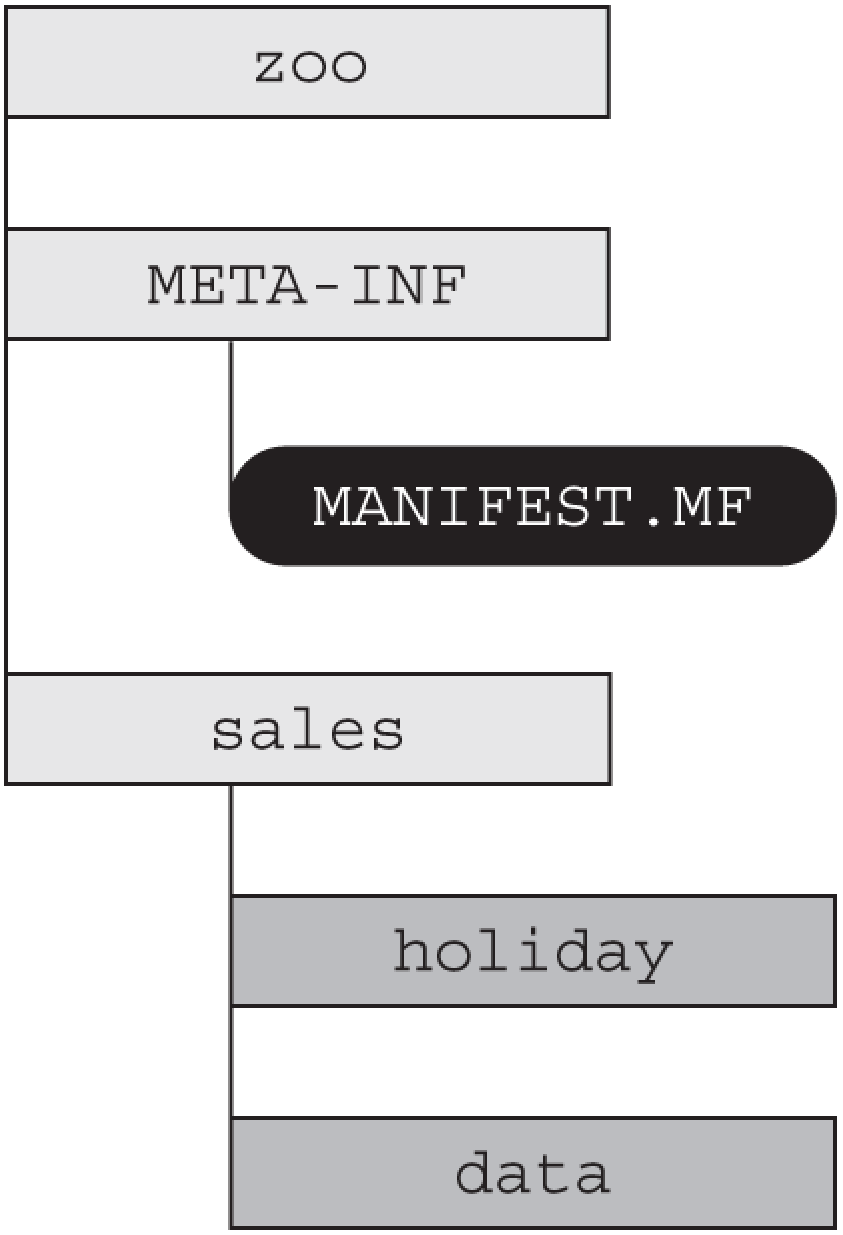


FIGURE 17.2 An automatic module

Real World Scenario

ABOUT THE MANIFEST

A JAR file is a zip file with a special directory named `META-INF`. This directory contains one or more files. The `MANIFEST.MF` file is always present. 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 11 looks like this:

```
Manifest-Version: 1.0
Created-By: 11.0.2 (Oracle Corporation)
```

The code referencing an automatic module treats it as if there is a `module-info` file present. It automatically exports all packages. It also determines the module name. How does it determine the module name? you ask. Excellent question.

When Java 9 was released, authors of Java libraries were encouraged to declare the name they intended to use for the module in the future. All they had to do was set a property called `Automatic-Module-Name` in the `MANIFEST.MF` file.

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 off 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-info` file 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 17.2 shows how to apply these rules to two examples where there is no automatic module name specified in the manifest.

TABLE 17.2 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. Figure 17.3 shows an unnamed module with one package.

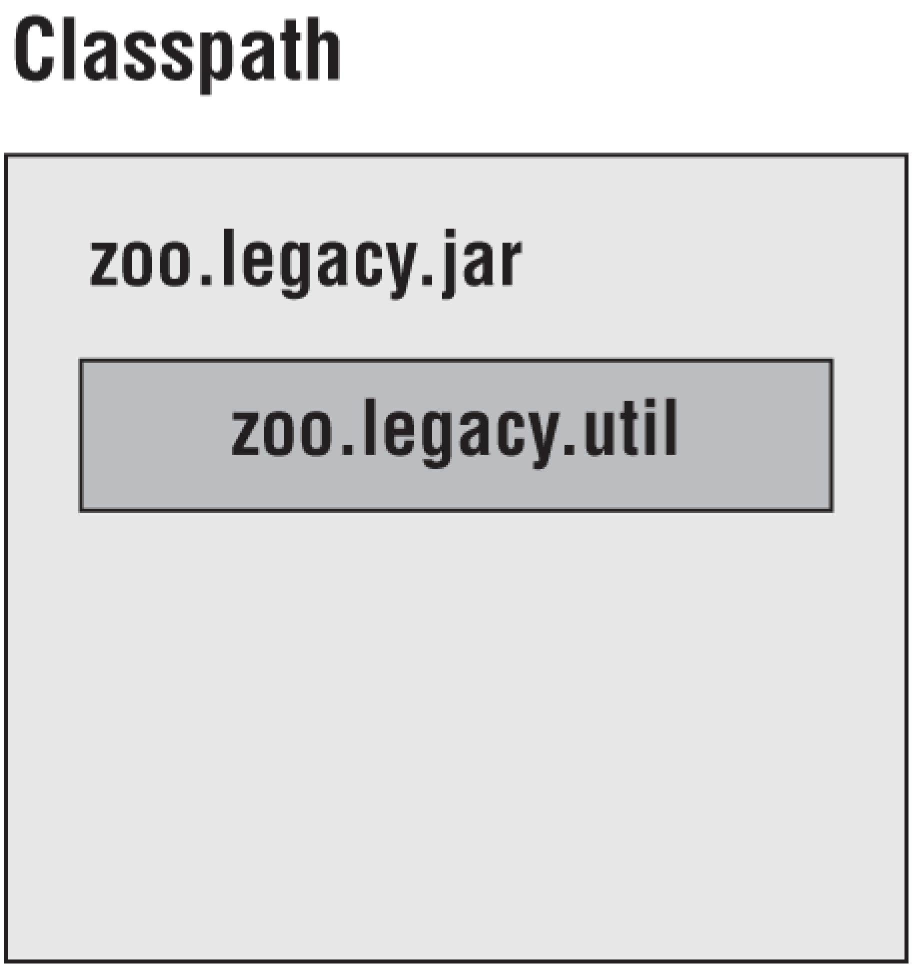


FIGURE 17.3 An unnamed module

An unnamed module does not usually contain a `module-info` 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 to have something that isn't really a module having the word *module* in its name.

Comparing Module Types

You can expect to get questions on the exam comparing the three types of modules. Please study Table 17.3 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 17.3 Properties of modules types

Property	Named	Automatic	Unnamed
A ______ module contains a `module-info` file?	Yes	No	Ignored if present
A ______ module exports which packages to other modules?	Those in the `module-info` file	All packages	No packages
A ______ module is readable by other modules on the module path?	Yes	Yes	No
A ______ module is readable by other JARs on the classpath?	Yes	Yes	Yes

Analyzing JDK Dependencies

In this part of the chapter, we look at modules that are supplied by the JDK. We also look at the `jdeps` command for identifying such module dependencies.

Identifying Built-in Modules

Prior to Java 9, developers could use any package in the JDK by merely importing it into the application. This meant the whole JDK had to be available at runtime because a program could potentially need anything. With modules, your application specifies which parts of the JDK it uses. This allows the application to run on a full JDK or a subset.

You might be wondering what happens if you try to run an application that references a package that isn't available in the subset. No worries! The `requires` directive in the `module-info` file specifies which modules need to be present at both compile time and runtime. This means they are guaranteed to be available for the application to run.

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 17.4 lists some common modules and what they contain.

TABLE 17.4 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 the exam beyond the module name
`java.logging`	Logging	Not on the exam beyond the module name
`java.sql`	JDBC	Chapter 21, “JDBC”
`java.xml`	Extensible Markup Language (XML)	Not on the exam beyond the 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 17.5 lists all the modules that begin with `java`.

TABLE 17.5 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 17.6 lists all the modules that begin with `jdk`. We recommend reviewing this right before the exam to increase the chances of them sounding familiar. You don't have to memorize them, but you should be able to pick them out of a lineup.

TABLE 17.6 Java modules prefixed with jdk

jdk.accessiblity	jdk.jconsole	jdk.naming.dns
jdk.attach	jdk.jdeps	jdk.naming.rmi
jdk.charsets	jdk.jdi	jdk.net
jdk.compiler	jdk.jdwp.agent	jdk.pack
jdk.crypto.cryptoki	jdk.jfr	jdk.rmic
jdk.crypto.ec	jdk.jlink	jdk.scripting.nashorn
jdk.dynalink	jdk.jshell	jdk.sctp
jdk.editpad	jdk.jsobject	jdk.security.auth
jdk.hotspot.agent	jdk.jstatd	jdk.security.jgss
jdk.httpserver	jdk.localdata	jdk.xml.dom
jdk.jartool	jdk.management	jdk.zipfs
jdk.javadoc	jdk.management.agent	
jdk.jcmd	jdk.management.jfr	

Using *jdeps*

The `jdeps` command gives you information about dependencies. Luckily, you are not expected to memorize all the options for the 1Z0-816 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 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'll 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)
```

If we run in summary mode, we only see just the first part where `jdeps` lists the modules.

```
jdeps -s 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.

NOTE

You might have noticed that `jdk.unsupported` is not in the list of modules you saw in Table 17.6. 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.

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 who 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 Scenario

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, it is likely you 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.

Migrating an Application

All applications developed for Java 8 and earlier were not designed to use the Java Platform Module System because it did not exist yet. Ideally, they were at least designed with projects instead of as a big ball of mud. This section will give you an overview of strategies for migrating an existing application to use modules. We will cover ordering modules, bottom-up migration, top-down migration, and how to split up an existing project.

Real World Scenario

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 11 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 for actually converting 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 17.4. The dependencies between projects form a graph. Both of the representations in Figure 17.4 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 the `exports` .

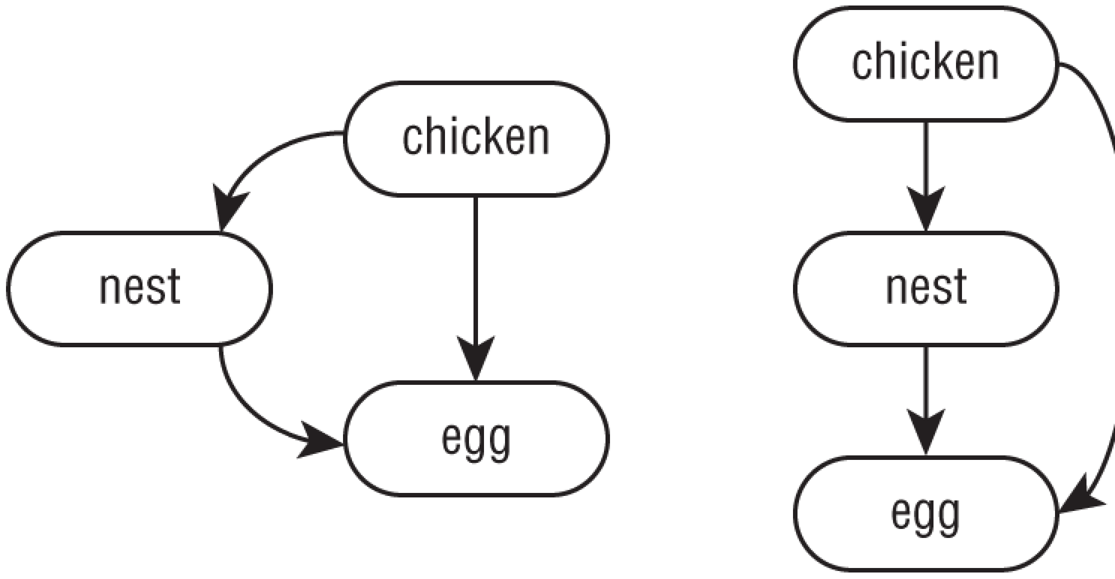


FIGURE 17.4 Determining the order

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 17.5 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.

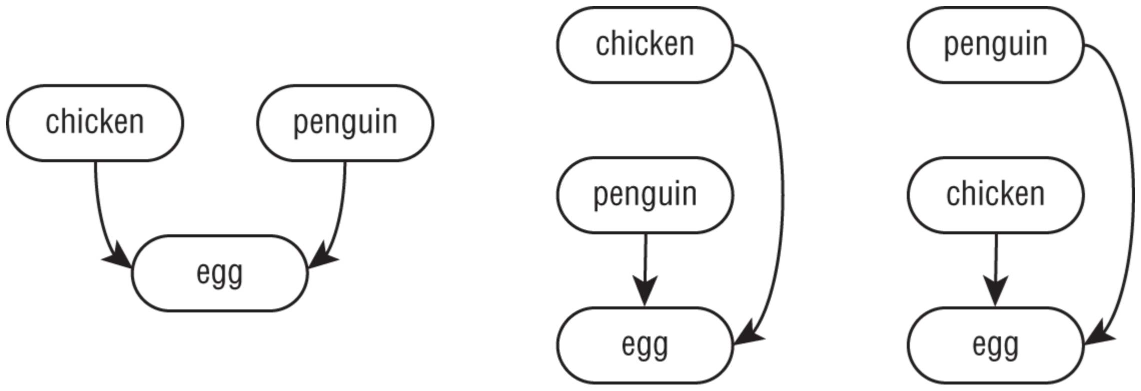


FIGURE 17.5 Determining the order when not unique

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 it depends on.
3. Move this newly migrated named module from the classpath to the module path.
4. Ensure 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 17.6. Notice that each project is converted to a module in turn.

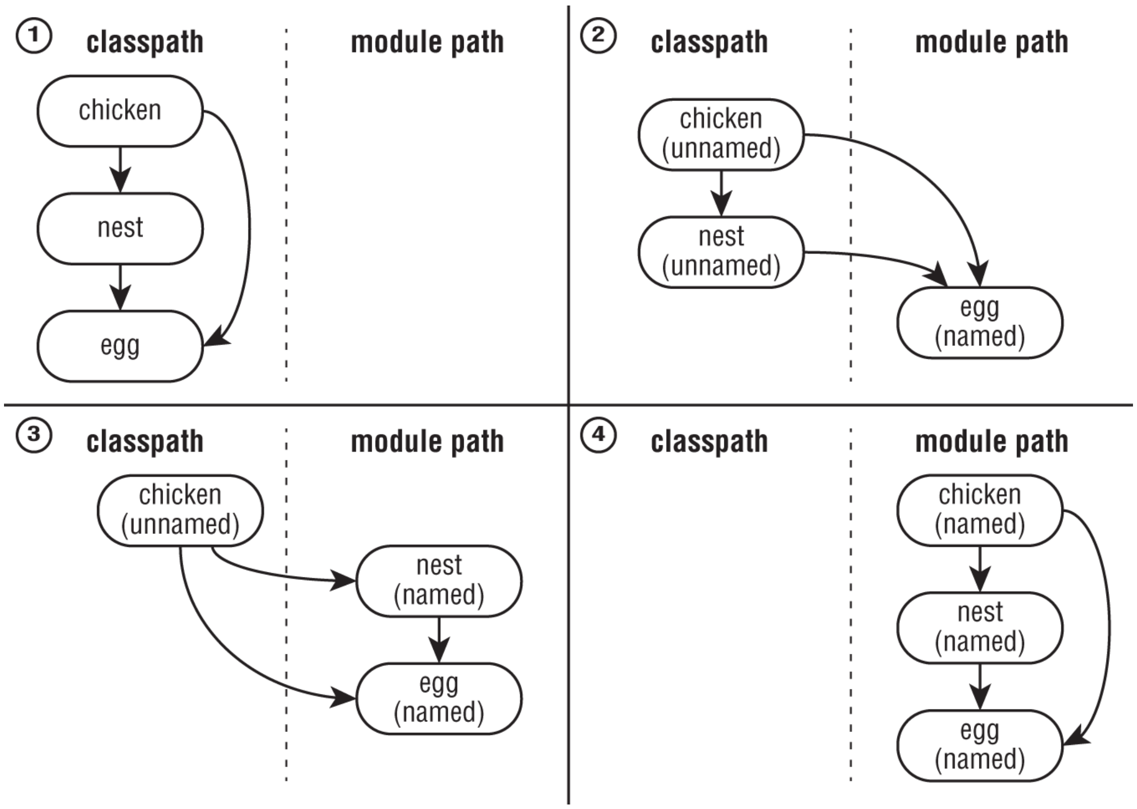


FIGURE 17.6 Bottom-up migration

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.

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` 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-lowest-level project until you are done.

You can see this procedure applied in order to migrate three projects in Figure 17.7. Notice that each project is converted to a module in turn.

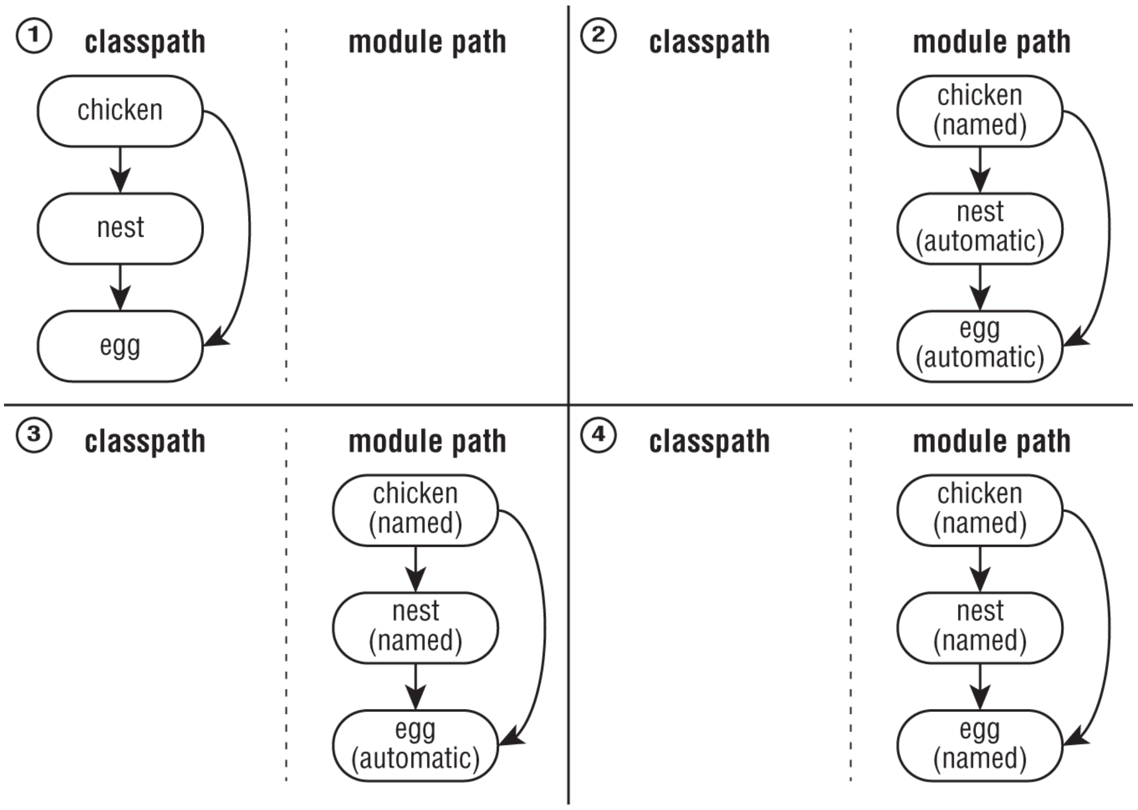


FIGURE 17.7 Top-down migration

With a top-down migration, you are conceding that all of the lower-level dependencies are not ready but 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 17.7 reviews what you need to know about the two main migration strategies. Make sure you know it well.

TABLE 17.7 Comparing migration strategies

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

Splitting a Big Project into Modules

For the exam, you need to understand the basic process of splitting up 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 up into logical groupings and draw the dependencies between them. Figure 17.8 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.

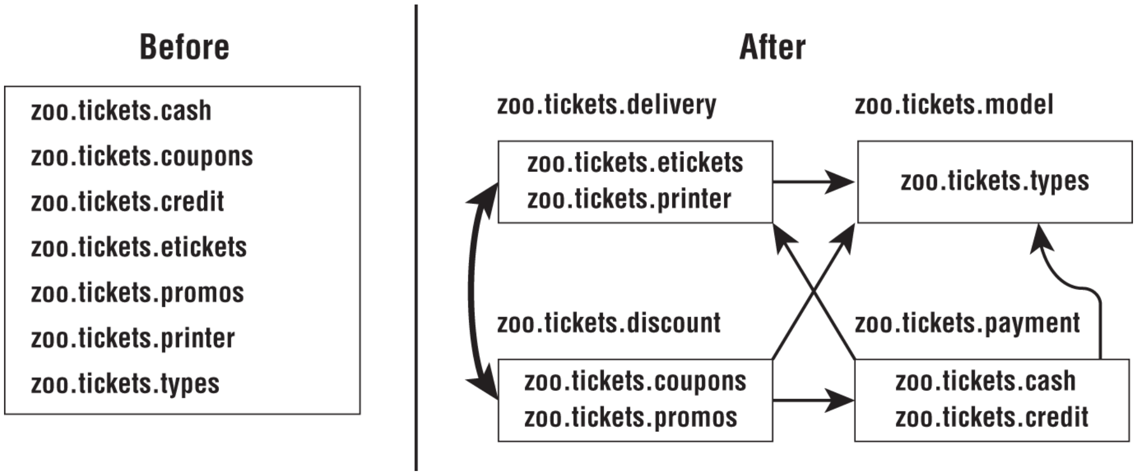


FIGURE 17.8 First attempt at decomposition

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, the `zoo.tickets.discount` is not allowed to require the `zoo.tickets.delivery` module.

Now that we all know that the decomposition in Figure 17.8 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 17.9 shows the new modules without any cyclic dependencies. Notice the new module `zoo.tickets.discount`. We created a new package to put in that module. This allows the developers to put the common code in there and break the dependency. No more cyclic dependencies!

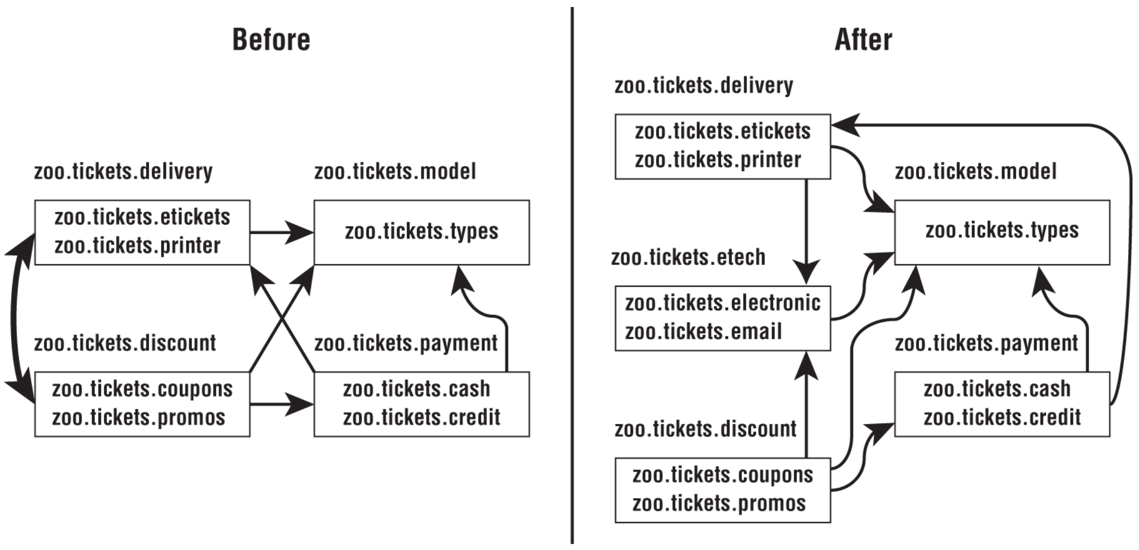


FIGURE 17.9 Removing the cyclic dependencies

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 between each other. In this section, we will look at an example leading to that compiler error.

First, let's create a module named `zoo.butterfly` that has a single class in addition to the `module-info.java` file. If you need a reminder where the files go in the directory structure, see Chapter 11 or the online code example.

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

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

We can compile the butterfly module and create a JAR file in the `mods` directory named `zoo.butterfly.jar`. Remember to create a `mods` directory if one doesn't exist in your folder structure.

```
javac -d butterflyModule
   butterflyModule/zoo/butterfly/Butterfly.java
   butterflyModule/module-info.java

jar -cvf mods/zoo.butterfly.jar -C butterflyModule/ .
```

Now we create a new module, `zoo.caterpillar`, that depends on the existing `zoo.butterfly` module. This time, we will create a module with two classes in addition to the `module-info.java` file.

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

// CaterpillarLifecycle.java
package zoo.caterpillar;
import zoo.butterfly.Butterfly;
public interface CaterpillarLifecycle {
   Butterfly emergeCocoon();
}

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

Again, we will compile and create a JAR file. This time it is named `zoo.caterpillar.jar`.

```
javac -p mods -d caterpillarModule
   caterpillarModule/zoo/caterpillar/*.java
   caterpillarModule/module-info.java
jar -cvf mods/zoo.caterpillar.jar -C caterpillarModule/ .
```

At this point, we want to add a method for a butterfly to make caterpillar eggs. We decide to put it in the `Butterfly` module instead of the `CaterpillarLifecycle` class to demonstrate a cyclic dependency.

We know this requires adding a dependency, so we do that first. Updating the `module-info.java` file in the `zoo.butterfly` module looks like this:

```
module zoo.butterfly {
   exports zoo.butterfly;
   requires zoo.caterpillar;
}
```

We then compile it with the module path `mods` so `zoo.caterpillar` is visible:

```
javac -p mods -d butterflyModule
   butterflyModule/zoo/butterfly/Butterfly.java
   butterflyModule/module-info.java
```

The compiler complains about our cyclic dependency.

```
butterflyModule/module-info.java:3: error:
   cyclic dependence involving zoo.caterpillar
      requires zoo.caterpillar;
```

This is one of the advantages of the module system. It prevents you from writing code that has 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 here!

TIP

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.

Creating a Service

In this section, you'll 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.

Services are not new to Java. In fact, the `ServiceLoader` class was introduced in Java 6. It was used to make applications *extensible*, so you could add functionality without recompiling the whole application. What is new is the integration with modules.

We will be using a tour application in the services section. It has four modules shown in Figure 17.10. In this example, the `zoo.tours.api` and `zoo.tours.reservations` models make up the service since they consist of the interface and lookup functionality.

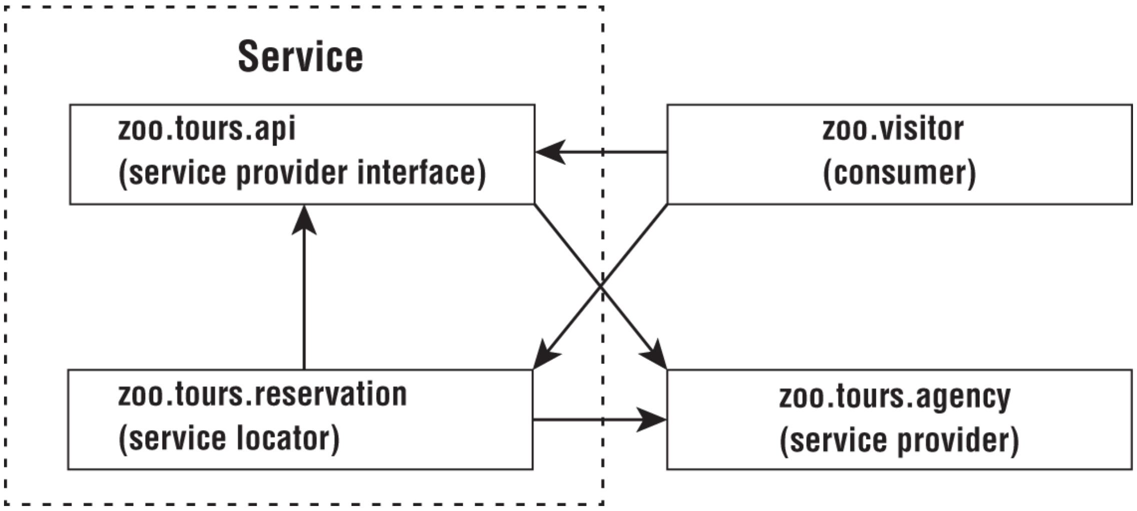


FIGURE 17.10 Modules in the tour application

NOTE

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 class Souvenir {
   private String description;
```

```
    public String getDescription() {
       return description;
    }
    public void setDescription(String description) {
       this.description = 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, as shown in Chapter 12, “Java Fundamentals.” 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/ .
```

NOTE

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 will define next.

Creating a Service Locator

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

Luckily, Java provides a `ServiceLocator` 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.

TIP

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.

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() {
      Souvenir gift = new Souvenir();
      gift.setDescription("stuffed animal");
      return gift;
   }
}
```

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;
```

NOTE

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 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.

NOTE

In software development, the concept of separating out 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.

Java allows only one service provider for a service provider interface in a module. If you wanted to offer another tour, you would need to create a separate module.

Merging Service Locator and Consumer

Now that you understand what all four pieces do, let's see if you understand how to merge pieces. We can even use streams from Chapter 15, “Functional Programming,” to implement a method that gets all implementation and the length of the shortest and longest tours.

First, let's create a second service provider. Remember the service provider `TourImpl` is the implementation of the service interface `Tour`.

```
package zoo.tours.hybrid;

import zoo.tours.api.*;

public class QuickTourImpl implements Tour {
   public String name() {
      return "Short Tour";
   }
   public int length() {
      return 30;
   }
   public Souvenir getSouvenir() {
      Souvenir gift = new Souvenir();
      gift.setDescription("keychain");
      return gift;
   }
}
```

Before we introduce the lookup code, it is important to be aware of a piece of trickery. 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
}
```

Conveniently, if you call `ServiceLoader.load()`, it 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`.

Now we can create the class that merges the service locator and consumer.

```
package zoo.tours.hybrid;

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

public class TourLengthCheck {

   public static void main(String[] args) {
      OptionalInt max = ServiceLoader.load(Tour.class)
         .stream()
         .map(Provider::get)
         .mapToInt(Tour::length)
         .max();
      max.ifPresent(System.out::println);

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

As we mentioned, there is an extra method call to use `get()` to retrieve the value out of the `Provider` since we are using a `Stream`.

Now comes the fun part. What directives do you think we need in `module-info.java`? It turns out we need three.

```
module zoo.tours.hybrid {
   requires zoo.tours.api;
   provides zoo.tours.api.Tour with zoo.tours.hybrid.QuickTourImpl;
   uses zoo.tours.api.Tour;
}
```

We need `requires` because we depend on the service provider interface. We still need `provides` so the `ServiceLocator` can look up the service. Additionally, we still need `uses` since we are looking up the service interface from another module.

For the last time, let's compile, package, and run.

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

java -p mods -m zoo.tours.hybrid/zoo.tours.hybrid.TourLengthCheck
```

And it works. The output sees both service providers and prints different values for the maximum and minimum tour lengths:

```
120
30
```

Reviewing Services

Table 17.8 summarizes what we've covered in the section about services. We recommend learning what is needed when each artifact is in a separate module really well. That is most likely what you will see on the exam and will ensure you understand the concepts.

TABLE 17.8 Reviewing services

Artifact	Part of the service	Directives required in `module-info.java`
Service provider interface	Yes	`exports`
Service provider	No	`requires` `provides`

Service locator	Yes	`exports`
		`requires`
		`uses`

Consumer	No	`requires`

Summary

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 `java.base` module is most common and is automatically supplied to all modules as a dependency. You do have to be familiar with the full list of modules provided in the JDK. The `jdeps` command provides a list of dependencies that a JAR needs. It can do so on a summary level or detailed level. Additionally, it can specify information about JDK internal modules and suggest replacements.

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 you do not have any cyclic dependencies since the Java Platform Module System will not allow cyclic dependencies to compile.

A service consists of the service provider interface and service locator. The service provider interface is the API for the service. One or more modules contain the service provider. These modules contain the implementing classes of the service provider interface. The service locator calls `ServiceLoader` to dynamically get any service providers. It can return the results so you can loop through them or get a stream. Finally, the consumer calls the service provider interface.

Exam Essentials

Identify the three types of modules. Named modules are JARs that have been modularized. Unnamed modules have not been modularized. Automatic modules are in between. They are on the module path but do not have a `module-info.java` file.

List built-in JDK modules. The `java.base` module is available to all modules. There are about 20 other modules provided by the JDK that begin with `java.*` and about 30 that begin with `jdk.*`.

Use *jdeps* to list required packages and internal packages. The `-s` flag gives a summary by only including module names. The `--jdk-internals` (`-jdkinternals`) flag provides additional information about unsupported APIs and suggests replacements.

Explain top-down and bottom-up migration. A top-down migration places all JARs on the module path, making them automatic modules while migrating from top to bottom. A bottom-up migration leaves all

JARs on the classpath, making them unnamed modules while migrating from bottom to top.

Differentiate the four main parts of a service. A service provider interface declares the interface that a service must implement. The service locator looks up the service, and a consumer calls the service. Finally, a service provider implements the service.

Code directives for use with services. A service provider implementation must have the `provides` directive to specify what service provider interface it supplies and what class it implements it `with`. The module containing the service locator must have the `uses` directive to specify which service provider implementation it will be looking up.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which of the following pairs make up a service?
   A. Consumer and service locator
   B. Consumer and service provider interface
   C. Service locator and service provider
   D. Service locator and service provider interface
   E. Service provider and service provider interface
2. A(n) ______________ module is on the classpath while a(n) ______________ module is on the module path. (Choose all that apply.)
   A. automatic, named
   B. automatic, unnamed
   C. named, automatic
   D. named, unnamed
   E. unnamed, automatic
   F. unnamed, named
   G. None of the above

3. An automatic module name is generated if one is not supplied. Which of the following JAR filename and generated automatic module name pairs are correct? (Choose all that apply.)
   A. `emily-1.0.0.jar` and `emily`
   B. `emily-1.0.0-SNAPSHOT.jar` and `emily`
   C. `emily_the_cat-1.0.0.jar` and `emily_the_cat`
   D. `emily_the_cat-1.0.0.jar` and `emily-the-cat`
   E. `emily.$.jar` and `emily`
   F. `emily.$.jar` and `emily.`
   G. `emily.$.jar` and `emily..`
4. Which of the following statements are true? (Choose all that apply.)
   A. Modules with cyclic dependencies will not compile.
   B. Packages with a cyclic dependency will not compile.
   C. A cyclic dependency always involves exactly two modules.
   D. A cyclic dependency always involves three or more modules.
   E. A cyclic dependency always involves at least two `requires` statements.
   F. An unnamed module can be involved in a cyclic dependency with an automatic module
5. Which module is available to your named module without needing a `requires` directive?
   A. `java.all`
   B. `java.base`
   C. `java.default`
   D. `java.lang`
   E. None of the above
6. Suppose you are creating a service provider that contains the following class. Which line of code needs to be in your `module-info.java`?

```
package dragon;
import magic.*;
public class Dragon implements Magic {
   public String getPower() {
      return "breathe fire";
      }
   }
```

A. `provides dragon.Dragon by magic.Magic;`
B. `provides dragon.Dragon using magic.Magic;`
C. `provides dragon.Dragon with magic.Magic;`
D. `provides magic.Magic by dragon.Dragon;`
E. `provides magic.Magic using dragon.Dragon;`
F. `provides magic.Magic with dragon.Dragon;`

7. Which of the following modules is provided by the JDK? (Choose all that apply.)
   A. `java.base;`
   B. `java.desktop;`
   C. `java.logging;`
   D. `java.util;`
   E. `jdk.base;`
   F. `jdk.compiler;`
   G. `jdk.xerces;`

8. Which of the following compiles and is equivalent to this loop?

```
List<Unicorn> all  = new ArrayList<>();
for (Unicorn current : ServiceLoader.load(Unicorn.class))
   all.add(current);
```

A.

```
List<Unicorn> all = ServiceLoader.load(Unicorn.class)
   .getStream()
   .collect(Collectors.toList());
```

B.

```
List<Unicorn> all = ServiceLoader.load(Unicorn.class)
  .stream()
```

```
         .collect(Collectors.toList());
```

C.

```
      List<Unicorn> all = ServiceLoader.load(Unicorn.class)
        .getStream()
        .map(Provider::get)
        .collect(Collectors.toList());
```

D.

```
      List<Unicorn> all = ServiceLoader.load(Unicorn.class)
        .stream()
        .map(Provider::get)
        .collect(Collectors.toList());
```

E. None of the above

9. Which command can you run to determine whether you have any code in your JAR file that depends on unsupported internal APIs and suggests an alternative?
   A. `jdeps -internal-jdk`
   B. `jdeps --internaljdk`
   C. `jdeps --internal-jdk`
   D. `jdeps -s`
   E. `jdeps -unsupported`
   F. `jdeps –unsupportedapi`
   G. `jdeps –unsupported-api`
   H. None of the above
10. For a top-down migration, all modules other than named modules are ____________ modules and on the ____________.
   A. automatic, classpath
   B. automatic, module path
   C. unnamed, classpath
   D. unnamed, module path

E. None of the above

11. Suppose you have separate modules for a service provider interface, service provider, service locator, and consumer. If you add a second service provider module, how many of these modules do you need to recompile?
    A. Zero
    B. One
    C. Two
    D. Three
    E. Four
12. Which of the following modules contains the `java.util` package? (Choose all that apply.)
    A. `java.all;`
    B. `java.base;`
    C. `java.main;`
    D. `java.util;`
    E. None of the above
13. Suppose you have separate modules for a service provider interface, service provider, service locator, and consumer. Which are true about the directives you need to specify? (Choose all that apply.)
    A. The service provider interface must use the `exports` directive.
    B. The service provider interface must use the `provides` directive.
    C. The service provider interface must use the `requires` directive.
    D. The service provider must use the `exports` directive.
    E. The service provider must use the `provides` directive.
    F. The service provider must use the `requires` directive.
14. Suppose you have a project with one package named `magic.wand` and another project with one package named `magic.potion`. These projects have a circular dependency, so you decide to create a third project named `magic.helper`. The `magic.helper` module has the common code containing a package named `magic.util`. For simplicity, let's give each module the same name as the package. Which of the following need to appear in your `module-info` files? (Choose all that apply.)
    A. `exports magic.potion;` in the potion project

   B. `exports magic.util;` in the magic helper project
   C. `exports magic.wand;` in the wand project
   D. `requires magic.util;` in the magic helper project
   E. `requires magic.util;` in the potion project
   F. `requires magic.util;` in the wand project

15. Suppose you have separate modules for a service provider interface, service provider, service locator, and consumer. Which module(s) need to specify a `requires` directive on the service provider?
   A. Service locator
   B. Service provider interface
   C. Consumer
   D. Consumer and service locator
   E. Consumer and service provider
   F. Service locator and service provider interface
   G. Consumer, service locator, and service provider interface
   H. None of the above
16. Which are true statements about a package in a JAR on the classpath containing a `module-info` file? (Choose all that apply.)
   A. It is possible to make it available to all other modules on the classpath.
   B. It is possible to make it available to all other modules on the module path.
   C. It is possible to make it available to exactly one other specific module on the classpath.
   D. It is possible to make it available to exactly one other specific module on the module path.
   E. It is possible to make sure it is not available to any other modules on the classpath.
17. Which are true statements? (Choose all that apply.)
   A. An automatic module exports all packages to named modules.
   B. An automatic module exports only the specified packages to named modules.
   C. An automatic module exports no packages to named modules.
   D. An unnamed module exports only the named packages to named modules.

E. An unnamed module exports all packages to named modules.
F. An unnamed module exports no packages to named modules.

18. Suppose you have separate modules for a service provider interface, service provider, service locator, and consumer. Which statements are true about the directives you need to specify? (Choose all that apply.)
    A. The consumer must use the `requires` directive.
    B. The consumer must use the `uses` directive.
    C. The service locator must use the `requires` directive.
    D. The service locator must use the `uses` directive.
19. Which statement is true about the `jdeps` command? (Choose all that apply.)
    A. It can provide information about dependencies on the class level only.
    B. It can provide information about dependencies on the package level only.
    C. It can provide information about dependencies on the class or package level.
    D. It can run only against a named module.
    E. It can run against a regular JAR.
20. Suppose we have a JAR file named `cat-1.2.3-RC1.jar` and `Automatic-Module-Name` in the `MANIFEST.MF` is set to `dog`. What should an unnamed module referencing this automatic module include in the `module-info.java`?
    A. `requires cat;`
    B. `requires cat.RC;`
    C. `requires cat-RC;`
    D. `requires dog;`
    E. None of the above