Chapter 17 - Modular Applications

Reviewing Module Directives

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

CLASSPATH VS. MODULE PATH

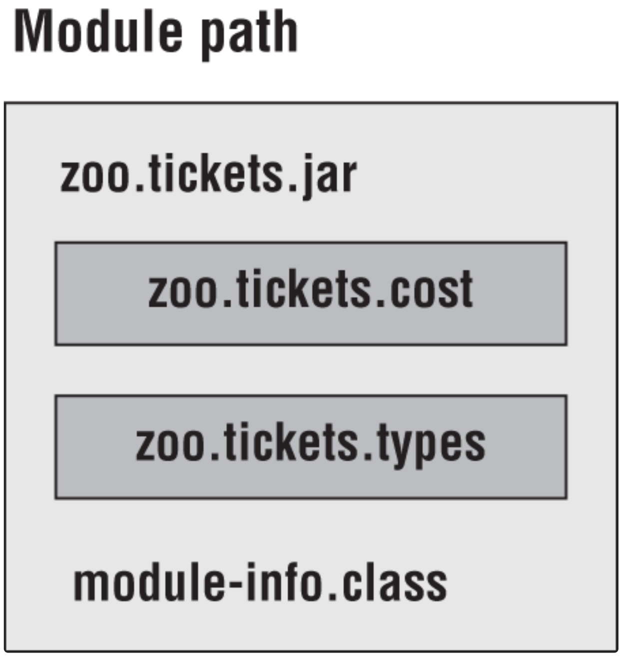
Before we get started, a brief reminder that the Java runtime is capable of using class and interface types from both the classpath and the module path, although the rules for each are a bit different. An application can access any type in the classpath that is exposed via standard Java access modifiers,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

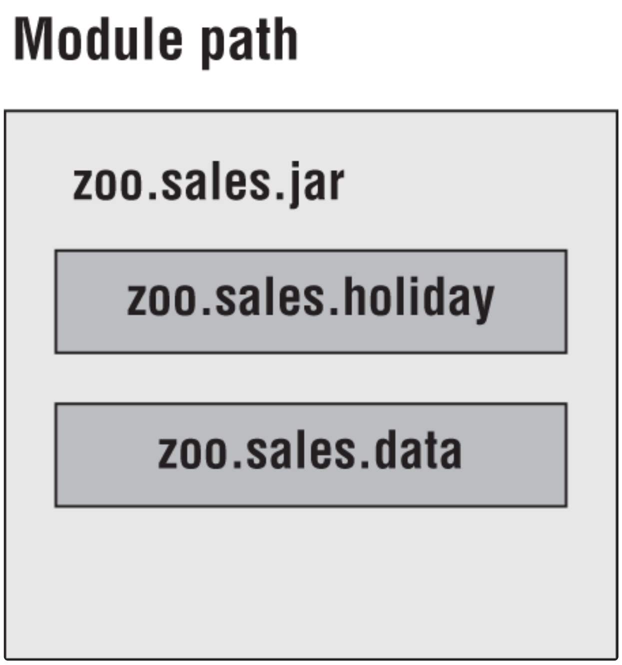
As a way of remembering this, a named module has the name inside the `module-info` file and is on the module path. Figure 17.1 shows the contents of a JAR file for a named module. It contains two packages in addition to the `module-info.class`.



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.



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.

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.

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 (`.`).

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

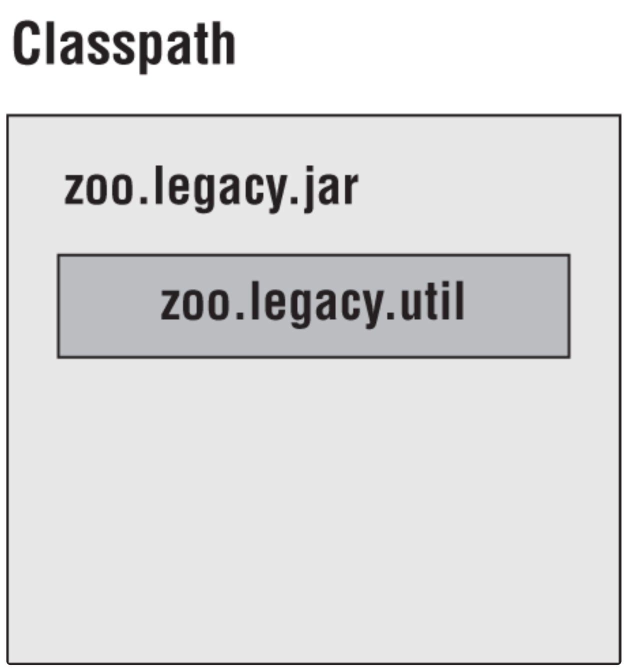
- If the `MANIFEST.MF` specifies an `Automatic-Module-Name`, use that. Otherwise, proceed with the remaining rules.
- Remove the file extension from the JAR name.
- Remove any version information from the end of the name. A version is digits and dots with possible extra information at the end, for example, `-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 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`

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.



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.

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

Identifying Built-in Modules

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.

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

`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`	

`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`	

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.

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.

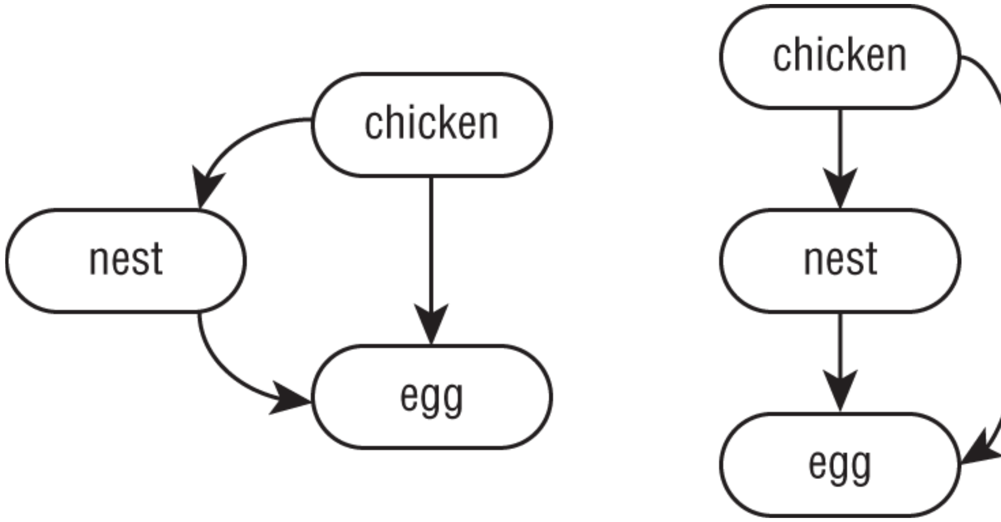
```
jdeps -s zoo.dino.jar

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

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
```

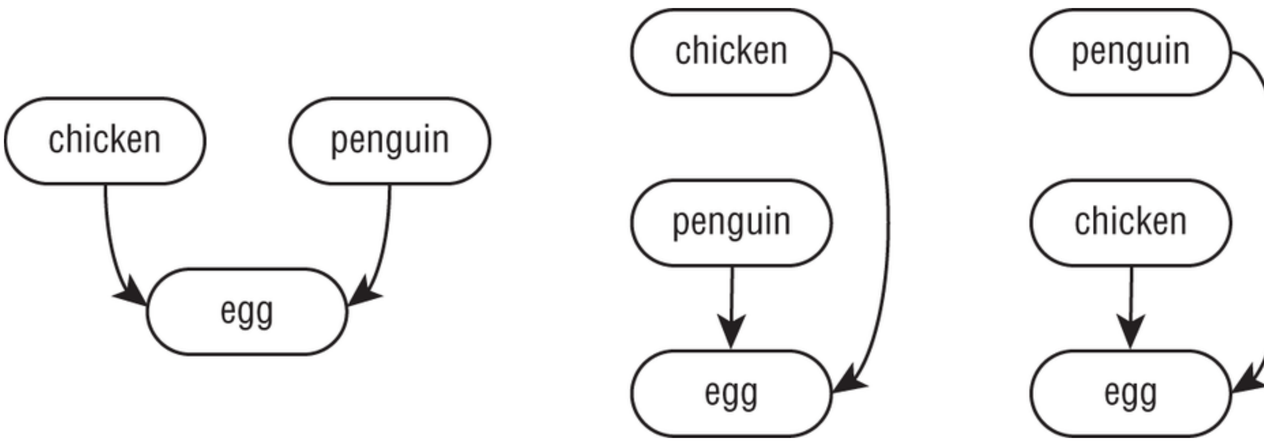
Migrating an Application

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.

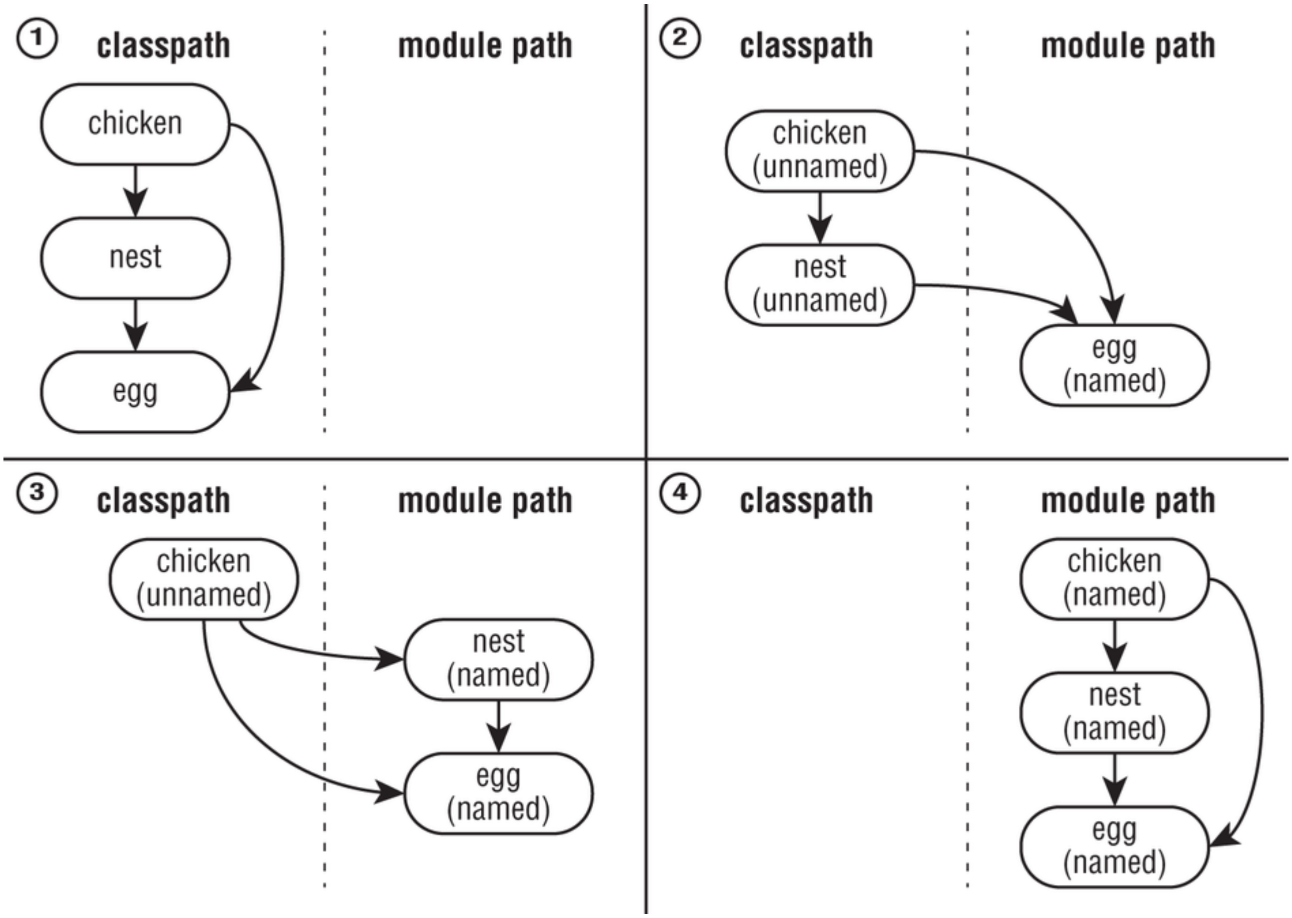
In this example, there is only one order from top to bottom that honors all the dependencies. Figure 17.5 shows that the order is not always unique. Since two of the projects do not have an arrow between them, either order is allowed when deciding migration order.



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.



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.

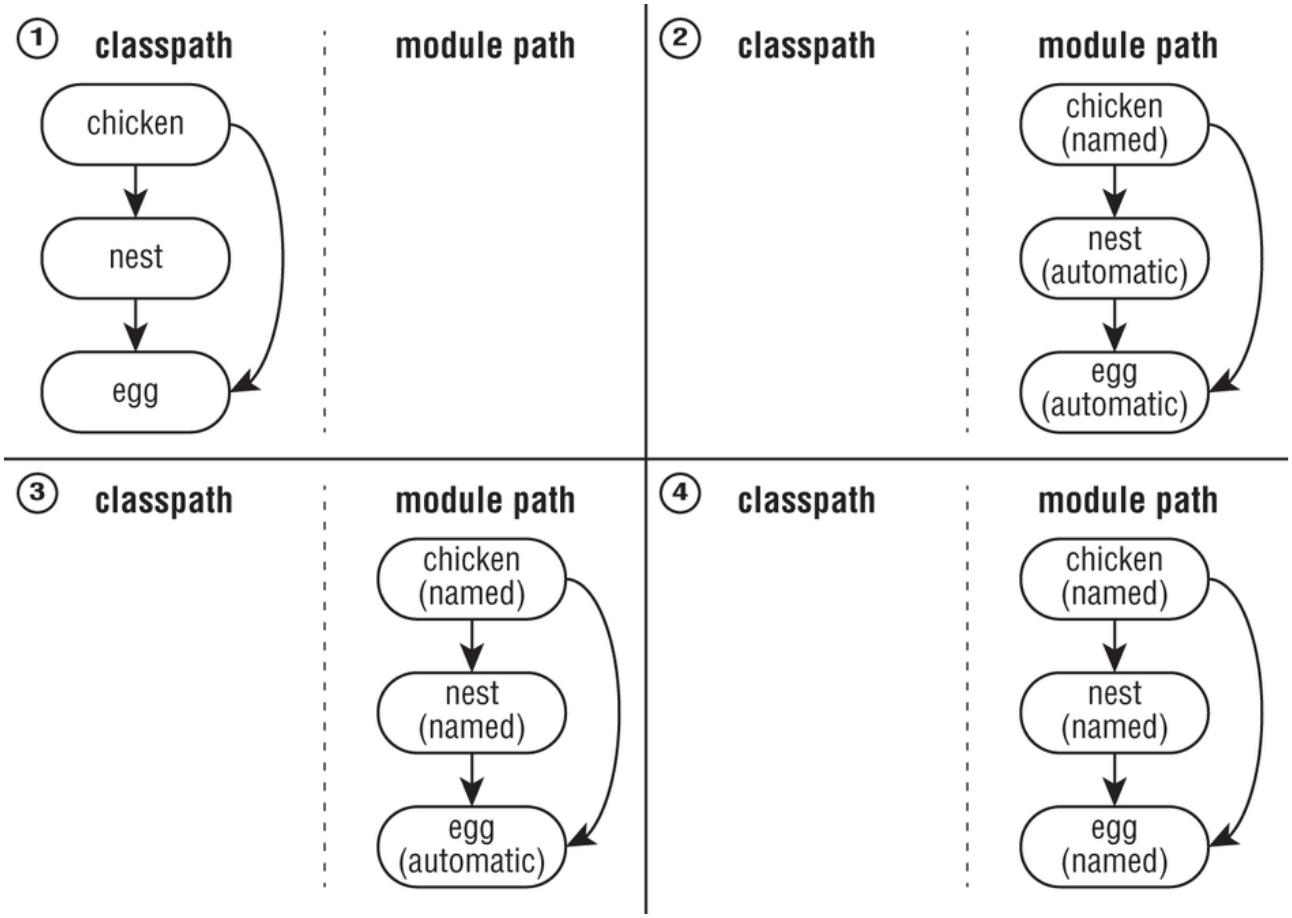
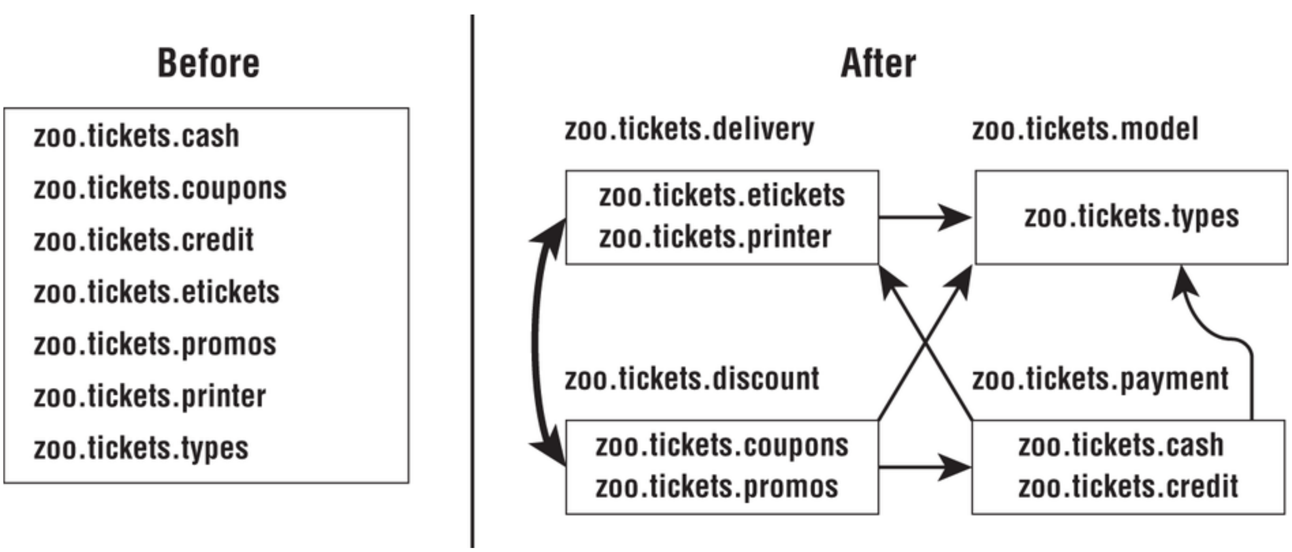


TABLE 17.7 Comparing migration strategies

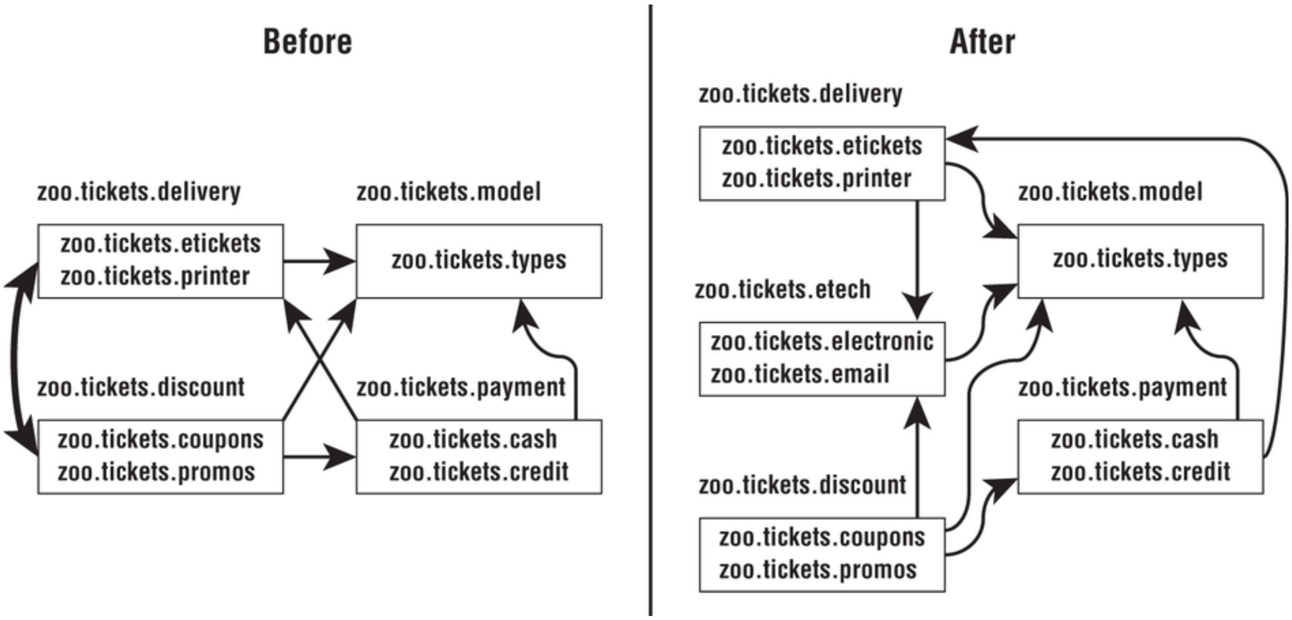
Category	Bottom-Up	Top-Down
A project that depends on all others	Unnamed module on the classpath	Named module on the module path
A project that has no dependencies	Named module on the module path	Automatic module on the module path

Splitting a Big Project into Modules



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!



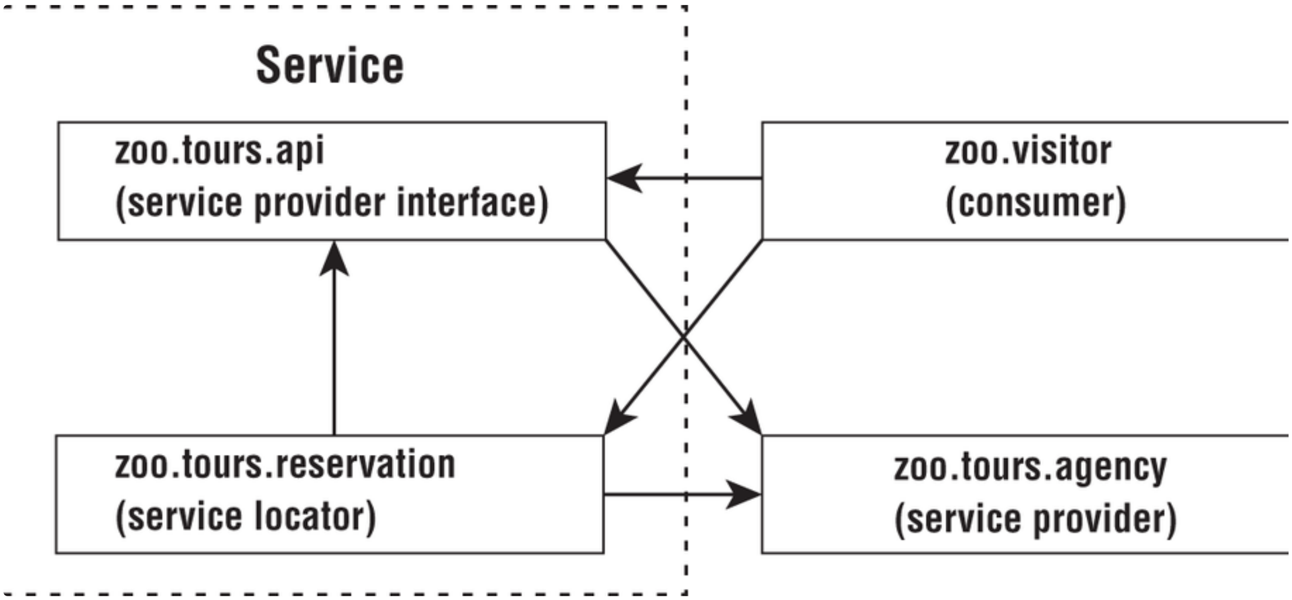
Failing to Compile with a Cyclic Dependency

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.

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.



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.

Declaring the Service Provider Interface

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;
}
```

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 `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;
   }
}
```

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;
}
```

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;
}
```

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;
}
```

`provides  interfaceName with className;`

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.

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`.

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`