Capitolul 12 - Modules
Introducing Modules

JAR

Is a ZIP file with some extra information, with extension `.jar`.

Java Platform Module System

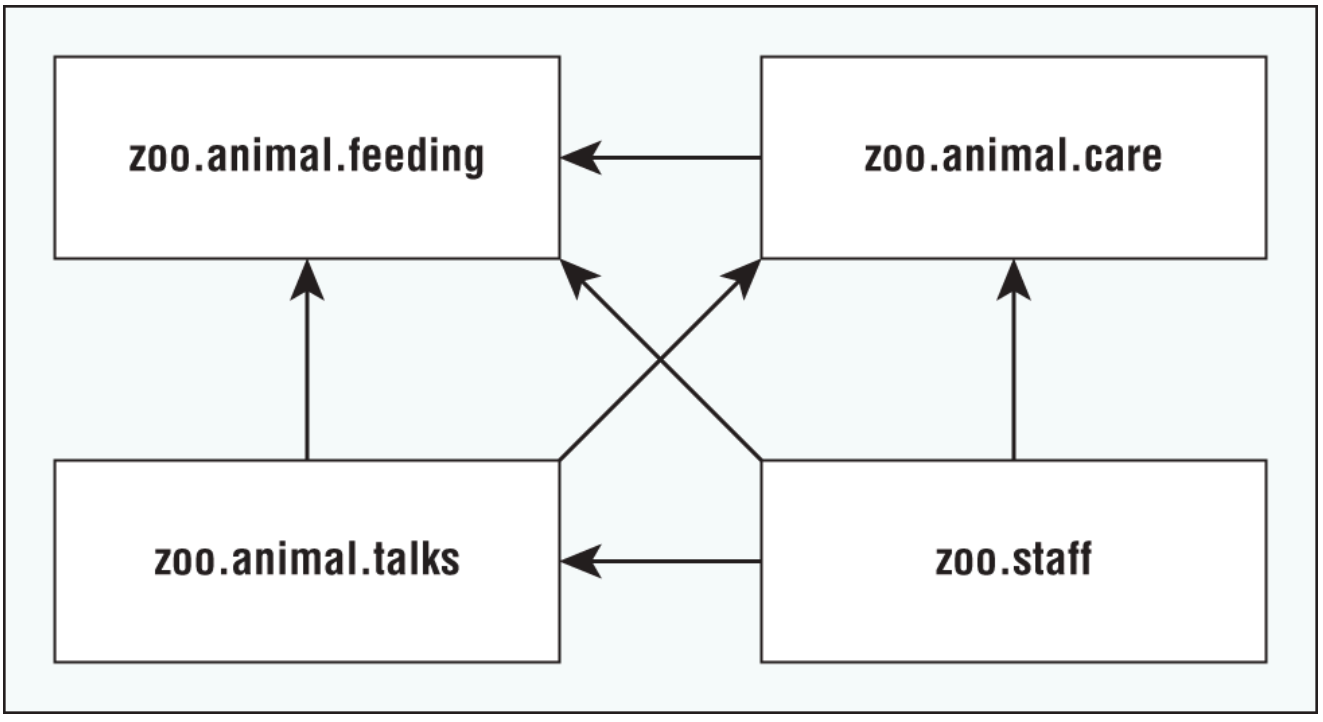
Groups code at a higher level.
The main purpose of a module is to provide groups of related packages that offer developers a particular set of functionality.

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

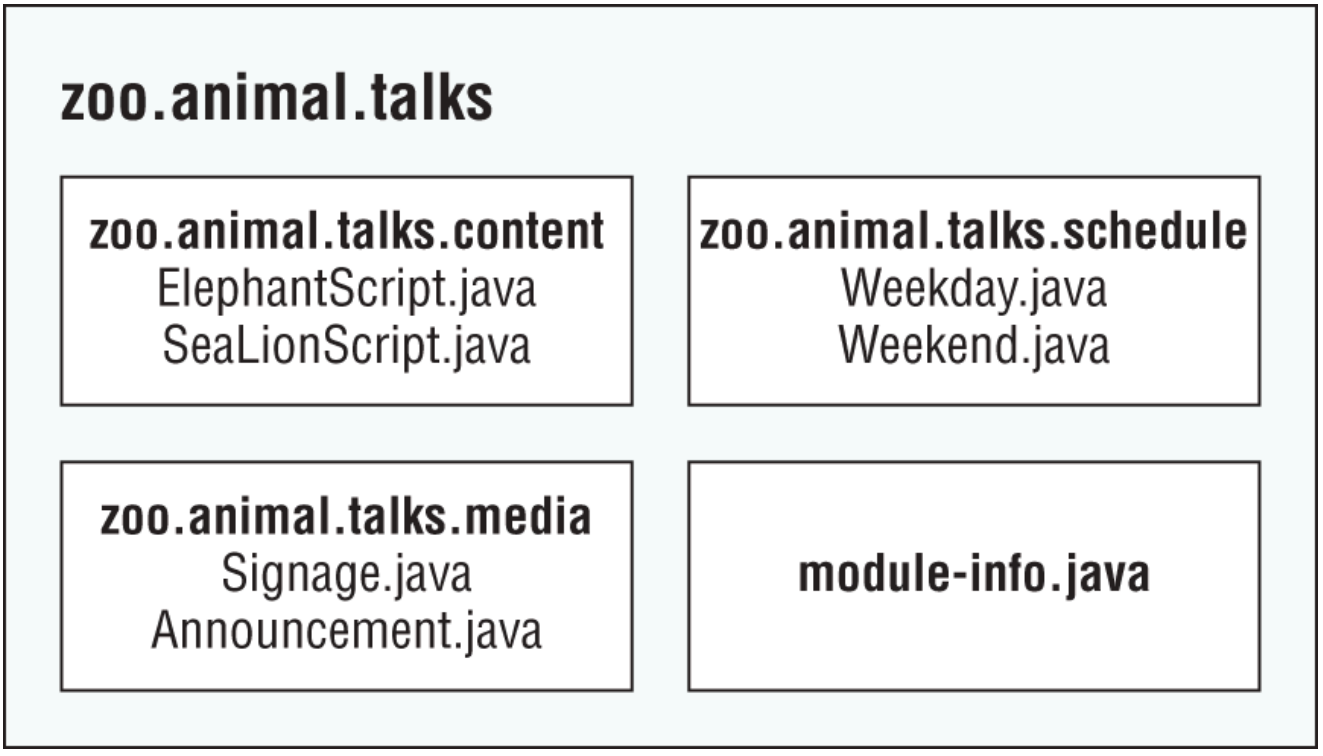
Module

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*.



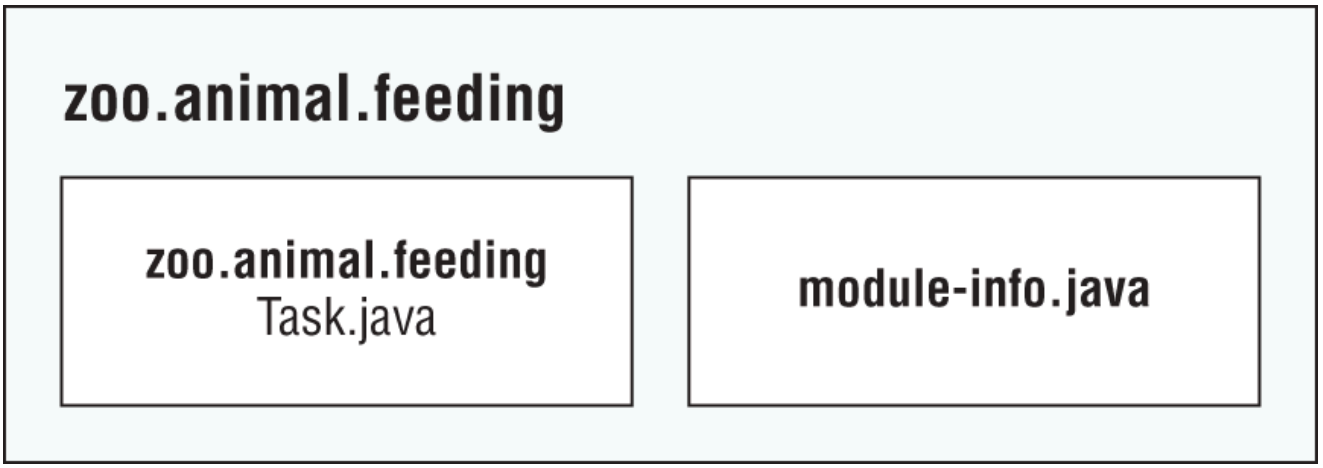
Benefits of Modules

1. **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.
2. **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.
3. **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.
4. **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.
5. **Improved performance:** Another benefit of a smaller Java package is improved startup time and a lower memory requirement.
6. **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

We'll using an example:



First, we need some classes. For this module is ok only `Task.java`.

```
package zoo.animal.feeding

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

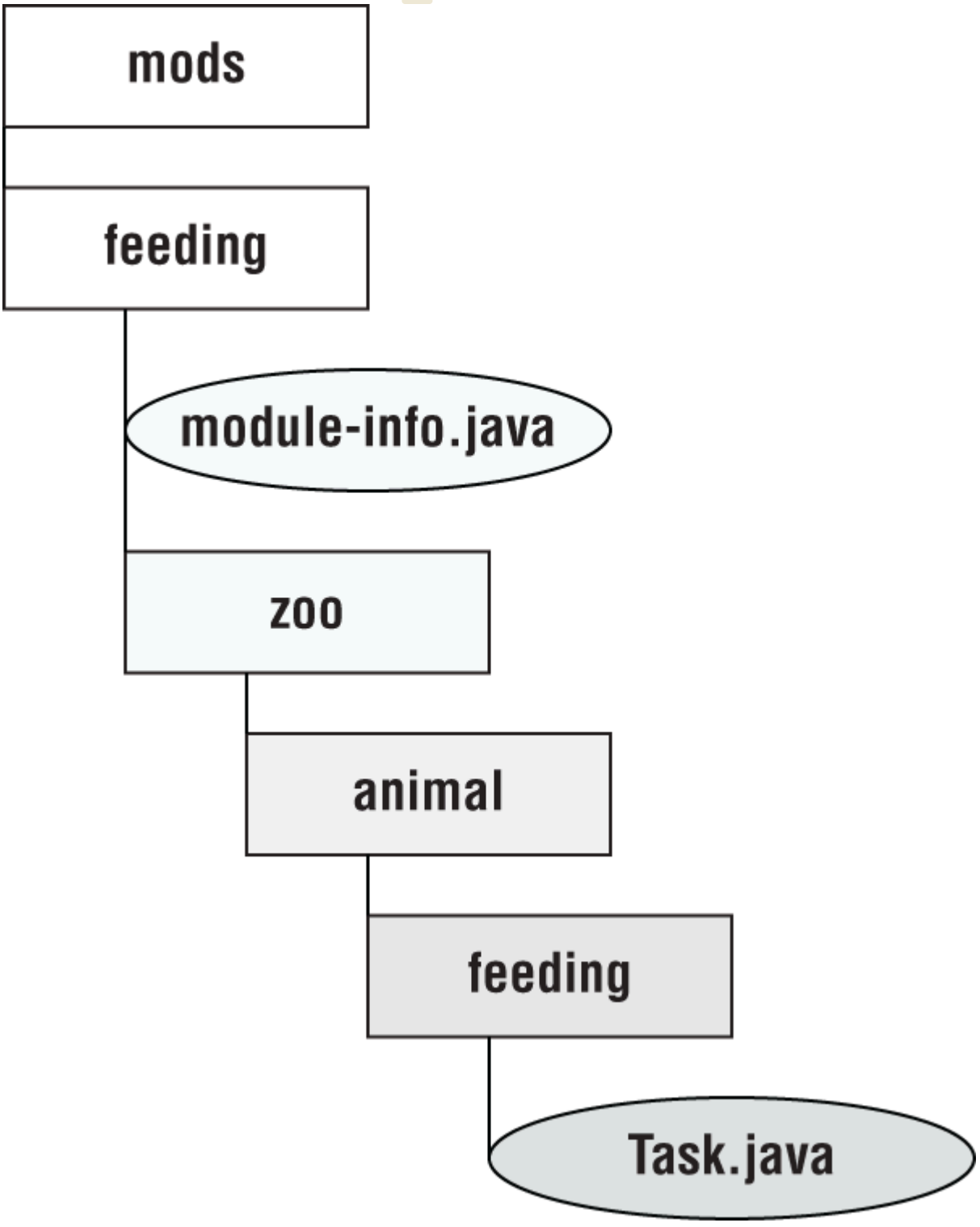
And, the most important, `module-info.java`

```
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.

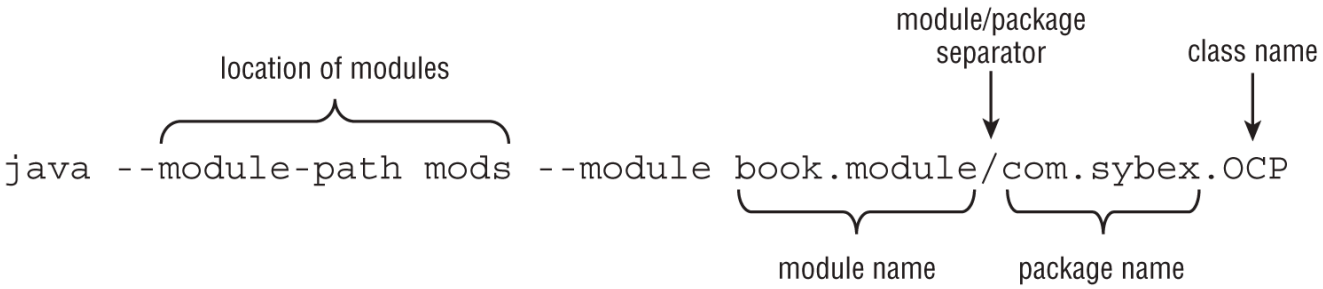


This is directory structure.

Compiling

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

Running Our First Module



Packaging Our First Module

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

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
```

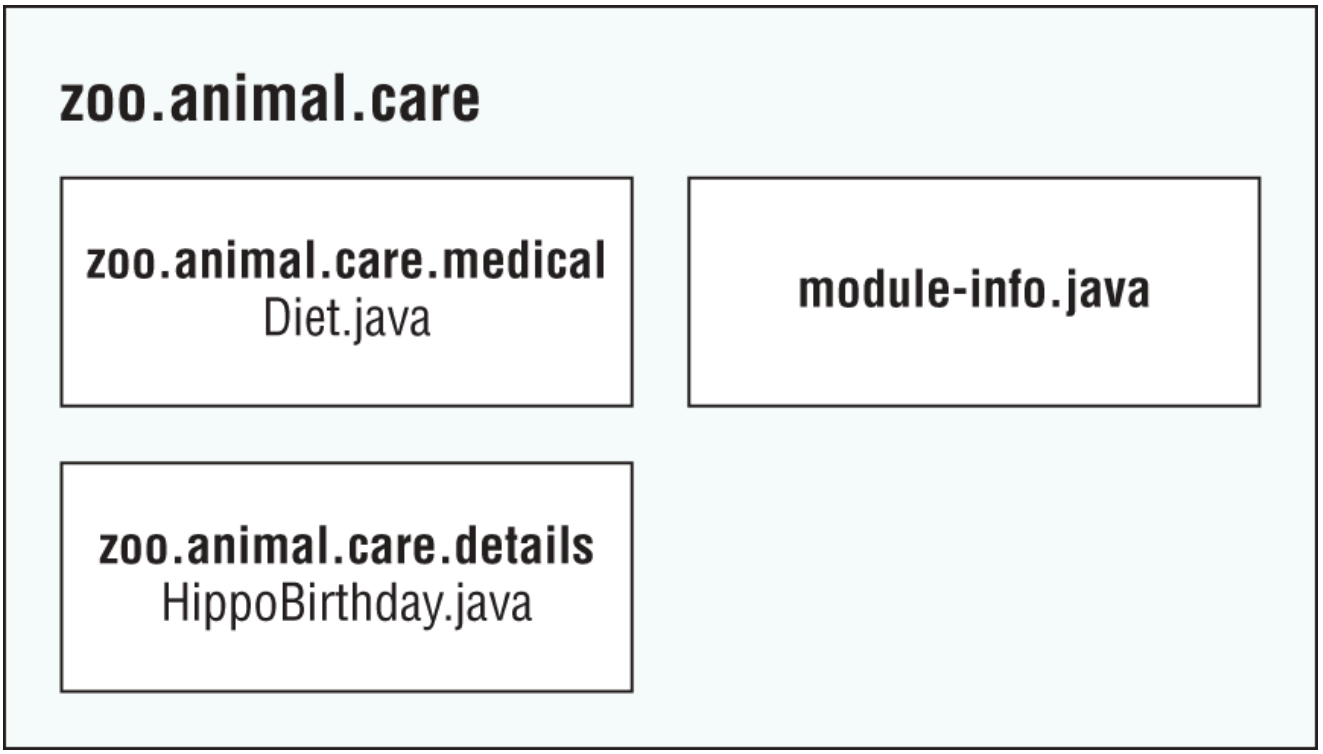
Updating Our Example for Multiple Modules

Updating the Feeding Module

> 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;
}
```

Creating a Care Module




```
// 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 { //specifies the name of module
2:    exports zoo.animal.care.medical; // list the package we are
exporting
3:    requires zoo.animal.feeding; // specifies that a module is
needed | zoo.animal.care - depends —> zoo.anima.feeding
4: }
```

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.

Diving into the Module Declaration

Exporting a Package

```
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;
}
```

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

> There's also a `requires transitive` *`moduleName`*, which means that any module that `requires` this module will also depend on `moduleName`.

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

```
}

---
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;
}
---
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

Applying the `transitive` modifier 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.care` module.

> 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

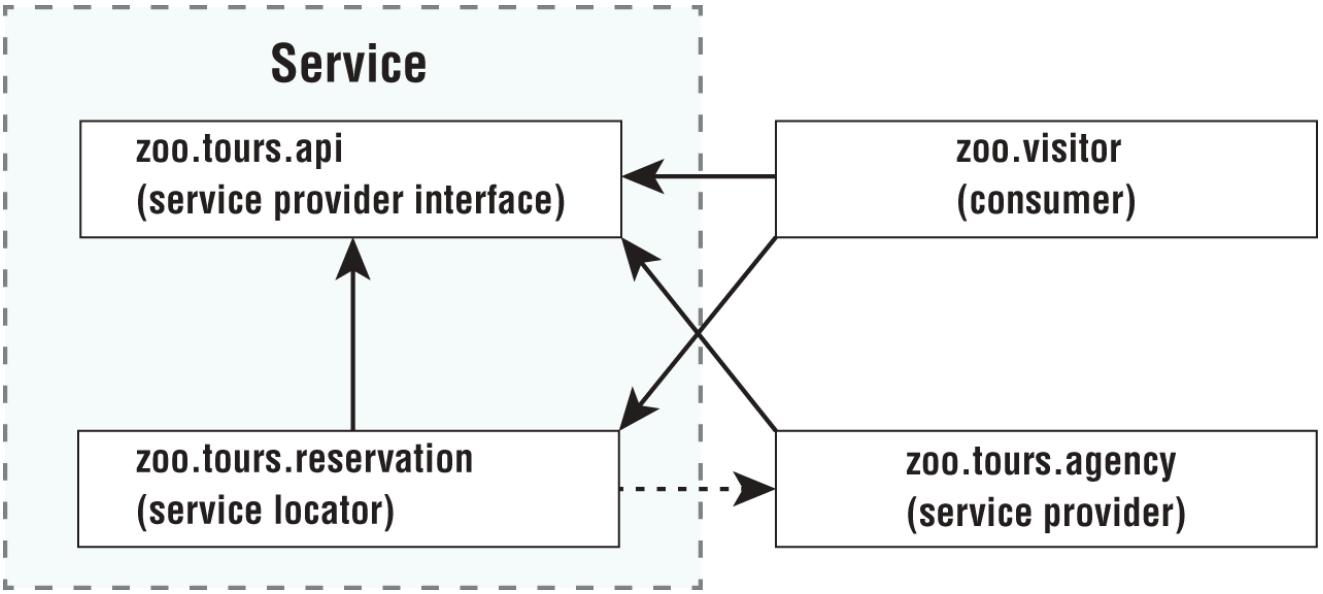
> The `opens` directive is used to enable reflection of a package within a module. You only need to be aware that the `opens` directive exists rather than understanding it in detail for the exam.

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

Creating 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 understand, we'll use this:



Declaring the Service Provider Interface

First, we'll define in `zoo.tours.api` a Java object named `Souvenir`

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

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

Now, we'll create an interface (*service provider interface*).

```
// 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;
}
```

Creating a Service Locator

> A *service locator* can find any classes that implement a service provider interface.

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.

```
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;
```

```
    }
}
```

```
// 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.

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) { …
} //returns an object that you can loop through normally.
    public Stream<Provider<S>> stream() { … }
    // Additional methods
}
```

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

> 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());
   }
}
```

```
// 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.

```
// 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");
    }
}
```

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

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;
```

Reviewing Directives and Services

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

Identifying Built-in Modules

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	Not on exam beyond module name
java.xml	Extensible Markup Language (XML)	Not on exam beyond module name

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 Java modules prefixed with jdk

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

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

//describe a module

java -p mods
   --describe-module zoo.animal.feeding
----
java --list-modules //list the modules that are available
----
java --show-module-resolution
   -p feeding
   -m zoo.animal.feeding/zoo.animal.feeding.Task
// 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.
----
```

Learning About Dependencies with *jdeps*

> The `jdeps` command gives you information about dependencies within a module.

```
// 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();
   }
}
```

```
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)

-----
```

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

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

-----

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
```

Using Module Files with *jmod*

> 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.

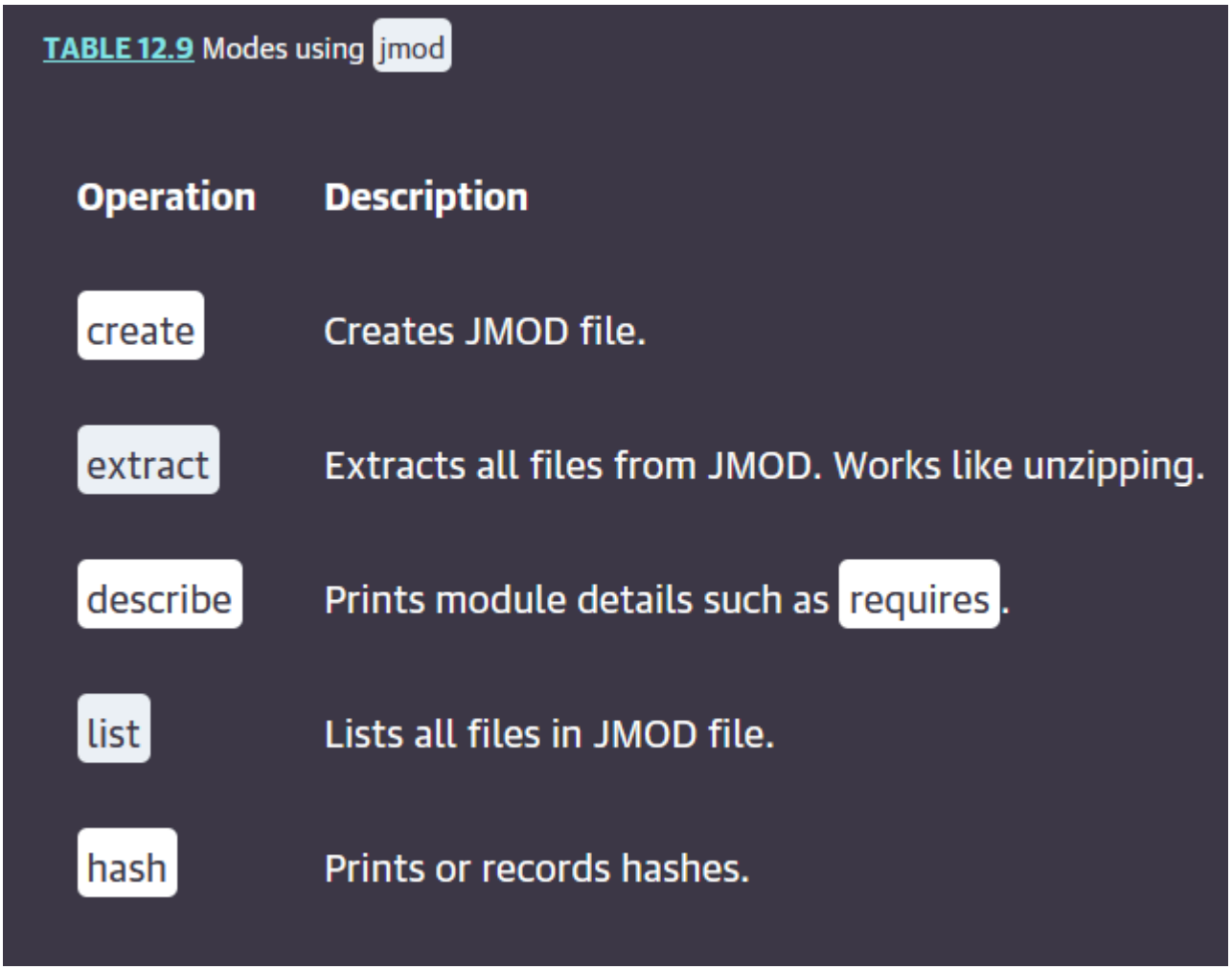


TABLE 12.9 Modes using jmod

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 Runtime Images with *jlink*

> This command creates our smaller distribution, referred to as a *runtime image*:

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

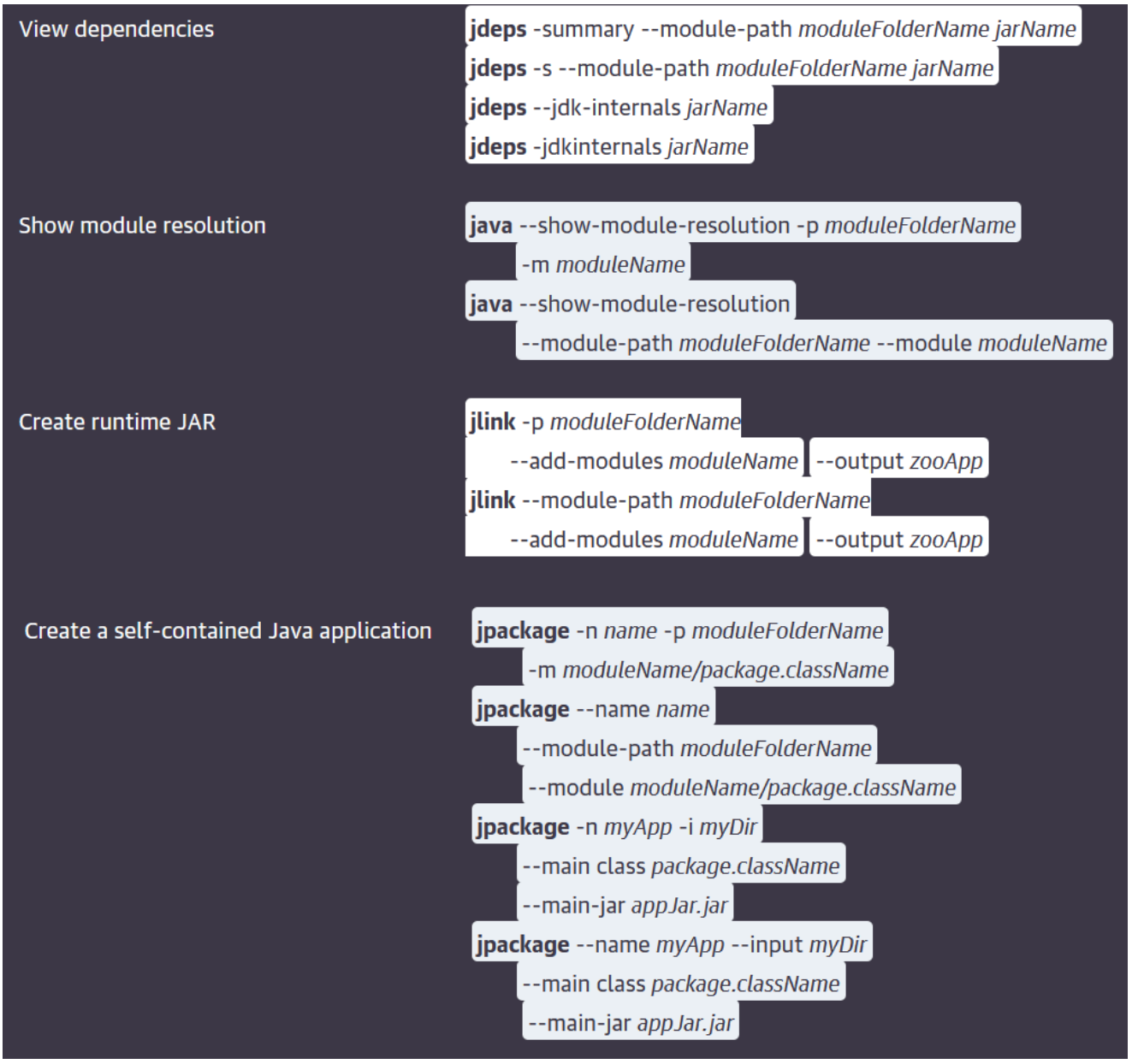
Creating Self-Contained Java Applications with *jpackage*

Unlike `jlink` which can only create a *runtime image*, the `jpackage` command can create an *application image*. An application image is a single executable file capable of running your application on a specific platform. For example, you get an `.exe` file for Windows or a `.dmg` file for Mac.

```
jpackage --name feedingTask --module-path mods --module
zoo.animal.feeding/zoo.animal.feeding.Task
```

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



Comparing Types of Modules

Named Module

Is one containing a `module-info.java` file.

Automatic Module

Appears on the module path but does not contain a `module-info.java` file.

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 12.17 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

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.

Reviewing Module Types

TABLE 12.18 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 named modules?	Those in `module-info.java` file	All packages	No packages
Which packages does a ____ module export to automatic modules?	Those in `module-info.java` file	All packages	All 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

Determining the 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 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.

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.

TABLE 12.19 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

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.