Chapter 11 - Modules

The main purpose of a module is to provide groups of related packages to offer a particular set of functionality to developers. It's like a JAR file except a developer chooses which packages are accessible outside the module. Let's look at what modules are and what problems they are designed to solve.

The Java Platform Module System includes the following:

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

A *module* is a group of one or more packages plus a special file called `module-info .java`

Benefits of Modules

Better Access Control

Modules act as a fifth level of access control. They can expose packages within the modular JAR to specific other packages. This stronger form of encapsulation really does create internal packages.

Clearer Dependency Management

In a fully modular environment, each of the open source projects would specify their dependencies in the `module-info.java` file. When launching the program, Java would complain that Hamcrest isn't in the module path and you'd know right away.

Custom Java Builds

The Java Platform Module System allows developers to specify what modules they actually need. This makes it possible to create a smaller runtime image that is customized to what the application needs and nothing more. Users can run that image without having Java installed at all.

Improved Performance

Since Java now knows which modules are required, it only needs to look at those at class loading time. This improves startup time for big programs and requires less memory to run.

Unique Package Enforcement

A package is only allowed to be supplied by one module. No more unpleasant surprises about a package at runtime.

There are a few key differences between a `module-info` file and a regular Java class:

- The `module-info` file must be in the root directory of your module. Regular Java classes should be in packages.
- The `module-info` file 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.

Compiling Our First Module

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

```
javac --module-path mods
   -d feeding
```

```
   feeding/zoo/animal/feeding/*.java
   feeding/module-info.java
```

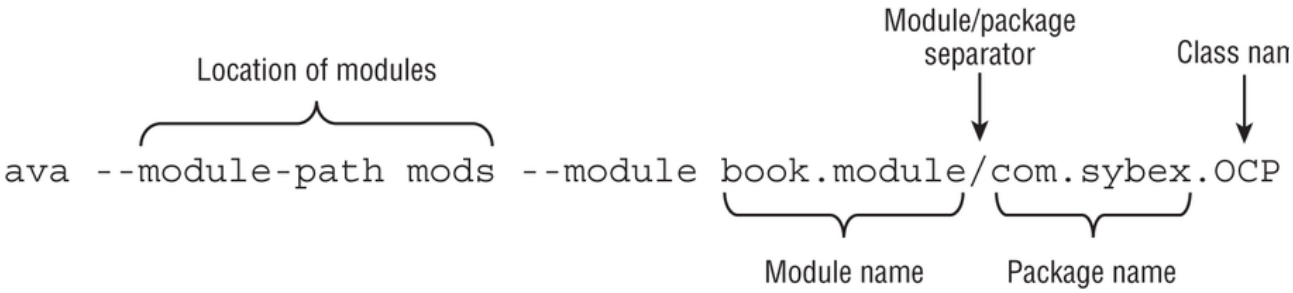
As a review, the `-d` option specifies the directory to place the class files in. The end of the command is a list of the `.java` files to compile. You can list the files individually or use a wildcard for all `.java` files in a subdirectory.

The new part is the `module-path`. This option indicates the location of any custom module files. In this example, `module-path` could have been omitted since there are no dependencies. You can think of `module-path` as replacing the `classpath` option when you are working on a modular program. The syntax `--module-path` and `-p` are equivalent. That means we could have written many other commands in place of the previous command. The following four commands show the `-p` option:

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

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

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

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

Running Our First Module

Before we package our module, we should make sure it works by running it. To do that, we need to learn the full syntax. Suppose there is a module named `book.module`. Inside that module is a package named `com.sybex`, which has a class named `OCP` with a `main()` method. Figure 11.5 shows the syntax for running a module. Pay special attention to the `book.module/com.sybex.OCP` part. It is important to remember that you specify the module name followed by a slash (`/`) followed by the fully qualified class name.



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

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

short form of `--module` is `-m`

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

Packaging Our First Module

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

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

Updating the Feeding Module

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

The `exports` keyword 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` keyword, 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;
}
```

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

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

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

Note that order matters when compiling a module. Suppose we list the `module-info` file first when trying to compile:

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

The compiler complains that it doesn't know anything about the package `zoo.animal.care.medical`.

Diving into the *module-info* File

Now that we've successfully created modules, we can learn more about the `module-info` file. In these sections, we will look at `exports`, `requires`, `provides`, `uses`, and `opens`. Now would be a good time to mention that these keywords can appear in any order in the `module-info` file.

exports

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

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

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

TABLE 11.3 Access control with modules

Level	Within module code	Outside module
`private`	Available only within class	No access
default (package-private)	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

requires transitive

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

Duplicate *requires* Statements

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

provides, *uses*, and *opens*

For the remaining three keywords (`provides`, `uses`, and `opens`), you only need to be aware they exist rather than understanding them in detail for the 1Z0-815 exam.

The `provides` keyword specifies that a class provides an implementation of a service. The topic of services is covered on the 1Z0-816 exam, so for now, you can just think of a service as a fancy interface. To use it, you supply the API and class name that implements the API:

```
provides zoo.staff.ZooApi with zoo.staff.ZooImpl
```

The `uses` keyword specifies that a module is relying on a service. To code it, you supply the API you want to call:

```
uses zoo.staff.ZooApi
```

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

Since reflection can be dangerous, the module system requires developers to explicitly allow reflection in the `module-info` if they want calling modules to be allowed to use it. Here are two examples:

```
opens zoo.animal.talks.schedule;
opens zoo.animal.talks.media to zoo.staff;
```

The first example allows any module using this one to use reflection. The second example only gives that privilege to the `zoo.staff` package.

Describing a Module

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

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

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

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

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

The `java.base` module is special. It is automatically added as a dependency to all modules. This module has frequently used packages like `java.util`. That's what the `mandated` is about. You get `java.base` whether you asked for it or not.

Listing Available Modules

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

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

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

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

Showing Module Resolution

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

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

The *jar* Command

Like the `java` command, the `jar` command can describe a module. Both of these commands are equivalent:

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

The *jdeps* Command

The `jdeps` command gives you information about dependencies within a module. Unlike describing a module, it looks at the code in addition to the `module-info` file. This tells you what dependencies are actually used rather than simply declared.

Let's start with a simple example and ask for a summary of the dependencies in `zoo.animal.feeding`. Both of these commands give the same output:

```
jdeps -s mods/zoo.animal.feeding.jar

jdeps -summary mods/zoo.animal.feeding.jar
```

Notice that there is one dash (`-`) before `-summary` rather than two. Regardless, the output tells you that there is only one package and it depends on the built-in `java.base` module.

```
zoo.animal.feeding -> java.base
```

Alternatively, you can call `jdeps` without the summary option and get the long form:

```
jdeps mods/zoo.animal.feeding.jar
[file:///absolutePath/mods/zoo.animal.feeding.jar]
   requires mandated java.base (@11.0.2)
zoo.animal.feeding -> java.base
   zoo.animal.feeding         -> java.io
      java.base
   zoo.animal.feeding         -> java.lang
      java.base
```

Now, let's look at a more complicated example. This time, we pick a module that depends on `zoo.animal.feeding`. We need to specify the module path so `jdeps` knows where to find information about the dependent module. We didn't need to do that before because all dependent modules were built into the JDK.

Following convention, these two commands are equivalent:

```
jdeps -s
   --module-path mods
   mods/zoo.animal.care.jar
```

```
4
5 jdeps -summary
6    --module-path mods
7    mods/zoo.animal.care.jar
```

There is not a short form of `--module-path` in the `jdeps` command.

The *jmod* Command

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

TABLE 11.5 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 a module	`javac -p moduleFolderName -d directory classesToCompileIncludingModuleInfo` `javac --module-path moduleFolderName -d directory classesToCompileIncludingModuleInfo`
Run a module	`java -p moduleFolderName -m moduleName/package.className` `java --module-path moduleFolderName --module moduleName/package.className`

Describe a module	`java -p moduleFolderName -d moduleName`
	`java --module-path moduleFolderName --describe-module moduleName`
	`jar --file jarName --describe-module`
	`jar -f jarName -d`

List available modules	`java --module-path moduleFolderName --list-modules`
	`java -p moduleFolderName --list-modules`
	`java --list-modules`

View dependencies	`jdeps -summary --module-path moduleFolderName jarName` `jdeps -s --module-path moduleFolderName jarName`
Show module resolution	`java --show-module-resolution -p moduleFolderName -m moduleName` `java --show-module-resolution --module-path moduleFolderName --module moduleName`

TABLE 11.6 Options you need to know for the exam: `javac`

Option	Description

`-cp <classpath>`
`-classpath <classpath>`
`--class-path <classpath>`
	Location of JARs in a nonmodular program

`-d <dir>`	Directory to place generated class files
`-p <path>` `--module-path <path>`	Location of JARs in a modular program

TABLE 11.7 Options you need to know for the exam: `java`

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

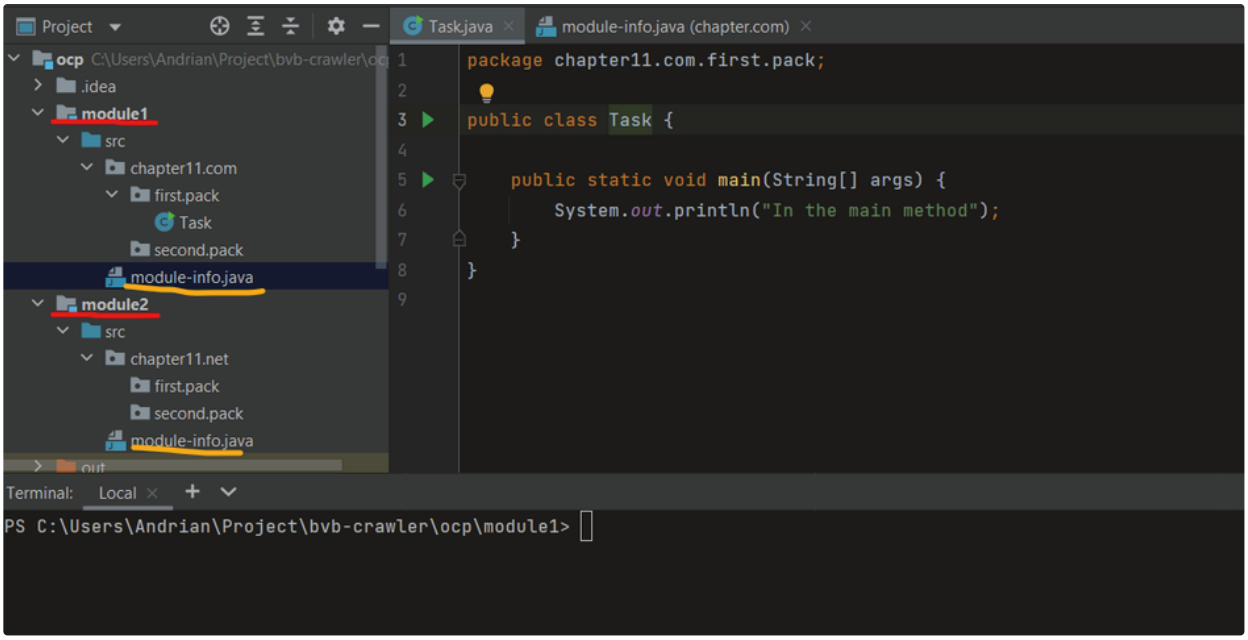
TABLE 11.8 Options you need to know for the exam: `jar`

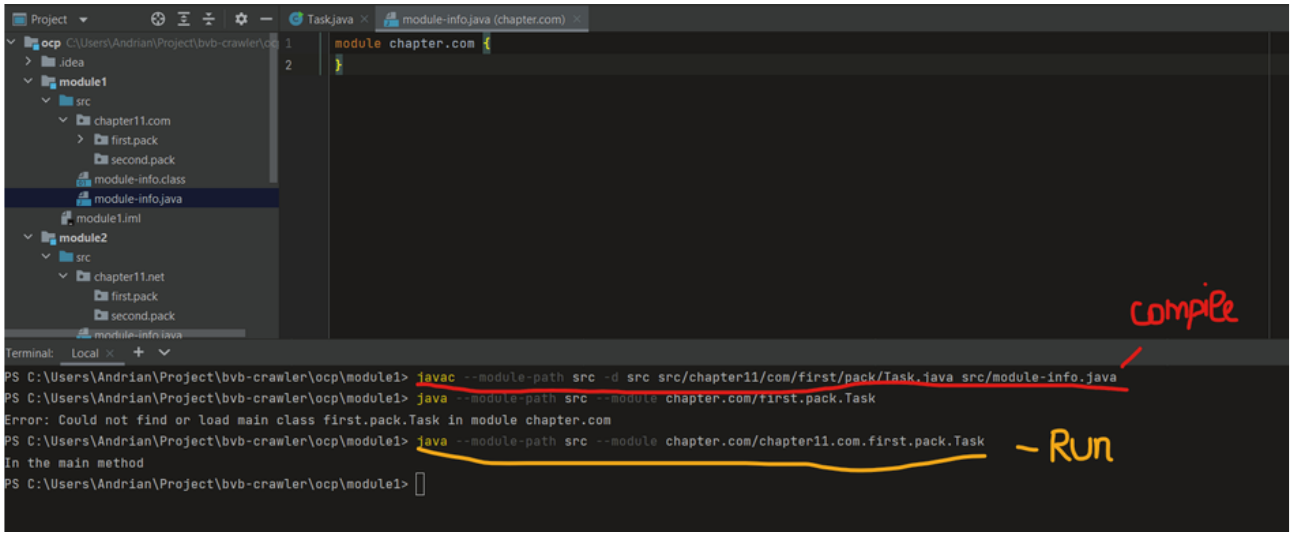
Option	Description
`-c` `--create`	Create a new JAR file
`-v` `--verbose`	Prints details when working with JAR files
`-f` `--file`	JAR filename
`-C`	Directory containing files to be used to create the JAR
`-d` `--describe-module`	Describes the details of a module

TABLE 11.9 Options you need to know for the exam: `jdeps`

Option	Description
`--module-path <path>`	Location of JARs in a modular program
`-s` `-summary`	

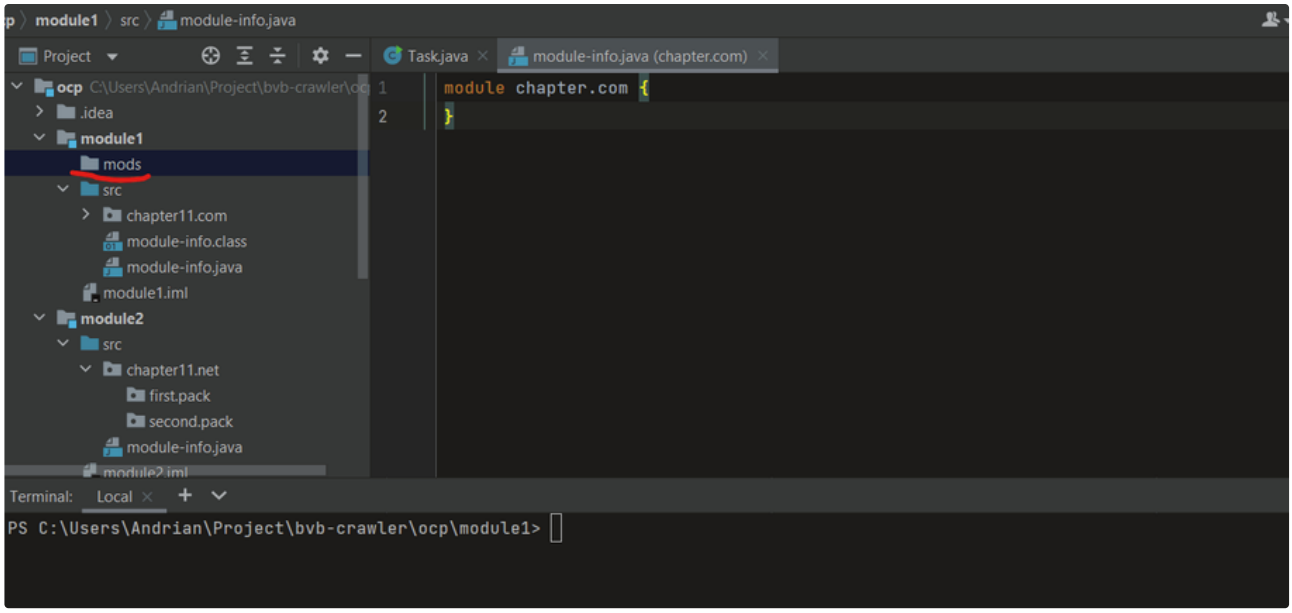
Personal example:



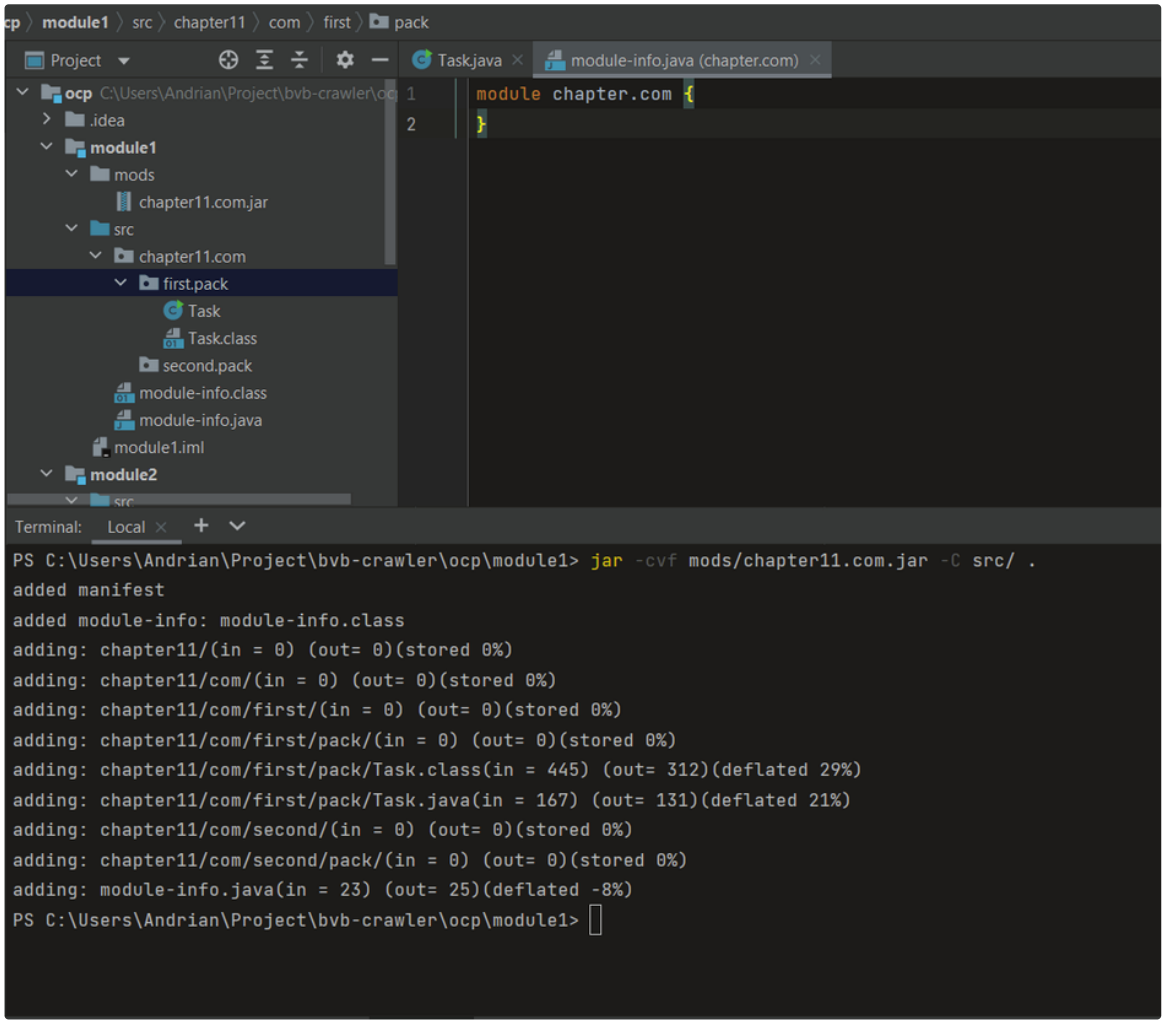


To pack everything up, create a new directory under module1

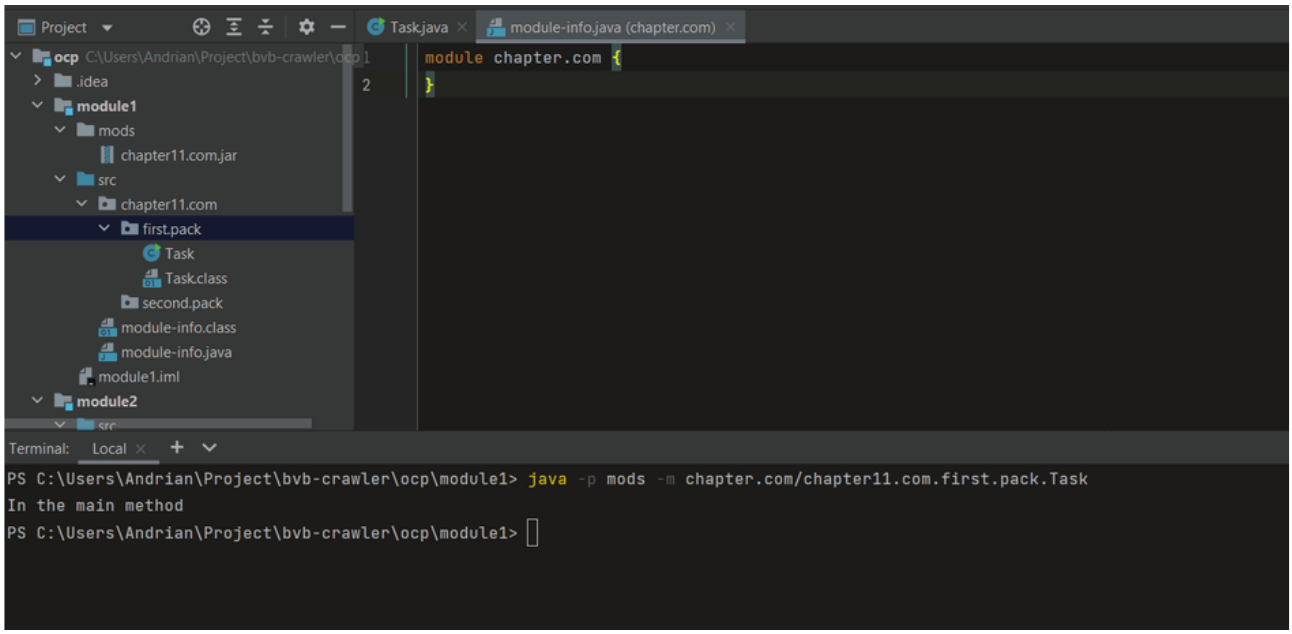
should look like this



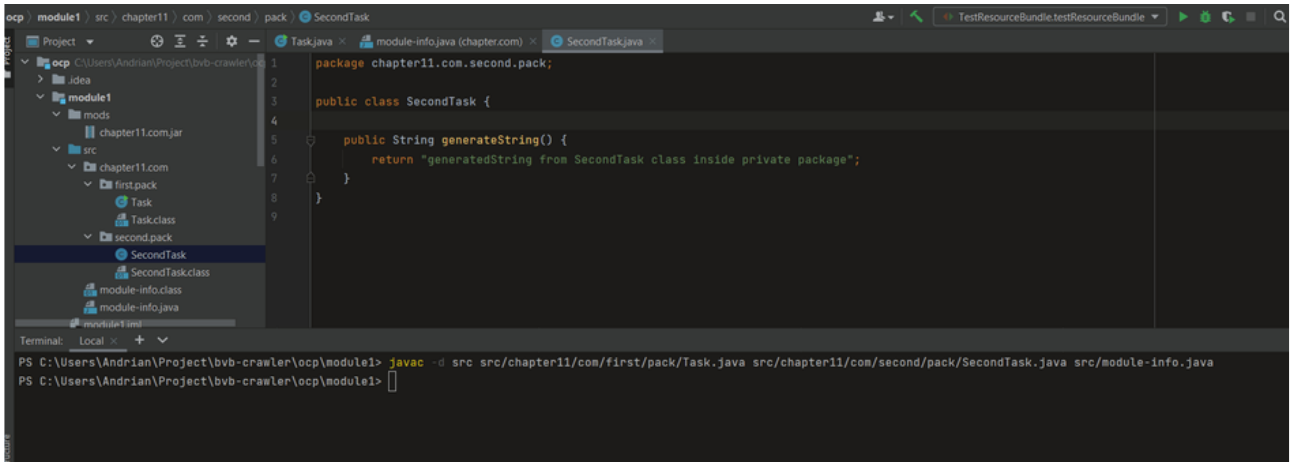
Then run the command to pack everything under a jar

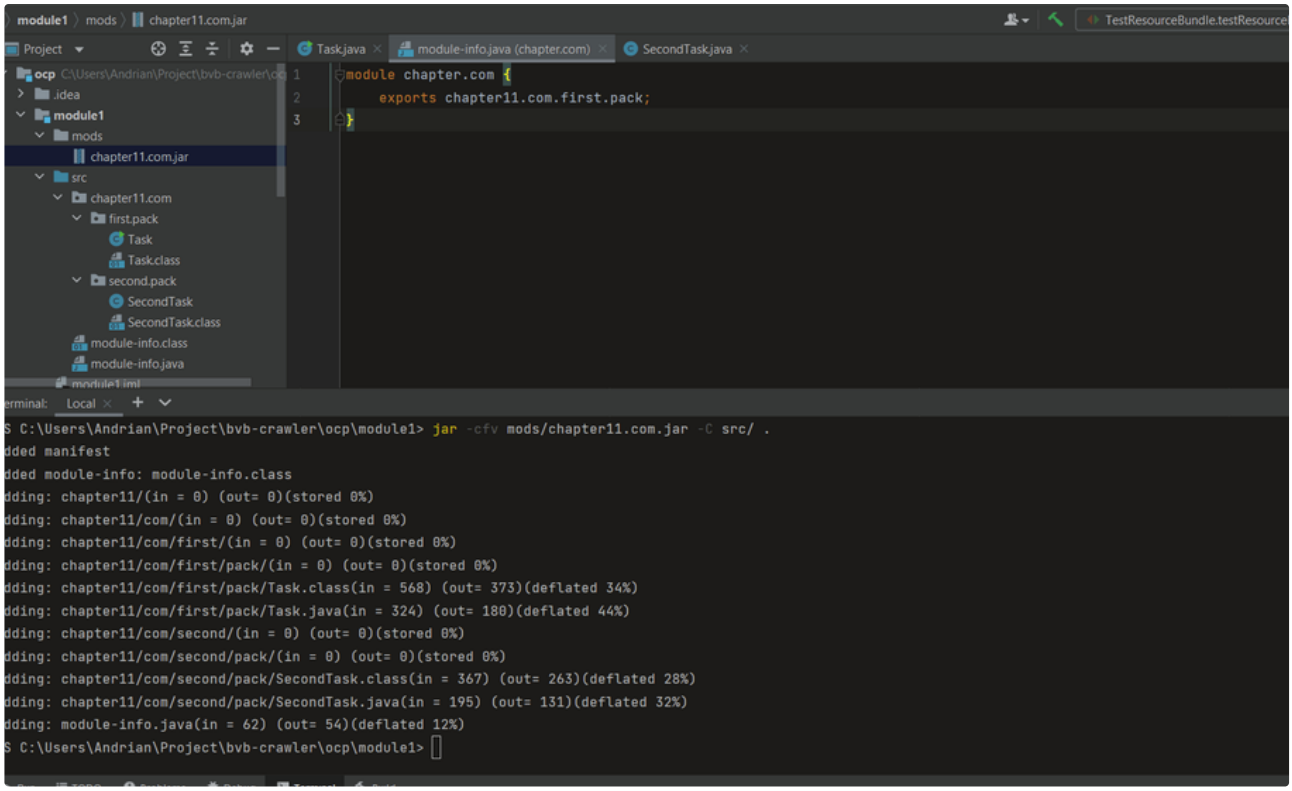


Running jar

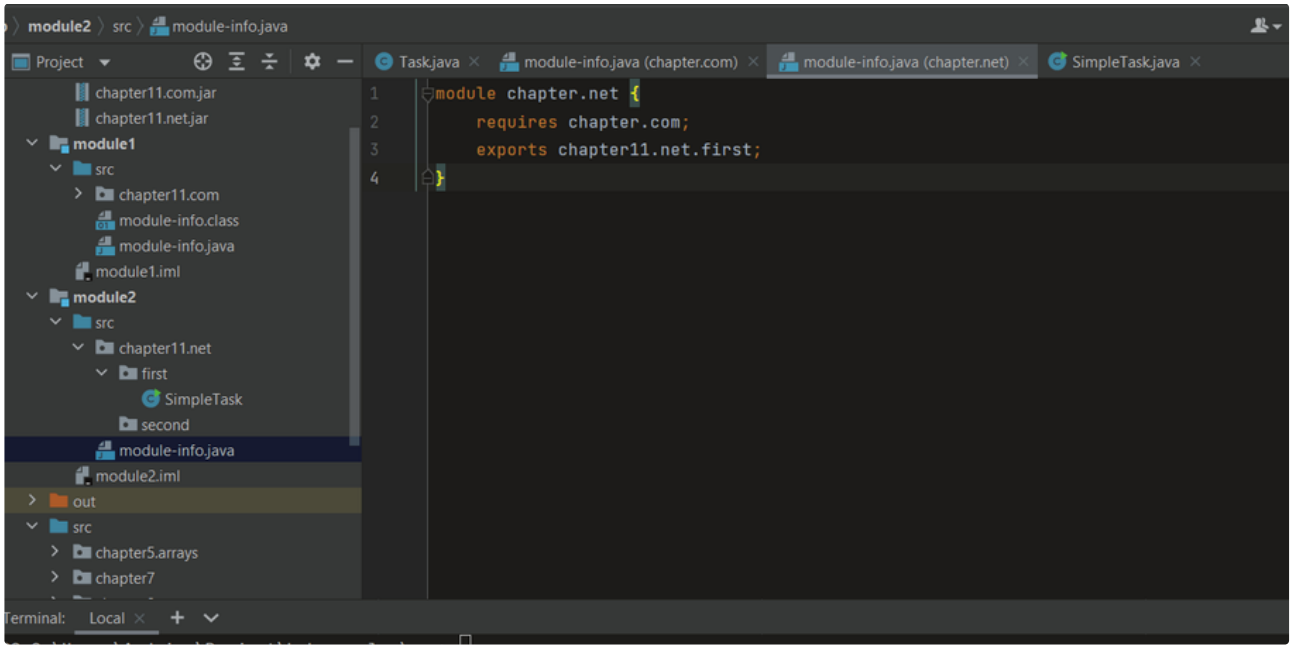


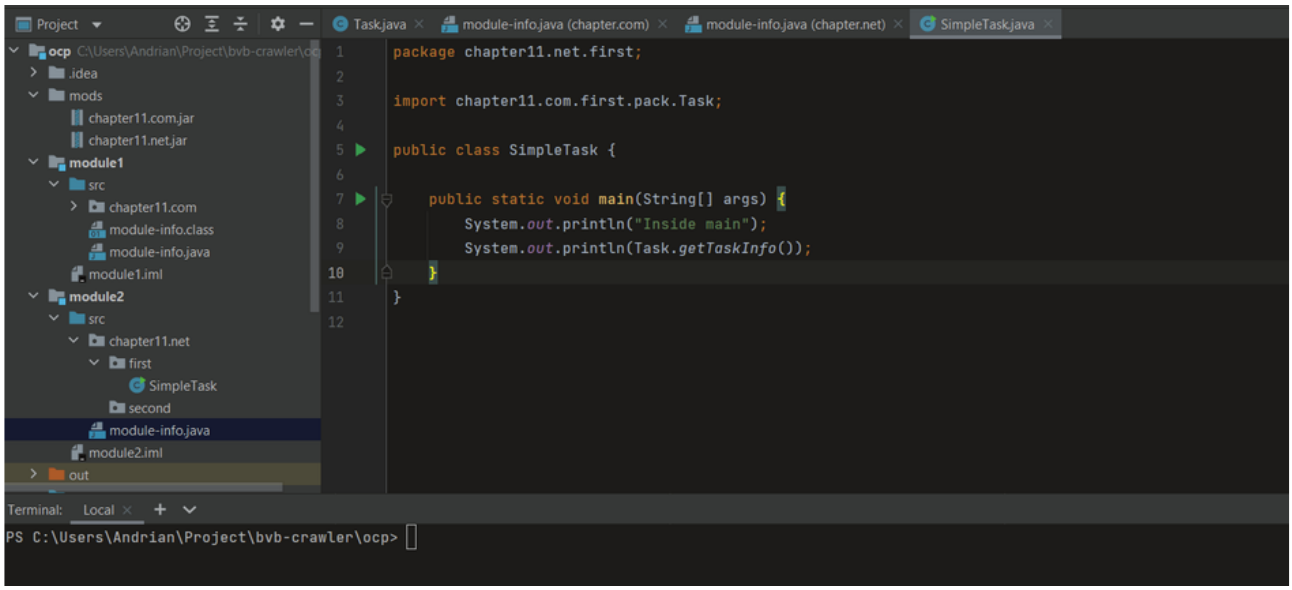
Now, let's update module-info file to export the package





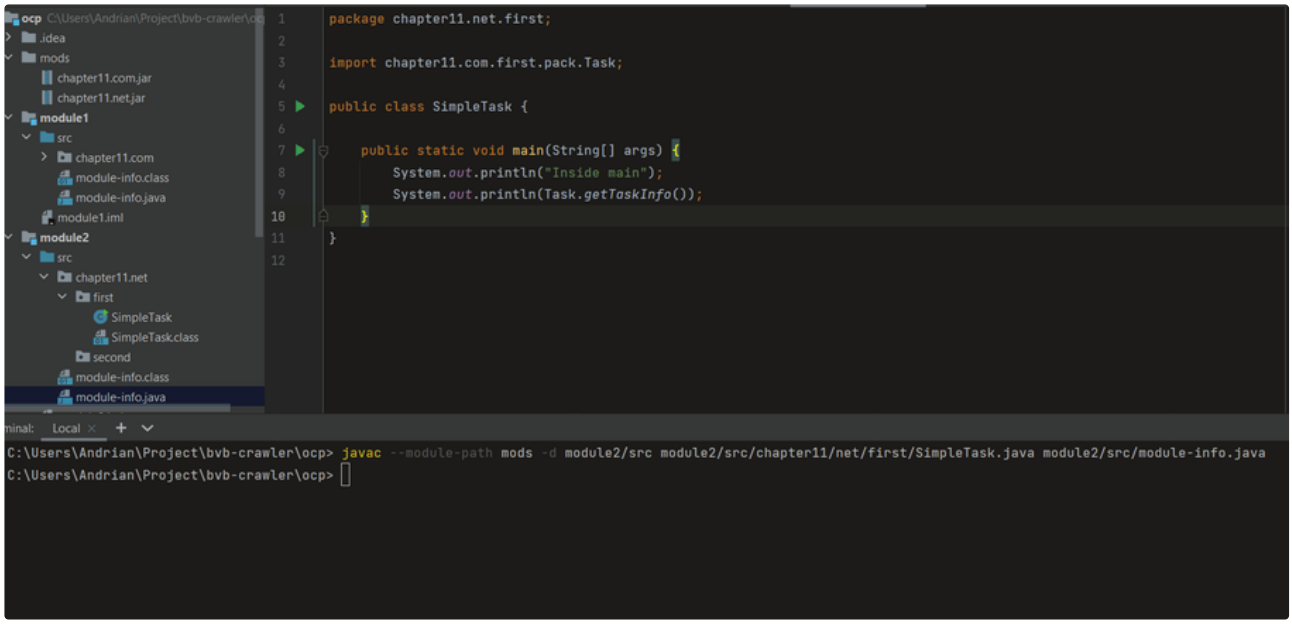
Now lets declare information in second module and compile it



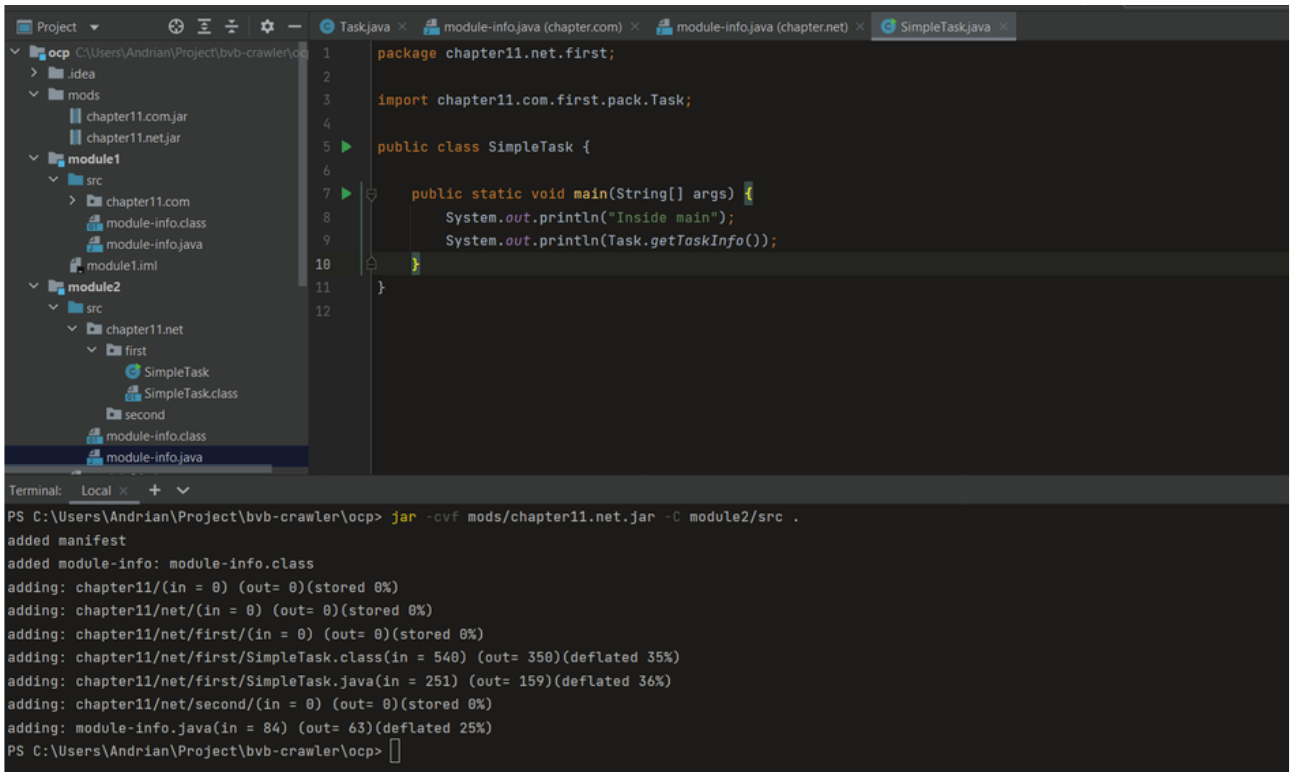


In order to have easier access to modules, we extracted mods directory outside module1. (I believe this is the right place to have it and we made a mistake in the beginning placing it in module1

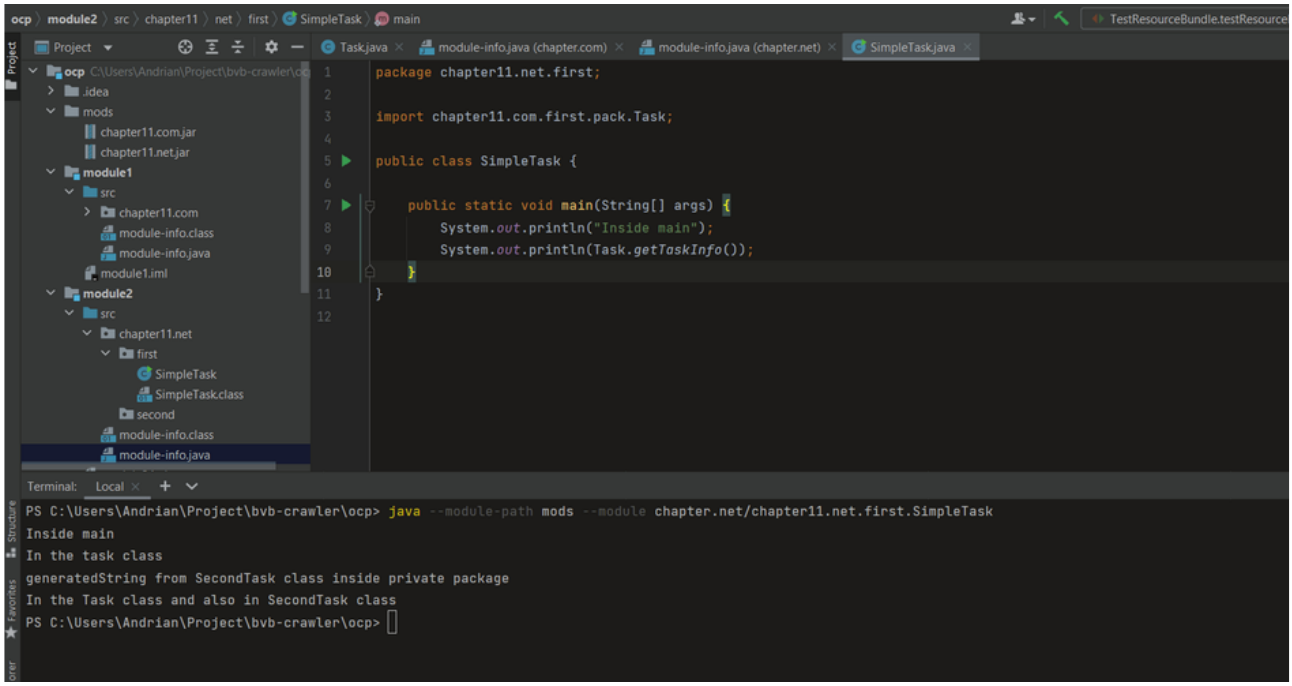
Now let's compile the module



And package it


Project
Task.java
module-info.java (chapter.com)
module-info.java (chapter.net)
SimpleTask.java
ocp C:\Users\Andrian\Project\bvb-crawler\oc
.idea
mods
chapter11.com.jar
chapter11.net.jar
module1
src
chapter11.com
module-info.class
module-info.java
module1.iml
module2
src
chapter11.net
first
SimpleTask
SimpleTask.class
second
module-info.class
module-info.java
package chapter11.net.first;
import chapter11.com.first.pack.Task;
public class SimpleTask {
public static void main(String[] args) {
System.out.println("Inside main");
System.out.println(Task.getTaskInfo());
}
}
Terminal:
Local
PS C:\Users\Andrian\Project\bvb-crawler\ocp> jar -cvf mods/chapter11.net.jar -C module2/src .
added manifest
added module-info: module-info.class
adding: chapter11/(in = 0) (out= 0)(stored 0%)
adding: chapter11/net/(in = 0) (out= 0)(stored 0%)
adding: chapter11/net/first/(in = 0) (out= 0)(stored 0%)
adding: chapter11/net/first/SimpleTask.class(in = 548) (out= 350)(deflated 35%)
adding: chapter11/net/first/SimpleTask.java(in = 251) (out= 159)(deflated 36%)
adding: chapter11/net/second/(in = 0) (out= 0)(stored 0%)
adding: module-info.java(in = 84) (out= 63)(deflated 25%)
PS C:\Users\Andrian\Project\bvb-crawler\ocp>

And run it


ocp module2 src chapter11 net first SimpleTask main
TestResourceBundle.testResource
Project
Task.java
module-info.java (chapter.com)
module-info.java (chapter.net)
SimpleTask.java
ocp C:\Users\Andrian\Project\bvb-crawler\oc
.idea
mods
chapter11.com.jar
chapter11.net.jar
module1
src
chapter11.com
module-info.class
module-info.java
module1.iml
module2
src
chapter11.net
first
SimpleTask
SimpleTask.class
second
module-info.class
module-info.java
package chapter11.net.first;
import chapter11.com.first.pack.Task;
public class SimpleTask {
public static void main(String[] args) {
System.out.println("Inside main");
System.out.println(Task.getTaskInfo());
}
}
Terminal:
Local
PS C:\Users\Andrian\Project\bvb-crawler\ocp> java --module-path mods --module chapter.net/chapter11.net.first.SimpleTask
Inside main
In the task class
generatedString from SecondTask class inside private package
In the Task class and also in SecondTask class
PS C:\Users\Andrian\Project\bvb-crawler\ocp>
Project
Structure
Favorites