Practicing the Java Module System

Table of Contents

To start hands-on lab session, first you should clone the repository[1] or download zipped version here https://github.com/rahmanusta/practicing-java9-module-system/archive/master.zip

Attendees who want to practice should install required tools in advance WORKSHOP_REQUIREMENTS .

[1] https://github.com/rahmanusta/practicing-java9-module-system

1

Practice 1: Generate JREs

JLink tool allows you to create different scale of JREs. Every application has different scale, and it is not required to use a module that we don’t need!

```
mkdir generate-image
cd generate-image

// Run
generate-images.bat or ./geneate-images.sh
```

After generation is completed, verify that you have `java.base`, `java.desktop`, `java.se`, and `java.se.ee` included JREs in the directory. Check folder sizes and add them to your notepad.

2

Practice 2: Create a hello world module

Create a "Hello world!" module. Compile, Package, and Run it.

1) Get the following directory structure, check the module-info.java and Hello.java classes.

module-info.java.

```
module com.foo {
    // no definition yet
}
```

com.foo.Hello.java.

```
package com.foo;

public class Hello {

    public static void main(String[] args) {
```

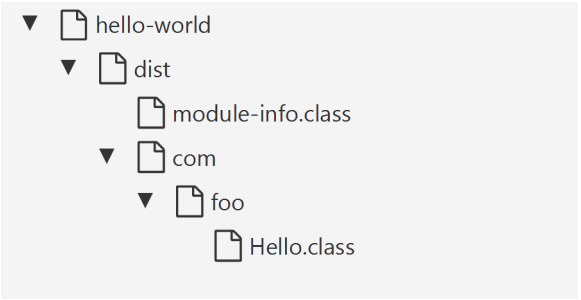
```
        System.out.println("Hello world!");

    }

}
```

2) Compile the module artifacts

module-info.java is a module descriptor file. It should be compiled along with other classes in the module.

```
cd hello-world
javac src/module-info.java src/com/foo/Hello.java -d dist
```

After compilation, hello-world/dist folder should include compiled module descriptor and Hello.class.



3) Run the module with dist folder

Add dist directory to the module path to be able to resolve **com.foo** module inside, and run **com.foo.Hello** class inside **com.foo** module.

```
java --module-path dist -m com.foo/com.foo.Hello
// Hello world!
```

4) Package the modular app

We can package standard Java module as a JAR (Java Archive) file. Then, we call it as modular JAR file.

```
jar --create --file hello.jar --main-class=com.foo.Hello -C dist .
```

After the jar command completed, verify that you have hello.jar file in the current directory.

5) Run the module with a modular JAR file

We can also add modular JAR files to the --module-path.

```
java --module-path hello.jar -m com.foo/com.foo.Hello
// Hello world!
```

6) Link

jlink is a link tool for Java 9. It creates a portable bundle of your application and JRE.

```
jlink --module-path %JAVA_HOME%/jmods;hello.jar --add-modules com.foo --
launcher hello=com.foo/com.foo.Hello --output release ❶

jlink --module-path $JAVA_HOME/jmods;hello.jar --add-modules com.foo --
launcher hello=com.foo/com.foo.Hello --output release ❷
```

❶ For win

❷ For *nix

After link, you will have a special JRE which has Hello module included. You can run the module with produced launcher.

```
cd release
bin\hello.bat ❶
./bin/hello ❷
```

❶ for win

❷ for *nix

3

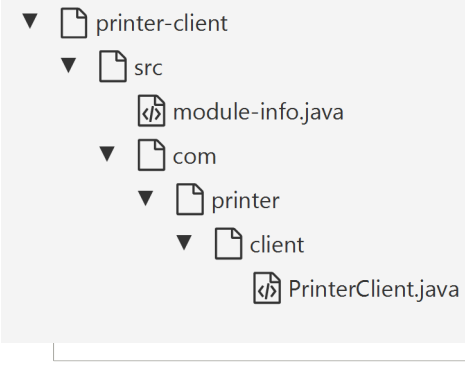
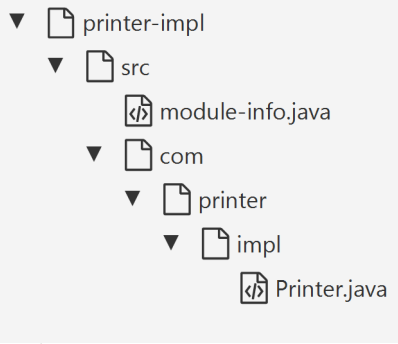
Practice 3: Control access between modules

Let's use `exports` and `requires` keywords to control **access** between modules.

exports
: declares which package(s) will be readable to outside modules.

requires
: declares which module(s) are needed to read/access.

Table 3.1. module-core directory

printer-client module	printer-impl module
	

In this practice, you are going to play with exports and requires keywords to understand them easily.

Case 1
: When exports and requires are not declared in module descriptor.

Open `module-core` directory, and check that `printer-client/module-info.java` doesn't include `requires` and `printer-impl/module-info.java` doesn't include `exports` keywords.

Compile printer-impl and printer-client modules, and log that how Java module system prevents access when `exports` and `requires` are missed.

Compile modules.

```
javac printer-impl/module-info.java printer-impl/src/com/printer/impl/
Printer.java -d dist/printer-impl ❶

javac printer-client/module-info.java printer-client/src/com/printer/
client/PrinterClient.java -p dist/printer-impl -d dist/printer-client ❷
```

❶ Compile printer-impl

❷ Compile printer-client

Run modules.

```
java --module-path dist -m com.printer.client/
com.printer.client.PrinterClient
```

Case2

When exports declared, but requires is not declared

Update `printer-impl/module-info.java` descriptor file to export `com.printer.impl` package to other modules.

printer-impl/module-info.java.

```
module com.printer.impl {
    exports com.printer.impl;
}
```

Compile printer-impl and printer-client modules, and log that how Java module system prevents access when `requires` is missed.

Compile modules.

```
javac printer-impl/module-info.java printer-impl/src/com/printer/impl/
Printer.java -d dist/printer-impl ❶
```

```
javac printer-client/module-info.java printer-client/src/com/printer/
client/PrinterClient.java -p dist/printer-impl -d dist/printer-client ❷
```

❶ Compile printer-impl

❷ Compile printer-client

Run modules.

```
java --module-path dist -m com.printer.client/
com.printer.client.PrinterClient
```

Case 3

When requires declared, but exports is not declared

Update `printer-impl/module-info.java` descriptor file to not export any package, and update `printer-client/module-info.java` to require printer-impl module.

printer-impl/module-info.java.

```
module com.printer.impl {

}
```

printer-client/module-info.java.

```
module com.printer.client {
    requires com.printer.impl;
}
```

Compile modules.

```
javac printer-impl/module-info.java printer-impl/src/com/printer/impl/
Printer.java -d dist/printer-impl ❶

javac printer-client/module-info.java printer-client/src/com/printer/
client/PrinterClient.java -p dist/printer-impl -d dist/printer-client ❷
```

❶ Compile printer-impl

❷ Compile printer-client

Run modules.

```
java --module-path dist -m com.printer.client/
com.printer.client.PrinterClient
```

Compile printer-impl and printer-client modules, and log that how Java module system prevents access when `exports` is missed.

Case 4

When both requires and exports are declared

Update `printer-impl/module-info.java` descriptor file to export `com.printer.impl` package, and update `printer-client/module-info.java` to require `com.printer.impl` module.

Compile modules.

```
javac printer-impl/module-info.java printer-impl/src/com/printer/impl/
Printer.java -d dist/printer-impl ❶

javac printer-client/module-info.java printer-client/src/com/printer/
client/PrinterClient.java -p dist/printer-impl -d dist/printer-client ❷
```

❶ Compile printer-impl

❷ Compile printer-client

Run modules.

```
java --module-path dist -m com.printer.client/
com.printer.client.PrinterClient
```

Compile printer-impl and printer-client modules, and log that how Java module system controls access among modules successfully.

4

Practice 4: Using Auto-modules

Auto-modules is designed for smooth migration to the Java module system.

When a non-modular classic JAR file is added to module path (`--module-path` or `-p`), then it becomes an auto-module.

All packages of an auto-module are readable by other modules.

Open `auto-module` directory, and check that there is a non-modular jansi-1.17.1.jar file. Then, edit module descriptors for both printer-client and printer-impl modules.

printer-client/module.descriptor.java.

```
module com.printer.client {
    requires com.printer.impl;
}
```

printer-impl/module.descriptor.java.

```
module com.printer.impl {
    exports com.printer.impl;

    requires jansi; ❶
}
```

❶ jansi is a non-modular Jar file and it behaves like a module (auto-module) if it is added to module path. Module name is resolved without version part from the file name.

Compile modules.

```
javac printer-impl/module-info.java printer-impl/src/com/printer/impl/
Printer.java -p lib -d dist/printer-impl

javac printer-client/module-info.java printer-client/src/com/printer/
client/PrinterClient.java -p dist/printer-impl;lib -d dist/printer-
client
```

Run modules.

```
java --module-path dist;lib  -m com.printer.client/
com.printer.client.PrinterClient
```

After compile, run the auto-module app, verify that console output is colored with jansi library.

5

Practice 5: Using Unnamed modules

In this practice, you are going to test access from an unnamed module to a module.

Open `unnamed-module` folder, and check that `printer-client` module doesn't have a module descriptor file, and compile the modules.

```
javac src/module-info.java src/com/printer/impl/Printer.java -d dist/
printer-impl ❶

javac src/com/printer/client/PrinterClient.java -d dist/printer-client ❷
```

❶ Compile printer-impl module

❷ Compile printer-client without module-info.java

After compilation, run non-modular printer-client and verify that it is able to access exported packages.

```
java -p dist/printer-impl --add-modules com.printer ❶
     -cp dist/printer-client com.printer.client.PrinterClient ❷
```

❶ com.printer is added to module path

❷ printer-client is added to classpath

6

Practice 6:: Testing HTTP/2 Client API

In this practice you are going to run a HTTP/2 server featured with Spring Boot + Undertow and a modular HTTP/2 client application to test the HTTP/2 Client API.

Open `http2/http-server` folder and run the following.

```
mvnw clean spring-boot:run ❶
./mvnw clean spring-boot:run ❷
```

❶ For windows

❷ For *nix

Server application listens on https://127.0.0.1:8443 and has simple logic to respond to the HTTP/2 client.

HelloController.java.

```
@RestController
public class HelloController {

    @GetMapping("/get")
    public String get() {
        return "Hello world!";
    }

    @PostMapping("/post")
    public String post(@RequestBody String body) {
        return isNull(body) ? "Hello world!" : body;
```

```
    }
}
```

HTTP/2 protocols forces to use SSL certificates. In application folder you will see `http2.keystore`, and it can be used for both server and client applications.

On the server side, the keystore can be registered in Spring Boot's application.properties file.

```
server.ssl.key-store=classpath:http2.keystore
server.ssl.key-store-password=123456
server.ssl.key-alias=http2

server.http2.enabled=true

server.port=8443
```

In the client application, we have two simple classes `GetClient` and `PostClient`. They simple requests to the HTTP/2 server we run in previous step and makes HTTP Get or Post requests.

Let's build the client application.

```
javac src/module-info.java src/com/kodedu/http2/client/GetClient.java
 src/com/kodedu/http2/client/PostClient.java -d dist
```

Then run the GetClient with required trustStore arguments.

```
java --module-path dist -Djavax.net.ssl.trustStore="src/resources/
http2.keystore" -Djavax.net.ssl.trustStorePassword=123456 -m
 com.kodedu.httpTwo/com.kodedu.http2.client.GetClient
```

After run, verify that version information (HTTP_2), http status code (200), and response body correctly printed.

Repeat the same operation for `PostClient`, and see that it successfully handles Http Post requests.

HTTP/2 API practice needs SSL keystore. If you want to generate your own keystore follow the instruction below.

Generates keystore.

```
keytool -genkeypair -alias http2 -keyalg RSA -
keysize 2048 -keystore http2.keystore -validity 3650
```

Answer questions asked during keystore generation.

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Rahman Usta
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=Rahman Usta, OU=Unknown, O=Unknown, L=Unknown,
 ST=Unknown, C=Unknown correct?
  [no]:  yes
```

7

Practice 7: Testing JSHell

JShell is a REPL tool for the Java platform.

```
jshell

|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro
jshell>
jshell> 3 + 5
jshell> /exit
```

Create a Calculator class and call it's methods in JShell terminal.

```
public class Calculator {

    public int sum(int a, int b) {
        return a+b;
    }

    // ...

}
```

The end.

8

License

MIT