Chapter 9: Modules

- Java Modules
- Java Module Benefits
- Java Module Basics
    - Modules Contain One or More Packages
    - Module Naming
    - Module Root Directory
    - Module Descriptor (module-info.java)
    - Module Exports
    - Module Requires
    - Circular Dependencies Not Allowed
    - Split Packages Not Allowed
- Compiling a Java Module
- Running a Java Module
- Building a Java Module JAR File
    - Setting the JAR Main Class
- Running a Java Module From a JAR
    - Running a Java Module From a JAR With a Main Class Set
- Packing a Java Module as a Standalone Application
    - Running the Standalone Application
- Unnamed Module
- Automatic Modules
- Practical Example

Java Modules

A Java Module is a mechanism to package up your Java application and Java packages into Java modules. A Java module can specify which of the Java packages it contains that should be visible to other Java modules using this module. A Java module must also specify which other Java modules are required to do its job.

Java module is a new feature in Java 9 via the Java Platform Module System (JPMS). The Java Platform Module System is also sometimes referred to as Java Jigsaw or Project Jigsaw depending on where you read. Jigsaw was the internally used project name during development. Later Jigsaw changed name to Java Platform Module System.

Java Module Benefits

- Smaller Application Distributable via the Modular Java Platform
- Encapsulation of Internal Packages
- Start-up Detection of Missing Modules

Java Module Basics

Modules Contain One or More Packages

A Java module is one or more Java packages that belong together. A module could be either a full Java application, a Java Platform API, or a third party API.

Module Naming

A Java module must be given a unique name. For instance, a valid module name could be

com.vedisoft.mymodule

Module Root Directory

From Java 9 a module must be nested under a root directory with the same name as the module. In the example above we have a directory structure for a package named ***"com.vedisoft.mymodule"*** .

The directory structure for the above Java package contained in a Java module of the same name, would look like this:

com.vedisoft.mymodule/com/vedisoft/mypack

Module Descriptor (module-info.java)

Each Java module needs a Java module descriptor named module-info.java which has to be located in the corresponding module root directory. For the module root directory src/main/java/com.vedisoft.mymodule the path to the module's module descriptor will be src/main/java/com.vedisoft.mymodule/module-info.java .

The module descriptor specifies which packages a module exports, and what other modules the module requires. These details will be explained in the following sections. Here is how a basic, empty Java module descriptor looks:

```
module com.vedisoft.mymodule {

}
```

Module Exports

A Java module must explicitly export all packages in the module that are to be accessible for other modules using the module. The exported packages are declared in the module descriptor. Here is how a simple export declaration looks inside a module descriptor:

```
module com.vedisoft.mymodule {
   exports com.vedisoft.mypack;
}
```

To export a subpackage also, you must declare it explicitly in the module descriptor, like this:

```
module com.vedisoft.mymodule {
   exports com.vedisoft.mypack;
   exports com.vedisoft.mypack.util;
}
```

You do not have to export the parent package in order to export a subpackage. The following module descriptor exports statement is perfectly valid:

```
module com.vedisoft.mymodule {
   exports com.vedisoft.mypack.util;
}
```

This example only exports the com.vedisoft.mymodule.util package, and not the com.vedisoft.mymodule package

Module Requires

If a Java module requires another module to do its work, that other module must be specified in the module descriptor too. Here is an example of a Java module requires declaration:

```
module com.vedisoft.mymodule {
   requires java.desktop;
}
```

This example module descriptor declares that it requires the standard Java module named java.desktop.

Circular Dependencies Not Allowed

It is not allowed to have circular dependencies between modules. In other words, If module A requires module B, then module B cannot also require module A. The module dependency graph must be an acyclic graph.

Split Packages Not Allowed

The same Java package can only be exported by a single Java module at runtime. In other words, you cannot have two (or more) modules that export the same package in use at the same time. The Java VM will complain at startup if you do.

Having two modules export the same package is also sometimes referred to as a split package. By split package is meant that the total content (classes) of the package is split between multiple modules. This is not allowed.

Compiling a Java Module

javac -d out --module-source-path . --module com.vedisoft.mymodule

When javac compiles the module it writes the compiled result into the directory specified after the -d argument to the javac command. Inside that directory you will find a directory with the name of the module, and inside that directory you will find the compiled classes plus a compiled version of the module-info.java module descriptor named module-info.class.

The --module-source-path should point to the source root directory, not the module root directory. The source root directory is normally one level up from the module root directory.

The --module argument specifies which Java module to compile. In the example above it is the module named com.vedisoft.mymodule. You can specify multiple modules to be compiled by separating the module names with a comma. For instance:

... --module com.vedisoft.mymodule1, com.vedisoft.mymodule2

Running a Java Module

In order to run the main class of a Java module you use the java command, like this:

java --module-path out --module com.vedisoft.mymodule/com.vedisoft.mypack.Main

The --module-path argument points to the root directory where all the compiled modules are located. Remember, this is one level above the module root directory.

The --module argument tells what module + main class to run. In the example the module name is the com.vedisoft.mymodule part and the main class name is the com.vedisoft.mypack.Main . Notice how the module name and main class name are separated by a slash (/) character

Building a Java Module JAR File

The jar command needed to generate a JAR file from a compiled Java module

jar -c --file=out/com-vedisoft-mymodule.jar -C out/com.vedisoft.mymodule .

Setting the JAR Main Class

You can still set the JAR main class when generating the module JAR file. You do so by providing a --main-class argument. Here is an example of setting the main class of a Java module JAR file:

jar -c --file=out/com-vedisoft-mymodule.jar --main-class=com.vedisoft.mypack.Main -C out/com.vedisoft.mymodule .

Running a Java Module From a JAR

Once you have packaged your Java module into a JAR file, you can run it just like running a normal module. Just include the module JAR file on the module path. Here is how you run the main class from a Java module JAR file:

java --module-path out -m com.vedisoft.mymodule/com.vedisoft.mypack.Main

Running a Java Module From a JAR With a Main Class Set

If the Java module JAR file has a main class set, you can run the Java module main class with a little shorter command line. Here is an example of running a Java module from a JAR file with a main class set:

java -jar out/com-vedisoft-mymodule.jar

Notice how no --module-path argument is set. This requires that the Java module does not use any third party modules. Otherwise you should provide a --module-path argument too, so the Java VM can find the third party modules your module requires.

Packing a Java Module as a Standalone Application

You can package a Java module along with all required modules (recursively) and the Java Runtime Environment into a standalone application. The user of such a standalone application does not need to have Java pre-installed to run the application, as the application comes with Java included. That is, as much of the Java platform as the application actually uses.

You package a Java module into a standalone application using the jlink command which comes with the Java SDK. Here is how you package a Java module with jlink :

jlink --module-path "out;C:\Program Files\Java\jdk-17.0.0\jmods" --add-modules com.vedisoft.mymodule --output out-standalone

The --module-path argument specifies the module paths to look for modules in. The example above sets the out directory into which we have previously compiled our module, and the jmods directory of the JDK installation.

The --add-modules argument specifies the Java modules to package into the standalone application. The example above just includes the com.vedisoft.mymodule module.

The --output argument specifies what directory to write the generated standalone Java application to. The directory must not exist already.

Running the Standalone Application

Once packaged, you can run the standalone Java application by opening a console (or terminal), change directory into the standalone application directory, and execute this command:

bin\java --module com.vedisoft.mymodule/com.vedisoft.mypack.Main

The standalone Java application contains a bin directory with a java executable in. This java executable is used to run the application.

The --module argument specifies which module plus main class to run.

Unnamed Module

From Java 9 and forward, all Java classes must be located in a module for the Java VM to use them. But what do you do with older Java libraries where you just have the compiled classes, or a JAR file?

In Java 9 you can still use the -classpath argument to the Java VM when running an application. On the classpath you can include all your older Java classes, just like you have done before Java 9. All classes found on the classpath will be included in what Java calls the unnamed module.

The unnamed module exports all its packages. However, the classes in the unnamed module are only readable by other classes in the unnamed module. No named module can read the classes of the unnamed module.

If a package is exported by a named module, but also found in the unnamed module, the package from the named module will be used.

All classes in the unnamed module requires all modules found on the module path. That way, all classes in the unnamed module can read all classes exported by all the Java modules found on the module path.

Automatic Modules

What if you are modularizing your own code, but your code uses a third party library which is not yet modularized? While you can include the third party library on the classpath and thus include it in the unnamed module, your own named modules cannot use it, because named modules cannot read classes from the unnamed module.

The solution is called automatic modules. An automatic module is made from a JAR file with Java classes that are not modularized, meaning the JAR file has no module descriptor. This is the case with JAR files developed with Java 8 or

earlier.

When you place an ordinary JAR file on the module path (not the classpath) the Java VM will convert it to an automatic module at runtime.

An automatic module requires all named modules on the module path. In other words, it can read all packages exported by all named modules in the module path.

If your application contains multiple automatic modules, each automatic module can read the classes of all other automatic modules.

An automatic module can read classes in the unnamed module. This is different from explicit named modules (real Java modules) which cannot read classes in the unnamed module.

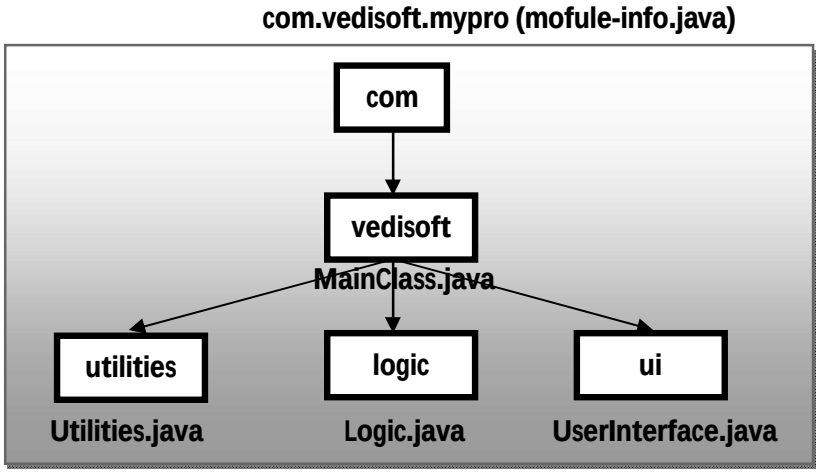
An automatic module exports all its packages, so all named modules on the module path can use the classes of an automatic module. Named modules still have to explicitly require the automatic module though.

The rule about not allowing split packages also counts for automatic modules. If multiple JAR files contain (and thus exports) the same Java package, then only one of these JAR files can be used as an automatic module.

An automatic module is a named module. The name of an automatic module is derived from the name of the JAR file. If the name of the JAR file is com-vedisoft-mymodule.jar the corresponding module name will be com.vedisoft.mymodule. The - (dash) characters are not allowed in a module name, so they are replaced with the . character. The .jar suffix is removed.

If a JAR file contains versioning in its file name, e.g. com-vedisoft-mymodule-2.9.1.jar then the versioning part is removed from the file name too, before the automatic module name is derived. The resulting automatic module name is thus still com.vedisoft.mymodule .

Practical Example



NOTE: classes from com.vedisoft.utilities package can also be used from outside the module

1. Create a folder "com.vedisoft.mypro"

2. Create the following package structure within "com.vedisoft.mypro" folder as per the above diagram.

3. Create a class called "Utilities.java" within "com.vedisoft.mypro\com\vedisoft\utilities" folder

```
package com.vedisoft.utilities;

public class Utilities {
	public Utilities() {
		System.out.println("These are utilities for the project.");
	}
}
```

4. Create a class called "Logic.java" within "com.vedisoft.mypro\com\vedisoft\logic" folder

```
package com.vedisoft.logic;

public class Logic {
	public Logic() {
		System.out.println("These are logics for the project.");
	}
}
```

5. Create a class called "UserInterface.java" within "com.vedisoft.mypro\com\vedisoft\ui" folder

```
package com.vedisoft.ui;

public class UserInterface {
	public UserInterface() {
		System.out.println("These are user interfaces for the project.");
	}
}
```

6. Create the "MainClass" within "com.vedisoft.mypro\com\vedisoft" that contains a main() and it creates objects of UserInterface class, Utilities class and Logic class. Compile and interpret it.

```
package com.vedisoft;

import com.vedisoft.utilities.*;
import com.vedisoft.ui.*;
import com.vedisoft.logic.*;

public class MainClass {

	public static void main(String args[]) {
		Utilities a = new Utilities();
```

```
        UserInterface b = new UserInterface();
        Logic c = new Logic();
    }
}
```

7. Create "module-info.java" within "com.vedisoft.mypro" folder

```
module com.vedisoft.mypro {
    exports com.vedisoft.utilities;
}
```

8. Compile and run the module "com.vedisoft.mypro"

```
G:\Data> javac -d out --module-source-path . --module com.vedisoft.mypro
G:\Data>java --module-path out --module com.vedisoft.mypro/com.vedisoft.MainClass
```

9. Create the Module Jar File and set "MainClass" class as the main class

```
G:\Data>jar -c --file=out-standalone/com-vedisoft-mypro.jar --main-class=com.vedisoft.MainClass -C
out/com.vedisoft.mypro .
```

NOTE: create a folder by the name "out-standalone" before executing this command

10. Execute the Java Module from a jar

```
java -jar out-standalone/com-vedisoft-mypro.jar
```