Java Language

Java Modules

Jakob Jenkov
Last update: 2018-03-06

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 is requires to do its job. This will be explained in more detail later in this Java modules tutorial.

Java modules 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

The Java Platform Module System brings several benefits to us Java developers. I will list the biggest benefits below.

modules your application requires, Java can package up your application including only the Java Platform modules that your application actually uses.

Before Java 9 and the Java Platform Module System you would have had to package all of the Java Platform APIs with your Java application because there was no official way of reliably checking what classes your Java application used. Since the Java Platform APIs have grown quite large over the years, your application would get a large amount of Java classes included in its distribution, many of which your application would probably not be using.

The unused classes makes your application distributable bigger than it needs to be. This can be a problem on small devices like mobile phones, Raspberry Pis etc. With the Java Platform Module System you can now package your application with only the modules of the Java Platform APIs that your application is actuallly using. This will result in smaller application distributables.

Encapsulation of Internal Packages

A Java module must explicitly tell which Java packages inside the module are to be exported (visible) to other Java modules using the module. A Java module can contain Java packages which are not exported. Classes in unexported packages cannot be used by other Java modules. Such packages can only be used internally in the Java module that contains them.

Packages that are not exported are also referred to as hidden packages, or encapsulated packages.

Startup Detection of Missing Modules

From Java 9 and forward, Java applications must be packaged as Java modules too. Therefore an application module specifies what other modules (Java API modules or third party modules) it uses. Therefore the Java VM can check the whole module dependency graph from the application module and forward, when the Java VM starts up. If any required modules are not found at startup, the Java VM reports the missing module and shuts down.

Before Java 9 missing classes (e.g. from a missing JAR file) would not be detected until the application actually tried to use the missing class. This would happen sometime at runtime - depending on when the application tried to use the missing class.

Having missing modules reported at application startup time is a big advantage compared to at runtime when trying to use the missing module / JAR / class.

Java Module Basics

Now you know what a Java module is and what the benefits of Java modules are, let us take a look at the basics of Java modules.

Modules Contain One or More Packages

All Trails | Trail TOC | Page TOC | Previous | Next

Module Naming

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

```
com.jenkov.mymodule
```

A Java module name follows the same naming rules as Java packages. However, you should not use underscores (_) in module names (or package names, class names, method names, variable names etc.) from Java 9 and forward, because Java wants to use underscore as a reserved identifier in the future.

It is recommended to name a Java module the same as the name of the root Java package contained in the module - if that is possible (some modules might contain multiple root packages).

Module Root Directory

Before Java 9 all Java classes for an application or API were nested directly inside a root class directory (which was added to the classpath), or directly inside a JAR file. For instance, the directory structure for the compiled packages of `com.jenkov.mymodule` would look like this:

```
com/jenkov/mymodule
```

A little more graphically, it would look like this:

- com
    - jenkov
        - mymodule

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.jenkov.mymodule` . This Java package is to be contained within a Java module with the same name (also `com.jenkov.mymodule`).

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

```
com.jenkov.mymodule/com/jenkov/mymodule
```

A little more graphically, it would look like this:

All Trails Trail TOC Page TOC Previous Next

Notice the fullstops (`.`) in the module root directory name. These fullstops need to be there because they are part of the module name! They are not to be interpreted as subdirectory path dividers!

The module root directory is used both for the source files and compiled classes of a Java module. That means, that if your Java project has a source root directory named `src/main/java` - then each module inside your project will have its own module root directory under `src/main/java`. For instance:

```
src/main/java/com.jenkov.module1

src/main/java/com.jenkov.module2
```

The same directory structure would be seen in the Java compiler's output directory.

It is common to only have one Java module per project. You will still need the module root directory in that case, but the source and compiler output root directories will only contain a single module root directory.

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.jenkov.mymodule` the path to the module's module descriptor will be `src/main/java/com.jenkov.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.jenkov.mymodule {

}
```

First is the `module` keyword, followed by the name of the module, and then a set of curly brackets. The exported packages and required modules will be specified inside the curly brackets.

Notice also how the module descriptor is suffixed `.java` and yet it uses a hyphen in the file name (`module-info.java`). Hyphens are not normally allowed in Java class names, but in module descriptor file names they are required!

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

```
    exports com.jenkov.mymodule;
}
```

This example exports the package called `com.jenkov.mymodule` .

Please note, that only the listed package itself is exported. No "subpackages" of the exported package are exported. That means, that if the `mymodule` package contained a subpackage named `util` then the `com.jenkov.mymodule.util` package is *not* exported just because `com.jenkov.mymodule` is.

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

```
module com.jenkov.mymodule {
    exports com.jenkov.mymodule;
    exports com.jenkov.mymodule.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.jenkov.mymodule {
    exports com.jenkov.mymodule.util;
}
```

This example only exports the `com.jenkov.mymodule.util` package, and not the `com.jenkov.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.jenkov.mymodule {
    requires javafx.graphics;
}
```

This example module descriptor declares that it requires the standard Java module named `javafx.graphics`.

Circular Dependencies Not Allowed

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

In order to compile a Java module you need to use the `javac` command that comes with the Java SDK. Remember, you need Java 9 or later to compile a Java module.

```
javac -d out --module-source-path src/main/java --module com.jenkov.mymodule
```

Remember you must have the `javac` command from the JDK installation on your path (environment variable) for this command to work. Alternatively you can replace the `javac` part in the command above with the full path to where the `javac` command is located, like this:

```
"C:\Program Files\Java\jdk-9.0.4\bin\javac" -d out --module-source-path src/main/java --module com.jenkov.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.jenkov.mymodule`. You can specify multiple modules to be compiled by separating the module names with a comma. For instance:

```
... --module com.jenkov.mymodule1,com.jenkov.mymodule2
```

Running a Java Module

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

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.jenkov.mymodule` part and the main class name is the `com.jenkov.mymodule.Main` . Notice how the module name and main class name are separated by a slash (`/`) character.

Building a Java Module JAR File

You can package a Java module inside a standard JAR file. You do so with the standard `jar` command that comes with the Java SDK. The package directory hierarchy must start at the root of the JAR file, just like for pre Java 9 JAR files. Additionally, a Java module JAR file contains a compiled version of the module descriptor at the root of the JAR file.

Here is the `jar` command needed to generate a JAR file from a compiled Java module:

```
jar -c --file=out-jar/com-jenkov-mymodule.jar -C out/com.jenkov.mymodule .
```

The `-c` argument tells `jar` to create a new JAR file.

The `--file` argument tells the path of the output file - the created JAR file. Any directories you want the output JAR file to be under must already exist!

The `-C` (uppercase C) argument tells the `jar` command to change directory to `out/com.jenkov.javafx` (the compiled module root directory) and then include everything found in that directory - due to the following `.` argument (which signals "current directory").

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-jar/com-jenkov-mymodule.jar --main-class=com.jenkov.mymodule.Main -C out/com.jenkov.mymodule .
```

You can now run the main class of this JAR file with a shortcut. This shortcut will be explained in the next section.

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

For this command to work the module JAR file must be located in the `out-jar` directory.

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

If the Java module JAR file has a main class set (see a few sections earlier in this tutorial for how to do that), 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-jar/com-jenkov-javafx.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-9.0.4\jmods" --add-modules com.jenkov.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.jenkov.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

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

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-jenkov-mymodule.jar` the corresponding module name will be `com.jenkov.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-jenkov-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.jenkov.mymodule` .

Services

With Java 9 comes a new concept called *services*. Java services is related to the Java Platform Module System, so I will explain Java services in this Java module tutorial.

A service consists of two major parts:

1. A service interface.
2. One or more service implementations.

The service interface is typically located in a service interface Java module which only contains the service interface, plus any classes and interfaces related to the service interface.

The service implementations are provided by separate Java modules - not the service interface module. Typically a service implementation Java module will contain a single service implementation.

A Java module or application can require the service interface module and code against the service interface, without knowing exactly which other module delivers the service implementation. The service implementation is discovered at runtime, and depends on what service implementation modules are available on the Java module path when the application is launched.

Service Interface Module

Java service interface modules do not require a special declaration of the service interface. You just create a regular Java module. Here is a Java service module descriptor example:

```
module com.jenkov.myservice {
    exports com.jenkov.myservice
```

service interface. The service interface is just a normal Java interface, so I have not shown an example of it.

Service Implementation Module

A Java module that wants to implement a service interface from a service interface module must:

- Require the service interface module in its own module descriptor.
- Implement the service interface with a Java class.
- Declare the service interface implementation in its module descriptor.

Imagine that the `com.jenkov.myservice` module contains an interface named `com.jenkov.myservice.MyService`. Imagine too, that you want to create a service implementation module for this service interface. Imagine that your implementation is called `com.blabla.myservice.MyServiceImpl`. To declare that service implementation the module descriptor for the service implementation module would have to look like this:

```
module com.blabla.myservice {
  requires com.jenkov.myservice;

  provides com.jenkov.myservice.MyService with
        com.blabla.myservice.MyServiceImpl
}
```

The module descriptor first declares that it requires the service interface module. Second, the module descriptor declares that it provides an implementation for the service interface `com.jenkov.myservice.MyService` via the class `com.blabla.myservice.MyServiceImpl`.

Now that this module declares that it implements the service interface, we need to see how a Java module can lookup an implementation of the service interface at runtime.

Service Client Module

Once you have both a service interface module and a service implementation module, you can create a client module that uses the service. Sometimes a service client module is referred to as a service consumer module or service user module, but the meaning is the same - a module that uses a service specified in an external module and implemented by yet another external module.

In order to use the service, the client module must declare in its module descriptor that it uses the service. Here is how to declare the use of a service in a module descriptor:

```
module com.client.myservicelient {
    requires com.jenkov.myservice;
```

Notice how the client module descriptor also declares that it requires the `com.jenkov.myservice` module which contains the service interface. It does not need to require the service implementation modules. Those are looked up at runtime. Only the service interface module must be required.

The advantage of not having to declare the service implementation modules is, that the implementation modules can be exchanged without breaking the client code. You can decide what service implementation to use when assembling the application - by dropping the desired service implementation module(s) onto the module path. The client module, and service interface module, are thus decoupled from the service implementation modules.

Now the service client module can lookup a service interface implementation at runtime like this:

```
Iterable<MyService> services =
        ServiceLoader.load(MyService.class);
```

The returned `Iterator` contains a list of implementations of the `MyService` interface. In fact, it will contain all the implementations found in the modules found on the module path. The client module can now iterate the `Iterator` and find the service implementation it wants to use.

Module Versioning

The Java Platform Module System does not support versioning of Java modules. You cannot have multiple versions of a module available on the module path when running a Java 9+ application. You will need to use a build tool like **Maven** or **Gradle** to handle versioning of your module, and the external modules your module depends on (uses / requires).

Multi Java Version Module JAR Files

From Java 9 it is possible to create JAR files for Java modules which contains code compiled specifically for different versions of Java. That means, that you can create a JAR file for your module that contains code compiled for Java 8, Java 9, Java 10 etc. - all within the same JAR file.

Here is how the structure looks of a multi Java version JAR file:

- META-INF
  - MANIFEST.MF
  - versions
    - 10
      - com

All Trails | Trail TOC | Page TOC | Previous | Next

The `com` folder at the root level of the JAR file contains the compiled Java classes for pre Java 9 versions. Earlier versions of Java do not understand multi Java version JAR files so they use the classes found here. Therefore you can only support one Java version earlier than Java 9.

The `META-INF` directory contains the `MANIFEST.MF` file and a directory named `versions`. The `MANIFEST.MF` file needs a special entry that marks the JAR file as a multi version JAR file. Here is how this entry looks:

```
Multi-Release: true
```

The `versions` directory which can contain the compiled classes for different versions of Java for your module. In the example above there are two subdirectories inside the `versions` directory. One subdirectory for Java 9, and one for Java 10. The names of the directories correspond to the numbers of the Java versions to support.

Migrating to Java 9

Parts of the Java Platform Module System are designed to ease migration of applications written in pre Java 9 to Java 9. The module system is designed to support "bottom up migration" - meaning that you migrate your small utility libraries first, and your main applications last.

The migration is intended to go something like this:

1. Upgrade to Java 9, and run your application without modularizing anything.
   Put classes and JAR files on the classpath as normal. These classes then become part of the unnamed module.

2. Move utility library JAR files to the module path. This way they become automatic modules. Do this for both internal and third party utility libraries. The main applications should still be on the classpath in the unnamed module. The unnamed module can read all named modules, automatic or non-automatic.

3. Migrate internal and third party utility libraries to Java modules when possible. Place the Java modules on the module path. Start with libraries that have no other module / JAR dependencies, and move up in the dependency hierarchy.

4. Migrate your main applications to Java modules.

By upgrading bottom up the chance is higher of your applications being able to work in the period before everything is upgraded to Java modules. Your application is supposed to be able to work during any of the above phases.

Upgrading utility libraries first to automatic modules, and later to full modules, starting at the bottom of the dependency hierarchy, should assure that your libraries can still read each other during upgrade, plus be readable by the main applications on the classpath in the you will have some time where you may not be able to assemble a working application, but once you are through it, you are through it. You are not dragging old dependencies around in your application. If possible, this is definitely preferable.

Next: Java Exercises

Tweet

Jakob Jenkov

Right 1
Right 2