(/)

A Guide to Java Modularity

Last updated: January 17, 2024

Written by: Christopher Franklin
(https://www.baeldung.com/author/christopher-franklin)

> Reviewed by: Josh Cummings (https://www.baeldung.com/editor/josh-cummings)

Core Java (https://www.baeldung.com/category/core-java)

>= Java 8 (https://www.baeldung.com/tag/jdk8-and-later)

1. Overview

Java 9 introduces a new level of abstraction above packages, formally known as the Java Platform Module System (JPMS), or “Modules” for short.

In this tutorial, we’ll go through the new system and discuss its various aspects.

We’ll also build a simple project to demonstrate all concepts we’ll be learning in this guide.

2. What's a Module? (/)

First of all, we need to understand what a module is before we can understand how to use them.

A Module is a group of closely related packages and resources along with a new module descriptor file.

In other words, it's a "package of Java Packages" abstraction that allows us to make our code even more reusable.

2.1. Packages

The packages inside a module are identical to the Java packages we've been using since the inception of Java.

When we create a module, **we organize the code internally in packages just like we previously did with any other project.**

Aside from organizing our code, packages are used to determine what code is publicly accessible outside of the module. We'll spend more time talking about this later in the article.

2.2. Resources

Each module is responsible for its resources, like media or configuration files.

Previously we'd put all resources into the root level of our project and manually manage which resources belonged to different parts of the application.

With modules, we can ship required images and XML files with the module that needs it, making our projects much easier to manage.

2.3. Module Descriptor

When we create a module, we include a descriptor file that defines several aspects of our new module:

- **Name** – the name of our module
- **Dependencies** – a list of other modules that this module depends on
- **Public Packages** – a list of all packages we want accessible from outside the module
- **Services Offered** – we can provide service implementations that can be consumed by other modules
- **Services Consumed** – allows the current module to be a consumer of a service
- **Reflection Permissions** – explicitly allows other classes to use reflection to access the private members of a package

The module naming rules are similar to how we name packages (dots are allowed, dashes are not). It's very common to do either project-style (my.module) or Reverse-DNS (*com.baeldung.mymodule*) style names. We'll

use project-style in this guide. (/)

We need to list all packages we want to be public because by default all packages are module private.

The same is true for reflection. By default, we cannot use reflection on classes we import from another module.

Later in the article, we'll look at examples of how to use the module descriptor file.

2.4. Module Types

There are four types of modules in the new module system:

- **System Modules** – These are the modules listed when we run the *list-modules* command above. They include the Java SE and JDK modules.
- **Application Modules** – These modules are what we usually want to build when we decide to use Modules. They are named and defined in the compiled *module-info.class* file included in the assembled JAR.
- **Automatic Modules** – We can include unofficial modules by adding existing JAR files to the module path. The name of the module will be derived from the name of the JAR. Automatic modules will have full read access to every other module loaded by the path.
- **Unnamed Module** – When a class or JAR is loaded onto the classpath, but not the module path, it's automatically added to the unnamed

module (/) it's a catch-all module to maintain backward compatibility with previously-written Java code.

2.5. Distribution

Modules can be distributed one of two ways: as a JAR file or as an "exploded" compiled project. This, of course, is the same as any other Java project so it should come as no surprise.

We can create multi-module projects comprised of a "main application" and several library modules.

We have to be careful though because we can only have one module per JAR file.

When we set up our build file, we need to make sure to bundle each module in our project as a separate jar.

3. Default Modules

When we install Java 9, we can see that the JDK now has a new structure.

They have taken all the original packages and moved them into the new module system.

We can see what these modules are by typing into the command line:

```
java --list-modules                    (/)
```

These modules are split into four major groups: *java*, *javafx*, *jdk*, and *Oracle*.

java modules are the implementation classes for the core SE Language Specification.

javafx modules are the FX UI libraries.

Anything needed by the JDK itself is kept in the *jdk* modules.

And finally, **anything that is Oracle-specific is in the *oracle* modules.**

4. Module Declarations

To set up a module, we need to put a special file at the root of our packages named *module-info.java*.

This file is known as the module descriptor and contains all of the data needed to build and use our new module.

We construct the module with a declaration whose body is either empty or made up of module directives:

```
module myModuleName {
    // all directives are optional
}
```

We start the module declaration with the *module* keyword, and we follow that with the name of the module.

The module will work with this declaration, but we'll commonly need more information.

That is where the module directives come in.

4.1. Requires

Our first directive is *requires*. This module directive allows us to declare module dependencies:

```
module my.module {
    requires module.name;
}
```

Now, *my.module* has **both a runtime and a compile-time dependency** on *module.name*.

And all public types exported from a dependency are accessible by our module when we use this directive.

4.2. Requires Static

Sometimes we write code that references another module, but that users of our library will never want to use.

For instance, we might write a utility function that pretty-prints our internal state when another logging module is present. But, not every consumer of our library will want this functionality, and they don't want to include an extra logging library.

In these cases, we want to use an optional dependency. By using the *requires static* directive, we create a compile-time-only dependency:

```
module my.module {
    requires static module.name;
}
```

4.3. Requires Transitive

We commonly work with libraries to make our lives easier.

But, we need to make sure that any module that brings in our code will also bring in these extra 'transitive' dependencies or they won't work.

Luckily, we can use the *requires transitive* directive to force any downstream consumers also to read our required dependencies:

```
module my.module {
    requires transitive module.name;
}
```

Now, when a developer *requires my.module*, they won't also have also to say *requires module.name* for our module to still work.

4.4. Exports

By default, a module doesn't expose any of its API to other modules. This *strong encapsulation* was one of the key motivators for creating the module system in the first place.

(/)

Our code is significantly more secure, but now we need to explicitly open our API up to the world if we want it to be usable.

We use the *exports* directive to expose all public members of the named package:

```
module my.module {
    exports com.my.package.name;
}
```

Now, when someone does *requires my.module*, they will have access to the public types in our *com.my.package.name* package, but not any other package.

4.5. Exports … To

We can use *exports…to* to open up our public classes to the world.

But, what if we don't want the entire world to access our API?

We can restrict which modules have access to our APIs using the *exports… to* directive.

Similar to the *exports* directive, we declare a package as exported. But, we also list which modules we are allowing to import this package as a *requires*. Let's see what this looks like:

```
module my.module {
    export com.my.package.name to com.specific.package;
}
```

4.6. Uses

A *service* is an implementation of a specific interface or abstract class that can be *consumed* by other classes.

We designate the services our module consumes with the *uses* directive.

Note that **the class name we *use* is either the interface or abstract class of the service, not the implementation class**:

```
module my.module {
    uses class.name;
}
```

We should note here that there's a difference between a *requires* directive and the *uses* directive.

We might *require* a module that provides a service we want to consume, but that service implements an interface from one of its transitive dependencies.

Instead of forcing our module to require *all* transitive dependencies just in case, we use the *uses* directive to add the required interface to the module path.

4.7. Provides … With

A module can also be a *service provider* that other modules can consume.

The first part of the directive is the *provides* keyword. Here is where we put the interface or abstract class name.

Next, we have the *with* directive where we provide the implementation class name that either *implements* the interface or *extends* the abstract class.

Here's what it looks like put together:

```
module my.module {
    provides MyInterface with MyInterfaceImpl;
}
```

4.8. Open

We mentioned earlier that encapsulation was a driving motivator for the design of this module system.

Before Java 9, it was possible to use reflection to examine every type and member in a package, even the *private* ones. Nothing was truly encapsulated, which can open up all kinds of problems for developers of the libraries.

Because Java 9 enforces *strong encapsulation*, **we now have to explicitly grant permission for other modules to reflect on our classes.**

If we want to continue to allow full reflection as older versions of Java did, we can simply *open* the entire module up:

```
open module my.module {
}
```

4.9. Opens

If we need to allow reflection of private types, but we don't want all of our code exposed, **we can use the *opens* directive to expose specific packages.**

But remember, this will open the package up to the entire world, so make sure that is what you want:

```
module my.module {
  opens com.my.package;
}
```


4.10. Opens … To

Okay, so reflection is great sometimes, but we still want as much security as we can get from *encapsulation*. **We can selectively open our packages to a pre-approved list of modules, in this case, using the *opens…to* directive**:

```
module my.module {
    opens com.my.package to moduleOne, moduleTwo, etc.;
}
```


5. Command Line Options

By now, support for Java 9 modules has been added to Maven and Gradle, so you won't need to do a lot of manual building of your projects. However, it's still valuable to know *how* to use the module system from the command line.

We'll be using the command line for our full example down below to help solidify how the entire system works in our minds.

- ***module-path* –** We use the *–module-path* option to specify the module path. This is a list of one or more directories that contain your modules.
- ***add-reads*** – Instead of relying on the module declaration file, we can use the command line equivalent of the *requires* directive; *–add-reads*.

- ***add-exports –*** Command line replacement for the *exports* directive.
- ***add-opens*** – Replace the *open* clause in the module declaration file.
- ***add-modules*** – Adds the list of modules into the default set of modules
- ***list-modules*** – Prints a list of all modules and their version strings
- ***patch-module*** – Add or override classes in a modules
- ***illegal-access=permit|warn|deny*** – Either relax strong encapsulation by showing a single global warning, shows every warning, or fails with errors. The default is *permit*.

6. Visibility

We should spend a little time talking about the visibility of our code.

A lot of libraries depend on reflection to work their magic (JUnit and Spring come to mind).

By default in Java 9, we will *only* have access to public classes, methods, and fields in our exported packages. Even if we use reflection to get access to non-public members and call *setAccessible(true),* we won't be able to access these members.

We can use the *open*, *opens*, and *opens…to* options to grant runtime-only access for reflection. Note, **this is runtime-only!**

We won't be able to compile against private types, and we should never need to anyway.

If we must have access to a module for reflection, and we're not the owner of that module (i.e., we can't use the *opens…to* directive), then it's possible to use the command line *–add-opens* option to allow own modules reflection access to the locked down module at runtime.

(/)

The only caveat here's that you need to have access to the command line arguments that are used to run a module for this to work.

7. Putting It All Together

Now that we know what a module is and how to use them let's go ahead and build a simple project to demonstrate all the concepts we just learned.

To keep things simple, we won't be using Maven or Gradle. Instead, we'll rely on the command line tools to build our modules.

7.1. Setting Up Our Project

First, we need to set up our project structure. We'll create several directories to organize our files.

Start by creating the project folder:

```
mkdir module-project
cd module-project
```

This is the base of our whole project, so add files in here such as Maven or Gradle build files, other source directories, and resources.

We also put a directory to hold all our project specific modules.

Next, we create a module directory:

```
mkdir simple-modules
```

Here's what our project structure will look like:

```
module-project
|- // src if we use the default package
|- // build files also go at this level
|- simple-modules
  |- hello.modules
    |- com
      |- baeldung
        |- modules
          |- hello
  |- main.app
    |- com
      |- baeldung
        |- modules
          |- main
```

7.2. Our First Module

Now that we have the basic structure in place, let's add our first module.

Under the *simple-modules* directory, create a new directory called *hello.modules*.

We can name this anything we want but follow package naming rules (i.e., periods to separate words, etc.). We can even use the name of our main package as the module name if we want, but usually, we want to stick to the same name we would use to create a JAR of this module.

Under our new module, we can create the packages we want. In our case, we are going to create one package structure:

```
com.baeldung.modules.hello
```

Next, create a new class called *HelloModules.java* in this package. We will keep the code simple:

```
package com.baeldung.modules.hello;

public class HelloModules {
    public static void doSomething() {
        System.out.println("Hello, Modules!");
    }
}
```

And finally, in the *hello.modules* root directory, add in our module descriptor; *module-info.java*:

```
module hello.modules {
    exports com.baeldung.modules.hello;
}
```


To keep this example simple, all we are doing is exporting all public members of the *com.baeldung.modules.hello* package.

(/)

7.3. Our Second Module

Our first module is great, but it doesn't do anything.

We can create a second module that uses it now.

Under our *simple-modules* directory, create another module directory called *main.app*. We are going to start with the module descriptor this time:

```
module main.app {
    requires hello.modules;
}
```

We don't need to expose anything to the outside world. Instead, all we need to do is depend on our first module, so we have access to the public classes it exports.

Now we can create an application that uses it.

Create a new package structure: *com.baeldung.modules.main*.

Now, create a new class file called *MainApp.java.*

```
package com.baeldung.modules.main;

import com.baeldung.modules.hello.HelloModules;

public class MainApp {
    public static void main(String[] args) {
        HelloModules.doSomething();
    }
}
```

And that is all the code we need to demonstrate modules. Our next step is to build and run this code from the command line.

7.4. Building Our Modules

To build our project, we can create a simple bash script and place it at the root of our project.

Create a file called *compile-simple-modules.sh*:

```
#!/usr/bin/env bash
javac -d outDir --module-source-path simple-modules $(find simple-
modules -name "*.java")
```

There are two parts to this command, the *javac* and *find* commands.

The *find* command is simply outputting a list of all *.java* files under our simple-modules directory. We can then feed that list directly into the Java compiler.

The only thing we have to do differently than the older versions of Java is to provide a *module-source-path* parameter to inform the compiler that it's building modules.

Once we run this command, we will have an *outDir* folder with two compiled modules inside.

7.5. Running Our Code

And now we can finally run our code to verify modules are working correctly.

Create another file in the root of the project: *run-simple-module-app.sh*.

```
#!/usr/bin/env bash
java --module-path outDir -m
main.app/com.baeldung.modules.main.MainApp
```

To run a module, we must provide at least the *module-path* and the main class. If all works, you should see:

```
>$ ./run-simple-module-app.sh
Hello, Modules!
```


7.6. Adding a Service

Now that we have a basic understanding of how to build a module, let's make it a little more complicated.

We're going to see how to use the *provides…with* and *uses* directives.

Start by defining a new file in the *hello.modules* module named *HelloInterface.java*:

```
public interface HelloInterface {
    void sayHello();
}
```

To make things easy, we're going to implement this interface with our existing *HelloModules.java* class:

```
public class HelloModules implements HelloInterface {
    public static void doSomething() {
        System.out.println("Hello, Modules!");
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}
```

That is all we need to do to create a *service*.

Now, we need to tell the world that our module provides this service.

Add the following to our *module-info.java*:

```
provides com.baeldung.modules.hello.HelloInterface with
com.baeldung.modules.hello.HelloModules;
```

As we can see, we declare the interface and which class implements it.

Next, we need to consume this *service*. In our *main.app* module, let's add the following to our *module-info.java*:

```
uses com.baeldung.modules.hello.HelloInterface;
```


Finally, in our main method we can use this service via a ServiceLoader (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ServiceLoader.html):

```
Iterable<HelloInterface> services =
ServiceLoader.load(HelloInterface.class);
HelloInterface service = services.iterator().next();
service.sayHello();
```

Compile and run:

```
#> ./run-simple-module-app.sh
Hello, Modules!
Hello!
```

We use these directives to be much more explicit about how our code is to be used.

We could put the implementation into a private package while exposing the interface in a public package.

This makes our code much more secure with very little extra overhead.

Go ahead and try out some of the other directives to learn more about modules and how they work.

8. Adding Modules to the Unnamed Module

The unnamed module concept is similar to the default package. Therefore, it's not considered a real module, but can be viewed as the default module.

If a class is not a member of a named module, then it will be automatically considered as part of this unnamed module.

Sometimes, to ensure specific platform, library, or service-provider modules in the module graph, we need to add modules to the default root set. For example, when we try to run Java 8 programs as-is with Java 9 compiler we may need to add modules.

In general, **the option to add the named modules to the default set of root modules is *–add-modules <module>*(*,<module>)** ** where *<module>* is a module name.

For example, to provide access to all *java.xml.bind* modules the syntax would be:

```
add-modules java.xml.bind (/)
```

To use this in Maven, we can embed the same to the *maven-compiler-plugin*:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.0</version>
    <configuration>
        <source>9</source>
        <target>9</target>
        <compilerArgs>
            <arg>--add-modules</arg>
            <arg>java.xml.bind</arg>
        </compilerArgs>
    </configuration>
</plugin>
```

9. Conclusion

In this extensive guide, we focused on and covered the basics of the new Java 9 Module system.

We started by talking about what a module is.

Next, we talked about how to discover which modules are included in the JDK.

We also covered the module declaration file in detail.

We rounded out the theory by talking about the various command line arguments we'll need to build our modules.

Finally, we put all our previous knowledge into practice and created a simple application built on top of the module system.

To see this code and more, be sure to check it out over on Github (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-9-jigsaw).

Comments are closed on this article!

(/)

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)