Java Modules Tutorial

**JPMS (Java Platform Module System)** is a significant enhancement in Java 9. It is also known as Project Jigsaw. In this Java 9 modules example, we will learn about modules (in general) and how your programming style will change in the future when you start writing modular code.

**1. What is a Module?**

In any programming language, modules are (package-like) artifacts containing code, with metadata describing the module and its relation to other modules. Ideally, these artifacts are recognizable from compile-time all the way through runtime. Any application generally is a combination of multiple modules which work together to perform the business objectives.

In terms of application architecture, a module shall represent a specific business capability. It should be self-sufficient for that capability and should expose only interfaces to use the module functionality. To complete its tasks, it may be dependent on other modules, which it should declare explicitly.

AD

So, in short, a module should adhere to **three core principles** –

**1.1. Strong Encapsulation**

Encapsulation means hiding implementation details, which are not essential to know to use the module correctly. The purpose is that encapsulated code may change freely without affecting users of the module.

**1.2. Stable Abstraction**

Abstraction helps to expose module functionality using interfaces, i.e., public APIs. Anytime you want to change the business logic or implementation inside module code, changes will be transparent to the module users.

**1.3. Explicit dependencies**

Modules can be dependent on other modules as well. These external dependencies must be part of the module definition itself.

AD

These dependencies between modules are often represented as graphs. Once you see the graph at the application level, you will better understand the application’s architecture.

AD

**2. Introduction to Java Platform Module System**

**2.1. The Problem**

Before Java 9, we had ‘*packages*‘ to group related classes as per business capabilities. Along with *packages*, we had ‘access modifiers‘ to control what would be visible and what would be hidden to other classes or packages. It has been working great so far.

But, explicit dependencies are where things start to fall apart. In Java, dependencies are declared with ‘*import*‘ statements; but they are strictly ‘compile time’ constructs.

AD

Once code is compiled, there is no mechanism to state its runtime dependencies clearly. In fact, Java runtime dependency resolution is such a problematic area that special tools have been created to fight this problem e.g., gradle or maven.

Also, a few frameworks started bundling their complete runtime dependencies as well e.g., Spring boot projects.

**2.2. How does JPMS Solve the Problem?**

With new **Java 9 modules**, we will have better capability to write well-structured applications. This enhancement is divided into two areas:

1. Modularize the JDK itself.
2. Offer a module system for other applications to use.

**Java Base Module**

Java 9 Module System has a “**java.base**” module. It’s known as **Base Module**. It’s an Independent module and does NOT dependent on any other modules. By default, all other modules are dependent on “**java.base**“.

In Java 9, modules help us in encapsulating packages and managing dependencies. So typically,

AD

* a class is a container of fields and methods
* a package is a container of classes and interfaces
* **a module is a container of packages**

**2.3. Difference between modular and non-modular code**

You will not feel any significant difference between regularcode and modular code if you don’t know the specific things to look for. e.g.

1. A module is typically just a *jar* file that has a module-info.class file at the root.
2. To use a module, include the *jar* file into modulepath instead of the classpath. A modular jar file added to classpath is normal jar file and module-info.class file will be ignored.

**3. How to Write Modular Code**

After reading all the above concepts, let’s see how modular code is written in reality.

**3.1. Create a new Java Modular Project**

Create a new modular project. I have created with the name JavaAppOne.

Create Java Modular Project

Create Java Modular Project – Step 2

**3.2. Create Java Modules**

Now add one or two modules to this project.

AD

Create New Module

I have added two modules helloworld and test. Let’s see their code and project structure.

Java 9 Modules Project Structure

/helloworld/module-info.java

**module** helloworld {}

HelloWorldApp.java

**package** com.module.demo;

**public** **class** HelloWorldApp {

**public** **static** **void** sayHello() {

System.out.println("Hello from HelloWorldApp");

}

}

/test/module-info.java

**module** test {

}

TestApp.java

**package** com.test;

**public** **class** TestApp {

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

*//some code*

}

}

So far, modules are independent.

Now suppose, we want to use HelloWorldApp.sayHello() method in TestApp class. If we try to use the class without importing the module, we will get compile-time error “package com.module.demo is not visible”.

**3.3. Export Packages and Import Module**

To be able to import HelloWorldApp, you must first export ‘com.howtodoinjava.demo’ package from helloworld module and then include helloworld module in test module.

**module** helloworld {

**exports** com.module.demo;

}

**module** test {

**requires** helloworld;

}

In the above code, requires keyword indicates a dependency, and exports keyword identifies the packages which are available to be exported to other modules.

Only when a package is explicitly exported, it can be accessed from other modules. Packages inside a module that are not exported, are inaccessible from other modules by default.

Now you will be able to use HelloWorldApp class inside TestApp class.

**package** com.test;

**import** com.module.demo.HelloWorldApp;

**public** **class** TestApp {

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

HelloWorldApp.sayHello();

}

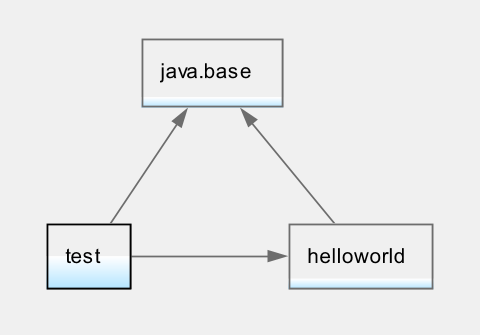
}

Output

Hello from HelloWorldApp

Lets look at the modules graph a well.

AD



Module Graph

**Info**

From Java 9 onwards, public means public only to all other packages inside that module. Only when the package containing the public type is exported, can it be used by other modules.

**4. Conclusion**

Modular applications have many advantages, which you appreciate even more when you come across applications having a non-modular codebase.

You must have heard terms like “*spaghetti architecture*” or “*messy monolith*“. Modularity is not a silver bullet, but it is an architectural principle that can prevent these problems to a high degree when applied correctly.

With JPMS, Java has taken a big step to be a modular language. Whether it is right or wrong decision, only time will tell. It will be interesting to see, how 3rd party libraries and frameworks adapt and use the module system. And how it will impact the development work, we do everyday.

### ****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.

### ****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.

**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.

**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**;

}

### ****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.

**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;

}

**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** {

}

### ****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**;

}

**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.;

}