HowToDoInJava

Java Modules Tutorial

Last Updated: January 25, 2022 By: Lokesh Gupta Java 9 Java Basics, Java Modules

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.

Table Of Contents

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.

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.

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.

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.

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,

- 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 regular code and modular code if you don't know the specific things to look for. e.g.

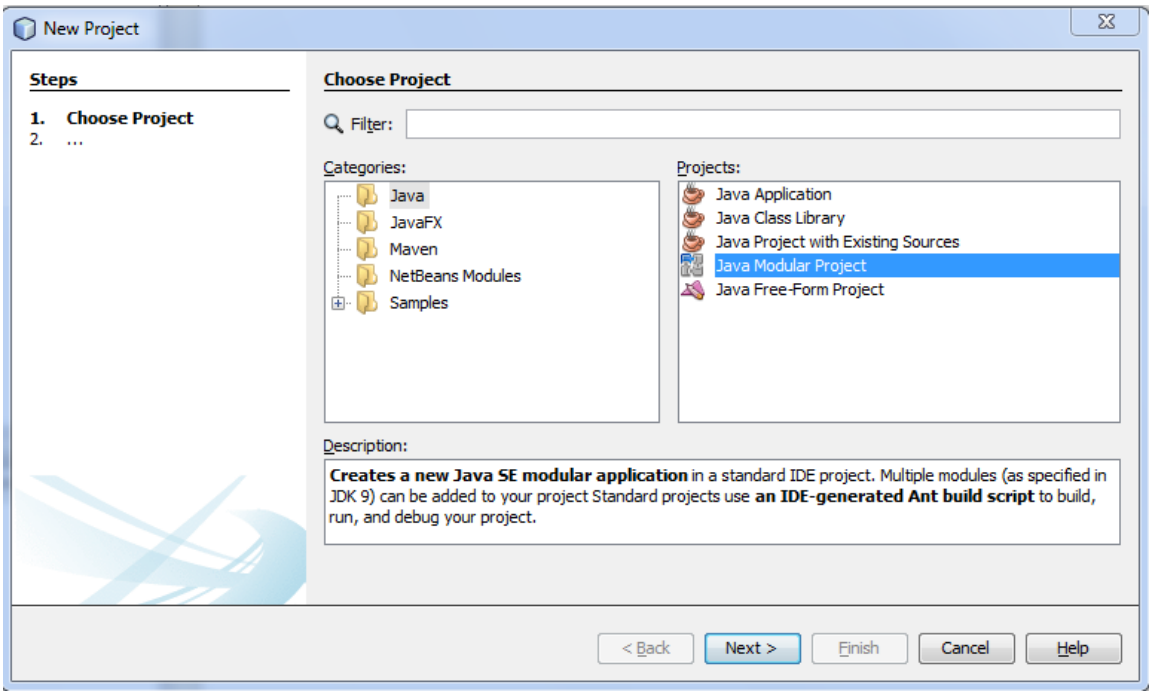
1. A module is typically just a *jar* file that has a `module-info.class` file at the root.
2. To use a module, include the *jar* file into `modulepath` instead of the `classpath`. A modular jar file added to classpath is normal jar file and `module-info.class` file will be ignored.

3. How to Write Modular Code

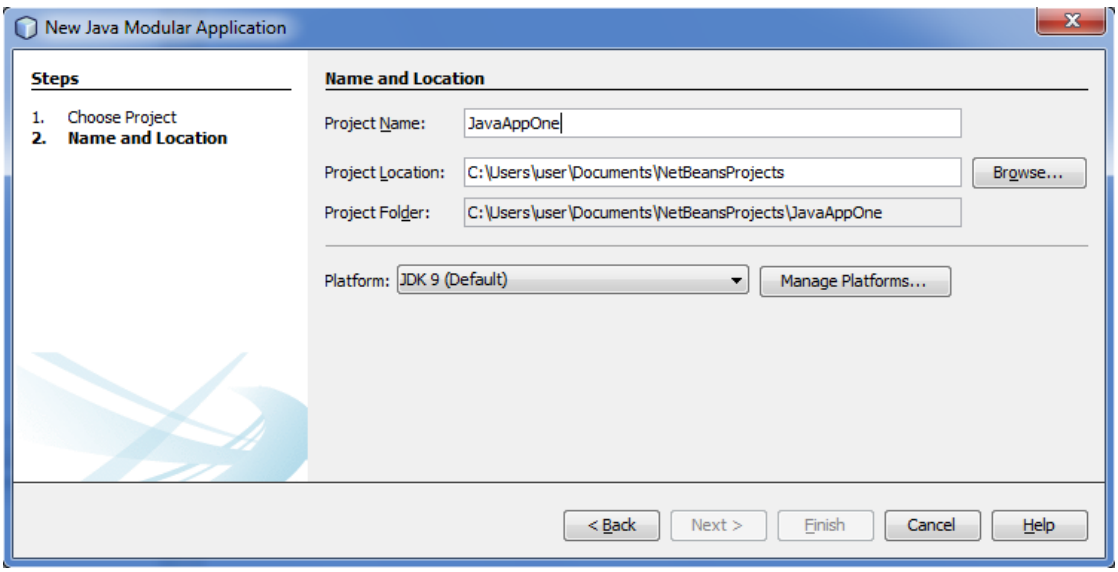
After reading all the above concepts, let's see how modular code is written in reality.

3.1. Create a new Java Modular Project

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



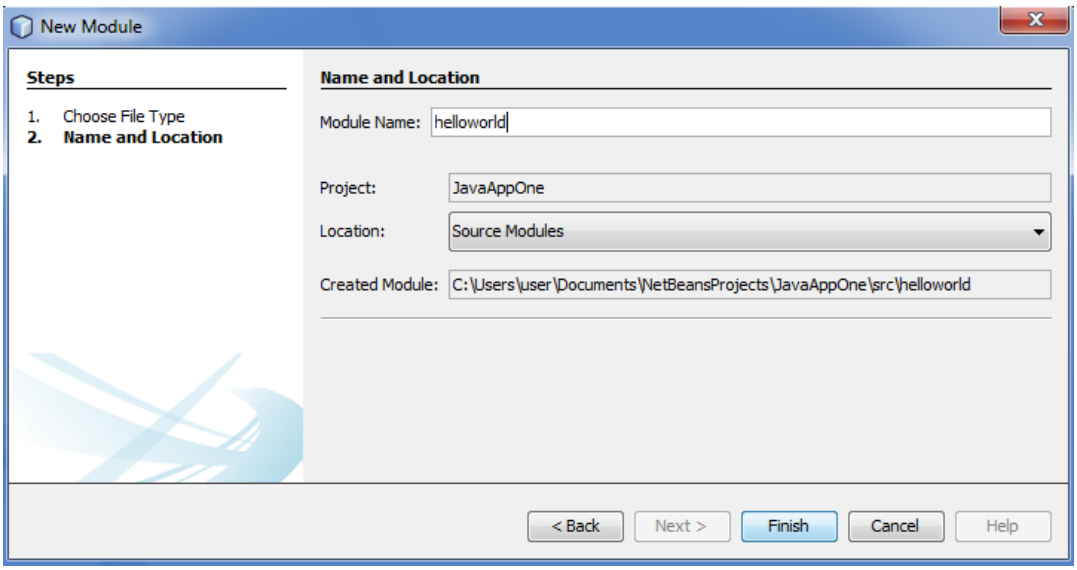
Create Java Modular Project



Create Java Modular Project – Step 2

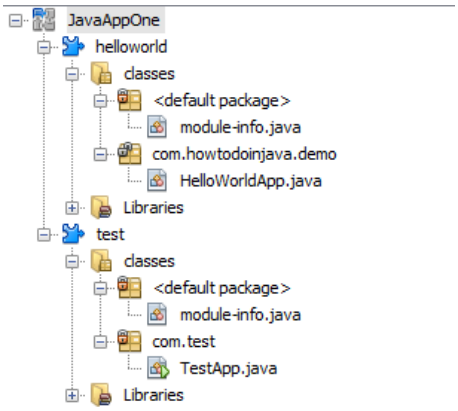
3.2. Create Java Modules

Now add one or two modules to this project.



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.howtodoinjava.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.howtodoinjava.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.howtodoinjava.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;
```

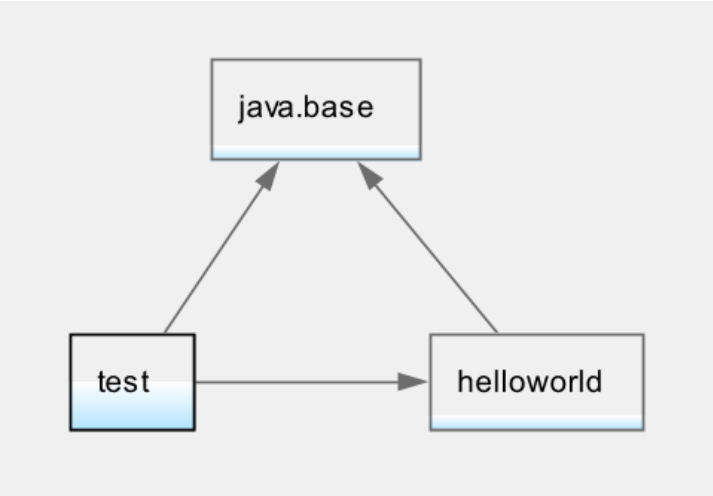
```
import com.howtodoinjava.demo.HelloWorldApp;

public class TestApp {
    public static void main(String[] args) {
        HelloWorldApp.sayHello();
    }
}
```

Output

```
Hello from HelloWorldApp
```

Lets look at the modules graph a well.



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.

Happy Learning !!

Sourcecode

Sourcecode Download

Recommended Reading:

1. JMS Tutorial – Java Message Service Tutorial
2. Java Tutorial
3. How to Use Locks in Java | java.util.concurrent.locks.Lock Tutorial and Example
4. Java enum tutorial
5. Java executor framework tutorial and best practices
6. Complete Java Annotations Tutorial
7. Java Generics Tutorial
8. Read write CSV file in Java – OpenCSV tutorial
9. Complete Java Servlets Tutorial
0. Java WatchService API Tutorial

Subscribe

* We do not spam !!

1 thought on “Java Modules Tutorial”

Viswanath

August 28, 2017 at 11:30 am

Hi Lokesh,

Nice to see that you have started to come up with the tutorials on Java 9.
The tutorial is good as a starter.
I have not downloaded Netbeans for JDK9 support will give it a try soon.
I was just going through the example to get the concept.
I would suggest you to elaborate a bit more as how to export Module and import packages.

Reply

Leave a Comment

Name *

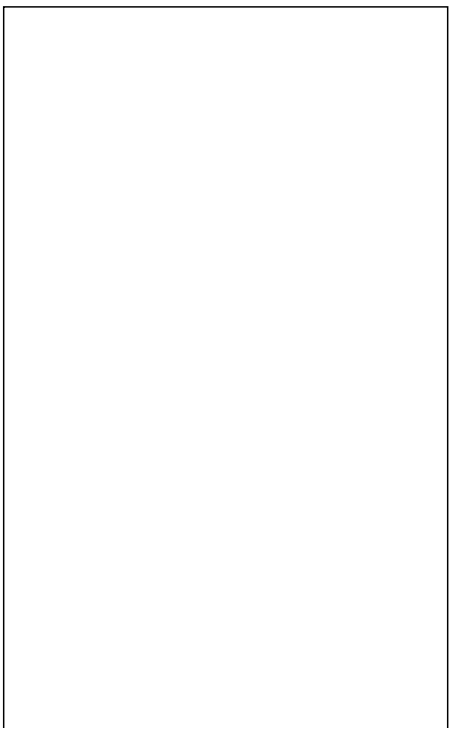
Email *

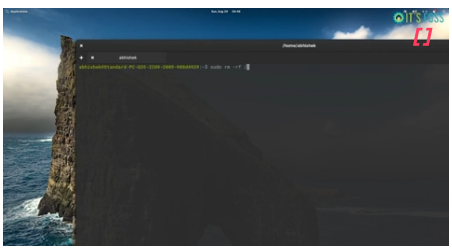
Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search …





HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial