Name of the Course : LEARNING JAVA 9 – MODULAR PROGRAMMING

Level : High

Tool Stack : Java 9

Problem Statement : Project Jigsaw is an umbrella project with the new features aimed at two aspects:

The introduction of module system in the Java language

and its implementation in JDK source and Java runtime

In this article, we'll introduce you to the Jigsaw project and its features and finally wrap it up with a simple modular application.

Step 1 : Let us create a simple modular application with modules and their dependencies as

1. Overview

The Java Platform Module System (JPMS) adds more reliability, better separation of concerns, and stronger encapsulation to Java applications. However, it's not a build tool, hence it lacks the ability for automatically managing project dependencies.

Of course, we may wonder whether if we can use well-established build tools, like Maven or Gradle, in modularized applications.

Actually, we can! In this tutorial, we'll learn how to create a multi-module Maven application using Java modules.

2. Encapsulating Maven Modules in Java Modules

Since modularity and dependency management are not mutually exclusive concepts in Java, we can seamlessly integrate the JPMS, for instance, with Maven, thus leveraging the best of both worlds.

In a standard multi-module Maven project, we add one or more child Maven modules by placing them under the project's root folder and declaring them in the parent POM, within the <modules> section.

In turn, we edit each child module's POM and specify its dependencies via the standard <groupId>, <artifactId> and <version> coordinates.

The reactor mechanism in Maven — responsible for handling multi-module projects — takes care of building the whole project in the right order.

In this case, we'll basically use the same design methodology, but with one subtle yet fundamental variant: we'll wrap each Maven module into a Java module by adding to it the module descriptor file, module-info.java.

3. The Parent Maven Module

To demonstrate how modularity and dependency management work great together, we'll build a basic demo multi-module Maven project, whose functionality will be narrowed to just fetching some domain objects from a persistence layer.

To keep the code simple, we'll use a plain Map as the underlying data structure for storing the domain objects. Of course, we can easily switch further down the road to a fully-fledged relational database.

Let's start by defining the parent Maven module. To accomplish this, let's create a root project directory called, for instance, multimodulemavenproject (but it could be anything else), and add to it the parent pom.xml file:

<groupId>com.iiht.java9.multimodulemavenproject</groupId>

<artifactId>multimodulemavenproject</artifactId>

<version>1.0</version>

<packaging>pom</packaging>

<name>multimodulemavenproject</name>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.0</version>

<configuration>

<source>11</source>

<target>11</target>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

There are a few details worth noting in the definition of the parent POM.

First off, since we're using Java 11, we'll need at least Maven 3.5.0 on our system, as Maven supports Java 9 and higher from that version onward.

And, we'll also need at least version 3.8.0 of the Maven compiler plugin. Therefore, let's make sure to check the latest version of the plugin on Maven Central.

4. The Child Maven Modules

Notice that up to this point, the parent POM doesn't declare any child modules.

Since our demo project will fetch some domain objects from the persistence layer, we'll create four child Maven modules:

entitymodule: will contain a simple domain class

daomodule: will hold the interface required for accessing the persistence layer (a basic DAO contract)

userdaomodule: will include an implementation of the daomodule‘s interface

mainappmodule: the project's entry point

4.1. The entitymodule Maven Module

Now, let's add the first child Maven module, which just includes a basic domain class.

Under the project's root directory, let's create the entitymodule/src/main/java/com/iiht/entity directory structure and add a User class:

public class User {

private final String name;

// standard constructor / getter / toString

}

Next, let's include the module's pom.xml file:

<parent>

<groupId>com.iiht.java9.multimodulemavenproject</groupId>

<artifactId>multimodulemavenproject</artifactId>

<version>1.0</version>

</parent>

<groupId>com.iiht.java9.entitymodule</groupId>

<artifactId>entitymodule</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<name>entitymodule</name>

As we can see, the Entity module doesn't have any dependencies to other modules, nor does it require additional Maven artifacts, as it only includes the User class.

Now, we need to encapsulate the Maven module into a Java module. To achieve this, let's simply place the following module descriptor file (module-info.java) under the entitymodule/src/main/java directory:

module com.iiht.java9.entitymodule {

exports com.iiht.java9.entitymodule;

}

Finally, let's add the child Maven module to the parent POM:

<modules>

<module>entitymodule</module>

</modules>

4.2. The daomodule Maven Module

Let's create a new Maven module that will contain a simple interface. This is convenient for defining an abstract contract for fetching generic types from the persistence layer.

As a matter of fact, there's a very compelling reason to place this interface in a separate Java module. By doing so, we have an abstract, highly-decoupled contract, which is easy to reuse in different contexts. At the core, this is an alternative implementation of the Dependency Inversion Principle, which yields a more flexible design.

Therefore, let's create the daomodule/src/main/java/com/iiht/dao directory structure under the project's root directory, and add to it the Dao<T> interface:

public interface Dao<T> {

Optional<T> findById(int id);

List<T> findAll();

}

Now, let's define the module's pom.xml file:

<parent>

// parent coordinates

</parent>

<groupId>com.iiht.java9.daomodule</groupId>

<artifactId>daomodule</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<name>daomodule</name>

The new module doesn't require other modules or artifacts either, so we'll just wrap it up into a Java module. Let's create the module descriptor under the daomodule/src/main/java directory:

module com.iiht.java9.daomodule {

exports com.iiht.java9.daomodule;

}

Finally, let's add the module to the parent POM:

<modules>

<module>entitymodule</module>

<module>daomodule</module>

</modules>

4.3. The userdaomodule Maven Module

Next, let's define the Maven module that holds an implementation of the Dao interface.

Under the project's root directory, let's create the userdaomodule/src/main/java/com/iiht/userdao directory structure, and add to it the following UserDao class:

public class UserDao implements Dao<User> {

private final Map<Integer, User> users;

// standard constructor

@Override

public Optional<User> findById(int id) {

return Optional.ofNullable(users.get(id));

}

@Override

public List<User> findAll() {

return new ArrayList<>(users.values());

}

}

Simply put, the UserDao class provides a basic API that allows us to fetch User objects from the persistence layer.

To keep things simple, we used a Map as the backing data structure for persisting the domain objects. Of course, it's possible to provide a more thorough implementation that uses, for instance, Hibernate's entity manager.

Now, let's define the Maven module's POM:

<parent>

// parent coordinates

</parent>

<groupId>com.iiht.java9.userdaomodule</groupId>

<artifactId>userdaomodule</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<name>userdaomodule</name>

<dependencies>

<dependency>

<groupId>com.iiht.java9.entitymodule</groupId>

<artifactId>entitymodule</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>com.iiht.java9.daomodule</groupId>

<artifactId>daomodule</artifactId>

<version>1.0</version>

</dependency>

</dependencies>

In this case, things are slightly different, as the userdaomodule module requires the entitymodule and daomodule modules. That's why we added them as dependencies in the pom.xml file.

We still need to encapsulate this Maven module into a Java module. So, let's add the following module descriptor under the userdaomodule/src/main/java directory:

module com.iiht.java9.userdaomodule {

requires com.iiht.java9.entitymodule;

requires com.iiht.java9.daomodule;

provides com.iiht.java9.daomodule.Dao with com.iiht.java9.userdaomodule.UserDao;

exports com.iiht.java9.userdaomodule;

}

Finally, we need to add this new module to the parent POM:

<modules>

<module>entitymodule</module>

<module>daomodule</module>

<module>userdaomodule</module>

</modules>

From a high-level view, it's easy to see that the pom.xml file and the module descriptor play different roles. Even so, they complement each other nicely.

Let's say that we need to update the versions of the entitymodule and daomodule Maven artifacts. We can easily do this without having to change the dependencies in the module descriptor. Maven will take care of including the right artifacts for us.

Similarly, we can change the service implementation that the module provides by modifying the “provides..with” directive in the module descriptor.

We gain a lot when we use Maven and Java modules together. The former brings the functionality of automatic, centralized dependency management, while the latter provides the intrinsic benefits of modularity.

4.4. The mainappmodule Maven Module

Additionally, we need to define the Maven module that contains the project's main class.

As we did before, let's create the mainappmodule/src/main/java/mainapp directory structure under the root directory, and add to it the following Application class:

public class Application {

public static void main(String[] args) {

Map<Integer, User> users = new HashMap<>();

users.put(1, new User("Julie"));

users.put(2, new User("David"));

Dao userDao = new UserDao(users);

userDao.findAll().forEach(System.out::println);

}

}

The Application class's main() method is quite simple. First, it populates a HashMap with a couple of User objects. Next, it uses a UserDao instance for fetching them from the Map, and then it displays them to the console.

In addition, we also need to define the module's pom.xml file:

<parent>

// parent coordinates

</parent>

<groupId>com.iiht.java9.mainappmodule</groupId>

<artifactId>mainappmodule</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<name>mainappmodule</name>

<dependencies>

<dependency>

<groupId>com.iiht.java9.entitymodule</groupId>

<artifactId>entitymodule</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>com.iiht.java9.daomodule</groupId>

<artifactId>daomodule</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>com.iiht.java9.userdaomodule</groupId>

<artifactId>userdaomodule</artifactId>

<version>1.0</version>

</dependency>

</dependencies>

The module's dependencies are pretty self-explanatory. So, we just need to place the module inside a Java module. Therefore, under the mainappmodule/src/main/java directory structure, let's include the module descriptor:

module com.iiht.java9.mainappmodule {

requires com.iiht.java9.entitypmodule;

requires com.iiht.java9.userdaopmodule;

requires com.iiht.java9.daopmodule;

uses com.iiht.java9.daopmodule.Dao;

}

Finally, let's add this module to the parent POM:

<modules>

<module>entitymodule</module>

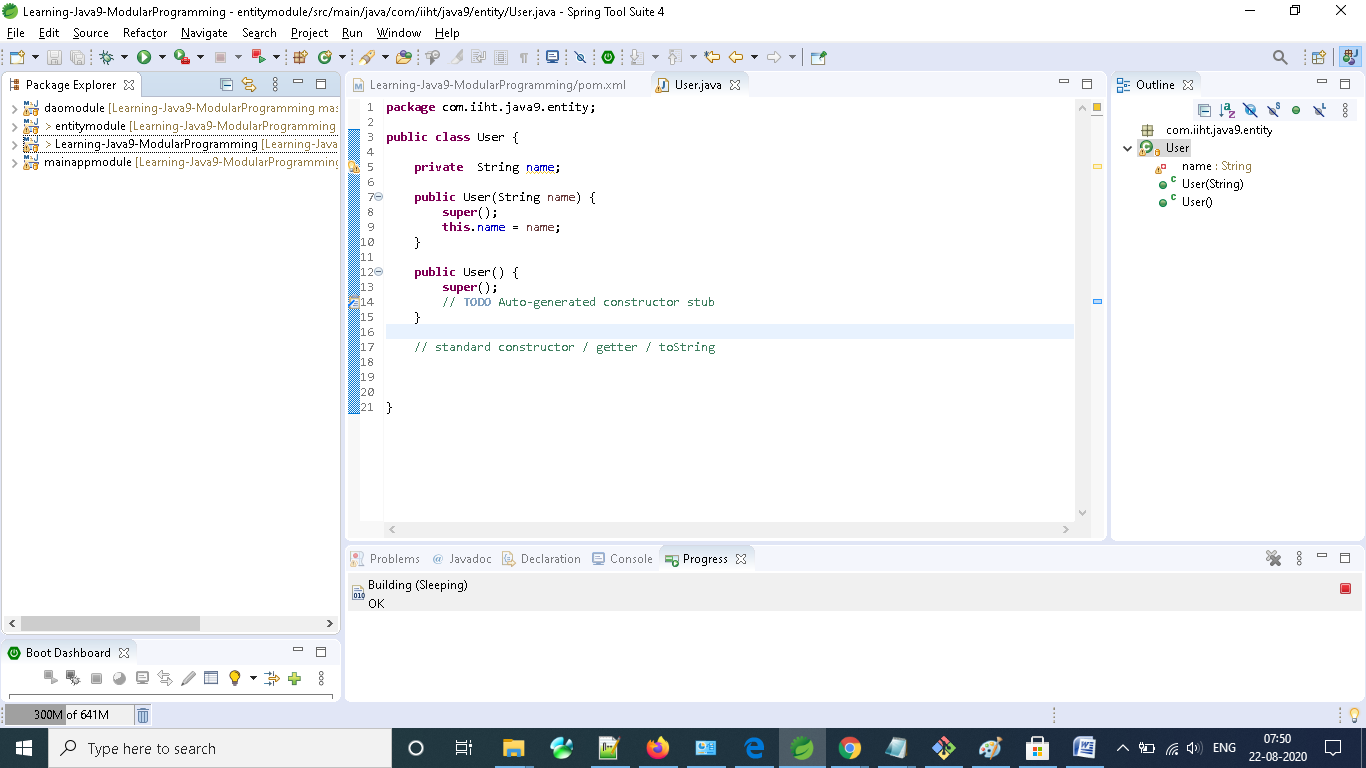
<module>daomodule</module>

<module>userdaomodule</module>

<module>mainappmodule</module>

</modules>

With all the child Maven modules already in place, and neatly encapsulated in Java modules, here's how the project's structure looks:



5. Running the Application

Finally, let's run the application, either from within our IDE or from a console.

As we might expect, we should see a couple of User objects printed out to the console when the application starts up:

User{name=Julie}

User{name=David}

6. Conclusion

In this tutorial, we learned in a pragmatic way how to put Maven and the JPMS to work side-by-side, in the development of a basic multi-module Maven project that uses Java modules.

Learning outcome

In this article, we looked at creating, compiling and running a simple modular application. We also saw how the JDK source code had been modularized.