Introduction to the Java Platform Module System

Leurs, Johannes

Jordan, Felix

Supervisor: Dollase, Stefan

April 26, 2019

Abstract

Before Java 9, it was not possible to hide internal packages of libraries and applications. To fix this, the Java Platform Module System was introduced to bundle different packages together into modules. Every module uses a module declaration to define which packages are available to other modules and which dependencies are required for the module to work. This allows Java to check accessibility rules for modules at compile and run time as well as their presence. Furthermore, the module system limits reflections usage on modules, includes support for the ServiceLoader API and allows to build custom java run time images.

1 Introduction

Organizing big software projects is often a challenge because developers have to manage complex systems. This challenge is usually approached by breaking such a system down into smaller, logical independent parts. This improves maintainability and reliability, as the system parts may be developed independently. To help developers implement this concept into program code, Java 9 introduces the Java Platform Module System (JPMS), which offers modularization with dependency management and an enhanced approach for strong encapsulation.

First, we discuss what modularization means in general and how to approach it (section 2). After that, we look at the situation before Java 9 (section 3) and which problems arose due to lacking concepts of modularization (section 3.1). Before introducing the approach of Java 9, we look at other existing concepts to bring modularization to Java (section 3.2). Then we take a look at the basic functions of the JPMS needed to build modular Java applications (section 4), before finally introducing advanced programming approaches that the JPMS offers (section 5).

2 Modularization

Before we introduce the Java Platform Module System, it is important to understand what modularization generally is all about.

2.1 What is modularization?

Modularization is a core concept of software engineering. It applies the principle of divide an conquer to the architecture of software. When a complex software system is developed, it is first divided into smaller subsystems, each of them having separate responsibilities. These subsystems can again be divided into smaller self-contained parts until the parts are so small that it is not a hard challenge to handle them. The parts that result from this modularization process are referred to as *modules*. Modules are much less complex than the overall system, which reduces vulnerability and improves maintainability and extensibility of a system [2].

2.2 What is a module?

A module is a part of a system that has a name, a clear definition of its functionality and its dependencies on other modules. Each module consists of a hidden, internal part and an external, visible part. The functionality of a module is specified by a set of operations that form the modules visible interface (API). The invisible, internal part is the actual implementation of the functionality [2].

This abstract concept can be applied to multiple layers of a Java program. On the class level, dependencies are the imported classes and functionality is defined by the classes public methods. Even on the method level, called methods can be interpreted as dependencies of the caller. A methods signature and the internal state the method changes then define its functionality. [5].

3 Situation before Java 9

Before Java 9, modularization was limited to the class and package level. Packages could hide internally shared APIs with classes using package-private visibility. Additional dependency management on a higher level was left to external build tools, so checking of the presence of all dependencies was only possible at compile time [5].

To inform the compiler and the Java Virtual Machine (JVM) where to find additional classes (e.g. application code) besides Java's standard libraries, the *class path* was used. Users may put class and JAR files on the class path to be included. Internally the class path is represented as a list of all classes and JAR files are just seen as containers of classes [5]. At run time, Java uses *class loaders* to resolve class names and load classes into the JVM. The *system class loader* is a predefined class loader that loads classes from the class path. It is also possible to define custom class loader implementations which may load classes from different locations like files or URLs .

3.1 Difficulties

Java loads classes lazily. That means the JVM does not start searching the class path for a specific class until code referencing it is executed for the first time. Because of the class path's inner representation as a list, the JVM ignores which JAR file a class belongs to. That makes intended modularization, by using JARs, undone at run time [5].
Due to that, some problems arose before Java 9: The JVM does not care if there are multiple class files with the same qualified name on the class path, as it finishes its search at the first match. This is called shadowing and can happen when different versions of some library are on the class path, thus only one can really be used. However, if a second version of the library, placed behind the first one, contains a class that the first one does not, the class loader is loading it. Because of this, the same code piece may use parts of both libraries at the same time, which is likely to cause conflicts.
Moreover, missing classes are not recognized at application launch, but only when they are first being requested. Therefore a `ClassNotFoundException`, which may cause the application to crash, might be thrown even after the application has been successfully running for a while. These dependency problems are often described as *dependency hell* or *JAR hell* and lead to further maintenance and security problems. Build tools may be used to help with dependency management [5].

Furthermore, developers may need to create bigger APIs than needed to allow their code to access other code beyond package boundaries. This weakens encapsulation: other developers may not identify project internal APIs as such when they work with these projects and access those APIs. One solution may be to put all project code into one package and declare package internal classes, methods, etc., using the default visibility. But such a workaround would be defying all conventions and give up the benefits of packages.

3.2 Former modularization approaches for Java

To avoid dependency problems, build tools, such as Maven or Gradle, have been offering dependency management for many years. They only assure the presence of all required dependencies at build time, though. Besides that, they may select the version of some dependency based on what versions the dependents are compatible with. As soon as an application is executed and the control is handed over to the JVM, the concept of the build tools' modularization is gone due to the class paths mechanism [5].

Another prominent module system for Java is OSGi, which has been introduced in 1999. It is built on the classloader mechanism of the JVM to offer strong encapsulation between modules, which are called bundles in OSGi. Moreover, OSGi supports the use of multiple different versions of some library at once but has the side effect of introducing more complexity. This may complicate the development process. Also, its strong encapsulation does not prevent *deep reflection* (see 5.3 Reflection) [5].

4 The Java Platform Module System

The Java Platform Module System, introduced in Java 9, provides the benefits of modularization for collections of packages. This helps with application and library development because these are usually shipped as JAR files, which exactly consist of several packages. Using the module system, packages belong to a specific *module*. Thus, APIs can be declared for applications and libraries as a whole and not only for their packages. The *Java Platform, Standard Edition* has been restructured into modules, too. For example, *java.lang*, *java.math*, *java.util* and other packages are grouped into the *java.base* module. Other examples of Java modules, contained in the Standard Edition, are *java.xml* and *java.sql*, which provide additional functionality [5].

4.1 What is a Java Module?

A *Java module* is a JAR file with some additional information: It explicitly defines its dependencies upon other modules and must explicitly define which of its own packages should be visible to other modules. Moreover, Java modules have a name that must be unique in the context of an application [1]. It is recommended to use the reverse internet domain name convention for module names so that they are globally unique (e.g. `com.example.projectname`). In most cases, the name of the package that defines the modules exported API should be used as the module name [4]. However, we are not following this convention in some examples because short names improve the readability and make examples easier to understand [5].

4.2 The Module Declaration

The additional information about a Java Module is defined in the *module declaration*, a file called *module-info.java* [1]. Its content may look like the following:

```
module com.example.projectname {
    requires com.example.library;
    exports com.example.projectname;
}
```

4.2.1 Module Dependencies

For every module that needs to be accessed, the dependency must be explicitly declared using the `requires` keyword. When module A requires module B, then it *reads* module B at run time. The only module that is required implicitly by every module is the `java.base` module because it contains `java.lang`, which is always needed [4]. Thus, the modules dependencie's are exactly the ones denoted in the module declaration. The module system checks at compile and run time whether all modules are available and throws an exception if dependencies are missing [5]. This improves the reliability of an application since missing dependencies are immediately detected at launch time and not when the application is already running.

4.2.2 Exported Packages

Packages that make up the API and therefore are designed to be used by other modules must be *exported* explicitly. In addition, it is also possible to export packages only to specific modules. One has to keep in mind that this may create stronger coupling [5].

```
module com.example.projectname {
    exports com.example.alpha to com.example.other.module;
}
```

If a package is not exported, there is no way to access its classes from another module. This enforces strong encapsulation between modules, which is one of the main goals of the Java Module System [1].

4.3 Accessibility

With the introduction of the module system, rules for class accessibility have been changed. Before, a class needed to be public to be accessible from other packages. These rules remain unchanged within a module, but when a module wants to access classes from another module, the new rules of the module system apply.
To access a class `com.example.Foo` in module B from module A, three conditions must be met [1]:

- Module A has to require module B
- Module B has to export the package `com.example`
- Class `com.example.Foo` has to be public in its package

The visibility rules for fields and methods are the same as in previous Java versions. Thus, to access a field or method of a class in another package, the class itself must be accessible to the module. The accessibility of classes is enforced by the module system at compile and run time. Even by using *reflection* it is, by default, no longer possible to access classes that are not visible [1]. For more details about limitations regarding reflection, see 5.3.

4.4 Compiling and Running

The main difference when compiling and running a modular application is the *module path*, which can be provided as an argument. Similar to the class path the module path is used to load classes and resources from the file system. It is a collection of all modules that an application needs to compile or run [3].
Compilation of a modularized Java project works almost the same way as before:

```
$ javac -d classes-dir --module-path modules ${source-files}
```

During compile time, the module descriptors extracted from the module path are used to build a module graph. This graph can then be used to check whether each type is accessible from a particular module. [5].

To package the classes into a modular JAR, the `jar` command can be used as it was used before Java 9. The only difference between a traditional JAR and a modular JAR is the 'module-info.class' that is packaged into the JAR. Also, it is possible to define a start class using the `main-class` option.

```
$ jar --create --file modules/app.jar
      --main-class com.example.app.Starter
      -C classes-dir .
```

The created module can then be executed with the following command:

```
$ java --module-path modules --module com.example.app
```

Similar as described for compile time the information form the module path is used to enforce the accessibility rules at run time. In this example, `com.example.app` is the start module and since we have defined a `main-class` for the module, there is no need to explicitly define the main class when executing the application [3].

If a module is missing or any accessibility rule is violated the module system instantly reports an error during compilation or at application launch. If code in the module `com.example.app` imports `com.example.foo` and this package is not exported by any other module, the JVM would throw an error which looks like this:

```
Error occurred during initialization of boot layer
java.lang.module.ResolutionException: Module com.example.app
    does not read a module that exports com.example.foo
```

The detailed error messages and fast failing of the JVM can help to find errors faster and improve the stability of an application at run time.

4.5 Module Types

For compatibility, the class path still exists, despite the module path was introduced. Nevertheless, the JVM has to treat the class path's content as one module due to the modularization of the Java Platform. It cannot be treated as a regular module, though, as there is no module declaration. Thus, Java knows different module types it treats differently.
First of all, the modules we know for now are called *named modules* and are treated as described above. These can be further divided into system modules that are the modules shipped with the Java Development Kit (JDK) and application modules, which are modules created by developers.
If a JAR file is placed on the class path, it is added to the *unnamed module*, even if it is modularized. The unnamed module automatically reads all available exported packages and exports all its packages. However, these cannot be read by the named modules due to missing `requires` statements of the named modules.
This missing connection is fixed by the introduction of *automatic modules*, which are named modules, too. These are JAR files without a module descriptor that are placed on the module path. The module descriptor is then automatically generated, so automatic

modules read all available exported packages of named modules and all packages of the unnamed module. Moreover, automatic modules export all their packages and force all other named modules and the unnamed module to read their packages. That creates practical possibilities to modularize applications step by step. An automatic modules name is derived by the file name of its JAR. For example, the module name derived from 'foo-bar-4.2.jar' would be 'foo.bar' [2].

5 Advanced Concepts

Beyond the basic features described in section 4, the module system offers further concepts for advanced usage. They may not be needed, but are useful in special situations.

5.1 Transitive Dependencies

Transitive dependencies come in use when a library exposes types from a second module in its API. If that API is used, an application would also need to require the second module, since it accesses the exposed types. For such a case, transitive dependencies can be used to make every module requiring the library to also require the transitive dependencies automatically.

```
module com.example.library {
    exports com.example.library;
    requires transitive com.example.exposedlibrary;
}
```

An application module would then only need to require the library and would then automatically read `com.example.exposedlibrary`:

```
module com.example.app {
    requires com.example.library;
}
```

5.2 Optional Dependencies

In some cases, it may be necessary to declare a module dependency as optional. For example, a library may provide some advanced functionality that is not used by every application. For this functionality, the library may depend on another huge library. To avoid that all users must include that huge library on their module path, the dependency can be declared as `static`. This means that it is required at compile time, but not at run time. Only applications that want to use the libraries advanced functionality must add it to the module path [4].

```
module util.library {
    exports util.library.defaultapi;
    exports util.library.advancedapi;
    requires static some.optional.library;  }
```

The drawback of optional dependencies is that the advantage of dependency checking during application launch gets lost.

5.3 Reflection

Reflection is a very powerful tool in Java programming. It can be used to dynamically access classes, read their structure or execute methods at run time. Annotations of types, fields, methods or parameters can be read to get additional information about their functionality. By using `.setAccessible(true)`, it is even possible to make types, fields or methods accessible that are not visible to a class. However, this so-called *deep reflection* violates the concept of encapsulation and allows applications to read and modify internal implementation details of packages and classes [5].

With the introduction of the module system, accessing types, methods, and fields through reflection is restricted in the same way as it is at compile time. Private methods and fields can no longer be accessed through reflection by default [1]. But many popular frameworks, like Hibernate, Spring, etc., depend on the usage of deep reflection. For that reason, a module can `open` packages. This makes a package only accessible at run time, but not at compile time. Moreover, it enables other modules to use deep reflection inside an opened package. If only specific modules should have reflection access to a package, it can be opened qualified the same way as packages can be exported to specific modules [4].

```
module com.example.app {
    opens com.example.app;
    opens com.example.app.foo to com.example.framework;
}
```

5.4 Services

One useful feature of the JPMS is the support for the *ServiceLoader* API, introduced in Java 6, to help to uncouple interfaces easily from their implementations and their service users. The interface is called *service*, while the implementation is called the *service provider*. It can be every interface, but its package has to be *exported* by its module. Because both, service provider and service user, need to use the interface, their modules have to require the module that defines the service [6].

Besides that, the provider module has to declare that it provides an implementation for a specific service in its module declaration. The implementation itself does not need to be exported, which keeps implementation details private. To use a service, a module just has to specify the service in its module declaration [6]. All in all, a working example would be:

```
module com.example.service {
    exports com.example.service;  }
```

```
module com.example.provider {
    requires com.example.service;
    provides com.example.service.FancyService
        with com.example.provider.FancyServiceImpl;
}
module com.example.app {
    requires com.example.service;
    uses com.example.service.FancyService;
}
```

Moreover, an application has to create a `ServiceLoader` for the service wanted. This service loader offers methods to access available services. Annotations may be read, using reflection, to get additional information about a service implementation to choose the most suitable implementation. An actual service instance can be received by calling the `findFirst()` method that returns an instance of the first found implementation if there is any [6]. To create that instance, the service loader either invokes the parameter-less constructor or the *provider method* (`public static FancyService provide()`). If the construction of a service implementation is more complicated, it is a recommended design to design the service provider as a factory and then create the actual service implementation using the factory [5]. Loading a service may look like this:

```
ServiceLoader<FancyService> loader
    = ServiceLoader.load(FancyService.class);
Optional<FancyService> serviceOptional = loader.findFirst();
if( serviceInstance.isPresent() ) {
    FancyService serviceInstance = serviceOptional.get();
    // work with the instance
}
```

As seen in the listing, the code should deal with the case in which there are no services available. One option may be to assure that the code is always shipped with some default implementation.

5.5 JLink

The module system brings another benefit: Because the Java Platform is split into modules, it is possible to select only a subset of modules to run an application. To take advantage of this, the JDK comes with a new tool: *jlink*. It allows building custom Java Runtime images of selected modules. Moreover, it is possible to include custom modules into that image and exclude all unnecessary JDK modules. Thus, the image size can be drastically reduced. An image only consisting of `java.base` has only a size of 45 MB, in opposition to the JDK having a size of 221 MB, using Java 9 on Linux. One advantage of this is a reduced overhead when running applications on hardware with only a small amount of memory (e.g. Internet Of Things devices or containers for web applications). When using jlink, the module-path option is once again used to specify the directories containing the modules and it has to be specified which of the modules should be included [5]. All required dependencies are automatically included in the image. The use of jlink might look like this:

```
$ jlink  --module-path ${jdk-folder}/jmods:modules
         --add-modules java.xml,com.example.app
         --output custom-runtime-image
         --launcher run-app=com.example.app
```

As seen in the example, there is an option to create a launcher script for a specified start module. If not using a launcher, programs may be started by using the `java` binary of the created runtime environment. This is done in the same manner as with Java's default binary [5].

6 Conclusion

We discussed why modularization is an important concept of software engineering and how this concept is integrated into the Java platform. The Java module system enforces encapsulation of modules and provides mechanisms for fast detection of dependency and accessibility errors. Through its built-in support for the ServiceLoader API, it supports loose coupling of modules. All this leads to a better software structure, improves maintainability and supports the development of complex software systems.
Further concepts not mentioned are module layers, which allow using multiple versions of a module at once. Moreover, it is possible to modify module declarations at launch time to easily allow the use of test frameworks, like JUnit.

References

[1] Paul Deitel. Undertanding java 9 modules, 2017. URL https://www.oracle.com/corporate/features/understanding-java-9-modules.html.

[2] Guido Oelmann. *Modularisierung mit Java 9 - Grundlagen und Techniken für langlebige Softwarearchitekturen.* dpunkt.verlag, Heidelberg, 2018. ISBN 978-3-960-88411-8.

[3] Unknown Author OpenJDK. Project jigsaw: Module system quick-start guide, 2017. URL http://openjdk.java.net/projects/jigsaw/quick-start.

[4] Nicolai Parlog. Code-first java module system tutorial, 2017. URL https://blog.codefx.org/java/java-module-system-tutorial/.

[5] Nicolai Parlog. *The Java Module System.* Manning, Shelter Island, 1 edition, 2018. ISBN 9781617294280.

[6] Peter Verhas. Java 9 module services, 2018. URL https://dzone.com/articles/java-9-module-services.