Java Modules

Table of Contents

Java Platform Module System (JPMS)

- As of Java 9, the Java Platform API has been split up into separate modules
- This is due to the Java Platform Module System (JPMS)
- Introduced through Project Jigsaw

What is a module?

- A group of closely related packages and resources along with a new module descriptor file.
- A package (not to be confused with a java package) that encapsulates a set of other class in a jar, called a *modular jar*

What problems does a module solve?

Before JDK 9 there were considerable issues with the way classloading worked and how dependencies were resolved:

- Module Erasure
- Not Knowing Requirements
- Shadowing classes with the same name
- Version Conflicts
- Complex Class Loading
- Weak Encapsulation between Jars

Module Erasure

- Modularization efforts to contain classes in a group and tighten any accessibility were effectively undone
- This lead to what is called *module erasure* at runtime
- Module erasure leads to a big ball of mud of classes, lacking structure

Not Knowing Requirements

- Often we bring in dependencies without knowing the requirements that are needed
- Up until Java 8 there has been no enforcement of ensuring that all dependencies are maintainted

Shadowing classes with the same name

- There may be two different versions of the same library.

- A JAR may contain its own dependencies—it's called a fat JAR or an uber JAR—but some of them are also pulled in as standalone JARs because other artifacts depend on them.
- A library may have been renamed or split, and some of its types are unknowingly added to the class path twice.
- Because a class will be loaded from the first JAR on the class path that contains it, it makes all other classes of the same name unavailable—it's said to shadow them.

Version Conflicts

- Version conflicts arise when two required libraries depend on different, incompatible versions of a third library.
- If two libraries are present on the class path, the behavior will be unpredictable
- Because of shadowing, classes that exist in both versions will only be loaded from one of them
- If a class that exists in one version but not the other is accessed, that class will be loaded as well
- Code calling into the library may find a mix of both versions

Complex Class Loading

- Default Behavior is that many Java Applications are loaded by a single class loader
- When developers perform custom class loading, that's when complexity arises
- They use this complex class loading to implement features like extend the application by loading new classes, or to be able to use conflicting versions of the same dependency.

Weak Encapsulation

- Since once everything is resolved, class security is defined by package, there is no way to keep access to a certain package from other classes
- Encapsulation is important since it not only prevents access but allows refactoring
- Reflection can also be performed to break into APIs not intended to be used
- Many applications have "internals" that are not meant to be used but are often used
- Example: `sun.misc.Unsafe` is used to allocate memory direct, but was never meant for it to be used directly
- Example: JUnit 4, many applications broke into internals, thus, making it difficult to add features without breaking code

Security Checks

- Previous to JDK9 module erasure forced APIs to be accessed that shouldn't.
- This forced developers to call `SecurityManager.checkPackageAccess` to prevent any malicious code from access

No Java Runtime Subsets

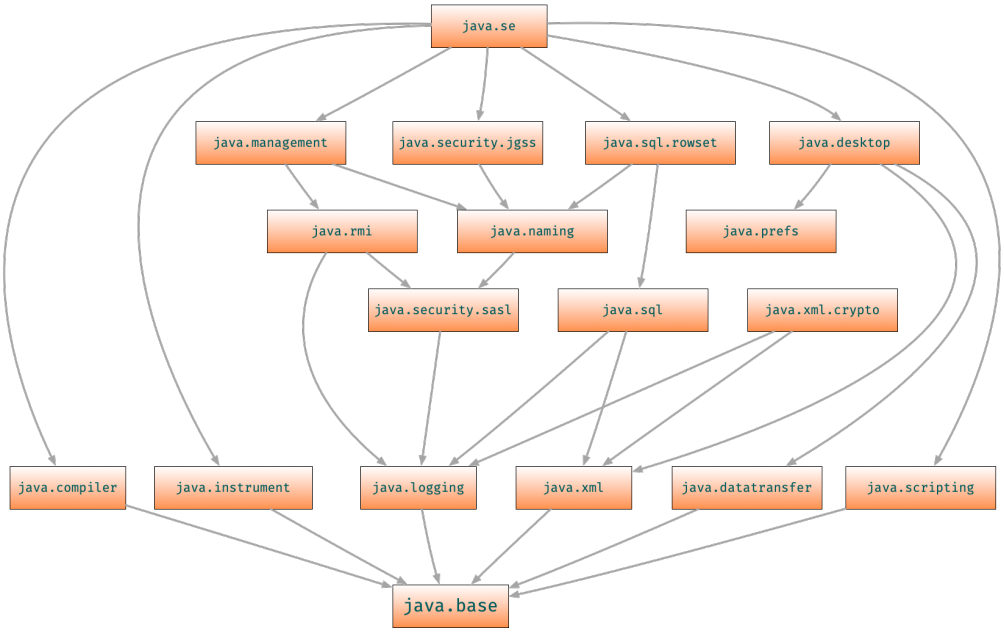
- Previous to JDK 9, there wasn't a way to install a subset of the JRE
- This is important on small devices and containers
- There were compact profiles, but didn't quick cover all use cases.

JDK Modules

- The JDK now has 100+ modules called *platform modules*
- 30+ modules start with the prefix `java.*`

Common Modules

JDK internally uses default common modules Note that `java.se` and `java.base` are separate in what they entail



Sample of Common Modules

- `java.base` — The module without which no JVM program functions. Contains packages like java.lang and java.util.
- `java.desktop` — Not only for those brave desktop UI developers out there. Contains the Abstract Window Toolkit (AWT; packages java.awt.**), Swing (packages javax.swing.**), and more APIs, among them JavaBeans (package java.beans.*).
- `java.logging` — Contains the package java.util.logging.
- `java.rmi` — Remote Method Invocation (RMI).
- `java.xml` — Contains most of the XML API word salad: Java API for XML Processing (JAXP), Streaming API for XML (StAX), Simple API for XML (SAX), and the document object model

(DOM).

- `java.xml.bind` — Java Architecture for XML Binding (JAXB).
- `java.sql` — Java Database Connectivity (JDBC).
- `java.sql.rowset` — JDBC RowSet API.
- `java.se` — References the modules making up the core Java SE API. (This is a so-called aggregator module; see section 11.1.5.)
- `java.se.ee` — References the modules making up the full Java SE API (another aggregator).

Module Resources

- 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

Module Descriptors

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

Module Naming Conventions

- It's very common to do either:
  - Project style - `my.module`
  - Reverse-DNS style - `com.xyzcorp.mymodule`
- You should not use an underscore

Creating a Module

- We need to declare all the packages that we want public
- By default all packages are private unless declared otherwise
- This includes via reflection, reflection is not open to access with modules

Module types

There are four main module types:

- **System Modules** – Java SE and JDK modules.
- **Application Modules** – These modules are what we usually want to build, and are named and defined in the compiled *module-info.class* file included in the assembled JAR
- **Automatic Modules** – 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.

Finding all system modules

You can locate all the modules using the following command:

```
$ java --list-modules
```

Module Declarations

- File known as the module descriptor, named *module-info.java*
- File should be created at the root of your project

```
module com.xyzcorp {}
```

`requires` Directive

- When we wish to reference another *module*
- This is a *runtime* and *compile-time* dependency
- All public types exported from this dependency are available and accessible within our application

```
module com.xyzcorp {
    requires module.name;
}
```

`exports` Directive

- A Java module must explicitly export all packages in the module that are to be accessible for other modules using the module.
- The exported *packages* are declared in the module descriptor, `module-info.java`

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

`exports` subpackages

- Only the listed package itself is exported
- No "subpackages" of the exported package are exported
- To export the subpackages as well, you will need to include that subpackage

```
module com.abccorp {
   exports package.name
   exports package.name.util;
}
```

`exports to..`

- An `exports···to` directive enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package
- This is known as a qualified export
- The following is some qualified exports from `java.base`

```
exports jdk.internal.org.objectweb.asm.tree to
    jdk.jfr,
    jdk.jlink;
exports jdk.internal.org.objectweb.asm.util to
    jdk.jfr;
exports jdk.internal.org.objectweb.asm.commons to
    jdk.jfr;
exports jdk.internal.org.xml.sax to
    jdk.jfr;
exports jdk.internal.org.xml.sax.helpers to
    jdk.jfr;
```

Source: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/module-info.java

Lab 1: Create a Module

We will be creating a set of projects *without an IDE*

1. Create a directory called *modular-lab* in any directory
2. In the *modular-lab* directory, create a directory called *abccorp.calculator*
3. In the *abccorp.calculator* directory, create directories called *src/com/abccorp*, using `mkdir -p src/com/abccorp`
4. In *abccorp.calculator/src/com/abccorp* directory, create a file `Calculator.java`, and provide it with the following content:

```
package com.abccorp;

public class Calculator {
   public int add(int a, int b) {
      return a + b;
   }
}
```

5. In *abccorp.calculator/src* directory, create a file `module-info.java`, please note that that is in the *src* directory. Provide it with the following content:

```
module abccorp.calculator {
  exports com.abccorp;
}
```

6. Go to the *modular-lab/abccorp.calculator* directory, and type `javac -d out $(find . -name "*.java")`
7. Look inside *modular_lab/abccorp.calculator/out* directory, and ensure that it has two classes, `Calculator.class` and the `module-info.class`

8. In the *modular-lab* directory, create a directory called *xyzcorp.main-app*
9. In the *xyzcorp.main-app* directory, create directories called *src/com/xyzcorp*, using `mkdir -p src/com/xyzcorp`
10. In *xyzcorp.main-app/src/com/xyzcorp* directory, create a file `Runner.java`, and provide it with the following content:

```
package com.xyzcorp;

import com.abccorp.Calculator;

public class Runner {
   public static void main(String[] args) {
      Calculator calculator = new Calculator();
      System.out.println(calculator.add(30, 10));
   }
}
```

11. In *xyzcorp.main-app/src* directory, create a file `module-info.java`, please note that that is in the *src* directory. Provide it with the following content:

```
module xyzcorp.mainapp {
  requires abccorp.calculator;
}
```

12. In a terminal, go to the *modular-lab/xyzcorp.main-app* directory, and type in a `javac --module-path ../abccorp.calculator/out -d out $(find . -name "*.java")`
13. Look inside *modular_lab/xyzcorp.main-app/out* directory, and ensure that it has two classes, `Runner.class` and the `module-info.class`
14. Finally, change back to the *modular-lab* directory, and run the application with the following in your shell

```
$ java --module-path abccorp.calculator/out:xyzcorp.main-app/out --module
com.xyzcorp/com.xyzcorp.Runner
```

15. Ensure you are in the *modular-lab* directory, and view all the modules available, including your own

```
$ java --module-path abccorp.calculator/out:xyzcorp.main-app/out --list-modules
```

16. Ask yourself, What would happen if you removed `exports com.abccorp` from the `abccorp.calculator` module? Try it by editing the `module-info.java` and retrace your steps. Where does it fail?

Circular Dependencies are not Allowed

- It is not allowed to have circular dependencies between modules.
- If `module-a.app` requires `module-b.app`, then `module-b.app`, cannot also require `module-b.app`

Split Packages are not Allowed

- The same Java package can only be exported by a single Java module at runtime.
- In other words, you cannot have two (or more) modules that export the same package in use at the same time.
- Having two modules export the same package is also sometimes referred to as a *split* package.
- By split package is meant that the total content (classes) of the package is split between multiple modules. This too is not allowed.
- In other words, two distinct modules, `module-a.app`, and `module-b.app` contains *packages*, and we are talking about packages, then using them in modules is not allowed.

`requires static`

- Required compile-time dependency
- Optional run-time dependency
- For example, not every consumer of our library will want this functionality, and they don't want to include an extra library

`requires transitive`

- `transitive` is that the target module is not only required by the current module
- It also be readable to other modules which are reading this module.
- `requires transitive` seems to be same as `exports` but they are different
  - `exports` is used to make packages visible
  - `requires transitive` is used to make imported modules visible outside.

The Unnamed module

- The unnamed module contains all non-modular classes, which are:
  - The classes being compiled at compile time, if they do not include a module descriptor
  - All classes loaded from the class path at compile time and runtime

The Automatic module

- For every JAR on the module path that has no module descriptor, the module system creates an

automatic module

- A choice for third party library providers to say "we hope to create a module but for now, here are the coordinates"

Viewing the Metadata of the Module

- Jar files carry metadata about the module
- Can also be applied to old jar files where metadata is lacking

```
$ jar -f <jar-file> -d
```

Lab 2: View Metadata of Jar Files

1. Go to https://dev.mysql.com/downloads/connector/j/
2. In the "Select Operating System…", select "Platform Independent"
3. Download the zip archive
4. On the next page, select "No thanks, just start my download."
5. Locate and expand the zip file
6. In the expanded folder, you will find the jar file, run the metadata command to see if it unnamed or automatic?

   ```
   $ jar -f <location-of-mysql-jar>.jar -d
   ```

7. Change to *modular-lab/abccorp.calculator* directory
8. Create a directory called `dist`
9. Create a jar file

   ```
   $ jar -cf dist/abccorp-calculator.jar -C out .
   ```

10. Determine what the metadata says

    ```
    $ jar -f dist/abccorp-calculator.jar -d
    ```

11. Run your application by referring to the jar file instead

```
$ java --module-path abccorp.calculator/dist/abccorp-calculator.jar:xyzcorp.main-
app/out --module com.xyzcorp/com.xyzcorp.Runner
```

Services

- We can declare Services and Service Providers as modules.
- A service module declares that it uses one or more interfaces whose implementations will be provided at run time by some provider modules.
- A provider module declares what implementations of service interfaces it provides

Creating a Service Module

- The Service Module represents the `interface`
- It is the *Service Provider Interface*

```
module com.service.api {
    exports com.service.api;
    uses com.service.api.ServiceInterface;
}
```

Using the Service Loader at Runtime

- `java.util.ServiceLoader`
  - locates and loads service providers, a.k.a implementation, deployed in the run time environment at a time of an application's choosing
  - is a `java.util.Iterable` that can be converted to a `Stream` or `Collection`
  - Can be provided by the `Service` interface itself as a helper method

```
package com.service.api;

interface ServiceInterface {
  static List<ServiceInterface> getInstances() {
    ServiceLoader<ServiceInterface> services = ServiceLoader.load
(ServiceInterface.class);
    return services.stream().map(ServiceLoader.Provider::get).toList();
  }
  public void execute() {}
}
```

Creating the Concrete Service, a.k.a Service Provider

- Once the interface is established, we can then create the concrete implementation

- This will require that we import the `interface`, which will inevitably come from the service api

```
package com.service.console;

import com.service.api.*;

public class ConcreteService implements ServiceInterface {
   public void execute() {
       getInstances().forEach(ServiceInterface::execute());
   }
}
```

Creating a Service Implementation

```
module com.service.provider {
    requires com.service.api;
    provides com.service.api.ServiceInterface with com.service.provider
.ConcreteService;
}
```

Client Structure

- In order to use the s we only need to provide the path to the service and *not the provider*
- This is very familiar to the Decorator Pattern where the core implementation is masked and not known until run-time

```
module com.xyzcorp {
  requires com.service.api;
}
```

Demo: Services

Reflection

- Reflection is used to gain access to `private` information from compiled classes using `AccessibleObject.setAccessible(..)`
- Reflection will be done by:
  - Obtaining the class via `getClass()`
  - Calling any of the classes properties like Constructors, Fields, and Methods, which extend `AccessibleObject`
  - From the constructor, field, or method's reference, call `setAccessible(true)` to allow access

Module Descriptor for Reflection Service

- The module descriptor is straightforward and has no special

```
module com.qrscorp {
  exports com.qrscorp.util;
}
```

Module Description for Client

- The client is the one that must grant accessibility to another module
- Module declaration must use either `open`, `opens`, or `opens to`

```
module com.xyzcorp {
   requires com.qrscorp;
   opens com.xyzcorp to com.qrscorp;
}
```

Alternative Module Descriptions

You can either declare the `module` as `open`

```
open module com.xyzcorp{
    requires com.qrscorp;
}
```

You can also open your package declaration

```
module com.xyzcorp {
    requires com.qrscorp;
    opens com.xyzcorp;
}
```

What's `jlink`?

- Creation of Java runtime images that contain only the modules you need for your application
- Further optimizes image size and improve VM performance, particularly startup time
- Allows the end user to create a customized JRE to run your application

`jlink` requirements

- Where to find the available modules (specified with `--module-path`)
- Which modules to use (specified with `--add-modules`)
- Folder in which to create the image (specified with `--output`)

What's `jdeps`?

- Used to determine which platform modules you need to sustain a large application
- Provides what modules that your application depends on
- It can output this to a text file, which can then be consumed by jlink

```
$ jdeps -summary -recursive --class-path 'jars/*' jars/app.jar
```

Creating the smallest image possible

- As a hack you can use the following command provide all modules that are required
- This can provide the smallest image possible by feeding into `jlink --add modules`

```
$ jdeps -summary -recursive --class-path 'jars/<strong>' jars/app.jar
    | grep '\-> java.\|\-> jdk.'
    | sed 's/^.<\/strong>-> //'
    | sort -u
```

Creating smaller docker images with `jdeps` and `jlink`

https://levelup.gitconnected.com/java-developing-smaller-docker-images-with-jdeps-and-jlink-d4278718c550

Relationship with build tools

- The thing to make sure of, is that compilation and runtime would need the information of what modules to use
- This can be obtained using arguments on how applications are run

Testing with Modules

- Unit (whitebox) tests
  - Need to access the internals of a module can be written in the traditional way by not adding a `module-info.java`.
  - In test execution, the modules are then treated as standard Java libraries with the encapsulation deactivated
- Blackbox (e.g. integration) tests
  - Follow's the encapsulation rules during test execution
  - Can be written by turning the corresponding test sources set itself into a module by adding a `module-info.java`.

Order of the module declaration

The following is the order of calls in your module declaration:

- `requires`, including `static` and `transitive`
- `exports`
- `exports to`
- `opens`
- `opens to`
- `uses`
- `provides`

Java 9 Modules are backwards compatible

- If you have:
  - Non-modularized Java 9+ code
  - Java 8 code
- You can still benefit using the JPMS using the unnamed module

Upgrading to JDK Modules

Upgrading can be made "bottom up" or "top down"

- **Bottom up** - Migrate the small libraries first, you main applications last
- **Top down** - Migrate you main application first, and then libraries later

Upgrading to JDK Module Steps

1. Use the JDK Unnamed Module
    a. Upgrade to JDK module compliant version.
    b. Do not "modularize" any of the applications
    c. All classes will be a part of unnamed module
2. Move libraries to the module path
    a. Make internal, and third party libraries automatic modules
    b. Main applications should still be on the classpath in the unnamed module
    c. Any class in the unnamed module should read all named modules
3. Convert libraries to Java Modules
    a. Place the Java modules on the module path.
    b. Start with libraries that have no other module / JAR dependencies,
4. Migrate your main applications
    a. This is possible when
        i. All dependencies have been upgraded
        ii. All dependencies are either automatic or named

A look at Layrry

https://www.morling.dev/blog/introducing-layrry-runner-and-api-for-modularized-java-applications/