Modularization Exercises

Contents

This Workshop is split into a few parts, which will take the participants closer to the modularization theme and the new features therein. In conjunction, there are exercises which the participants should solve on the computer – ideally in teams.

Prerequisites

1) Current JDK 11 installed
2) Current Eclipse installed (Alternatively: NetBeans or IntelliJ IDEA)

Participants

- Developers with Java experience, as well as
- Software architects, who would like to learn/evaluate Java 9 – 13

Course management and contact

Michael Inden
CTO & Teamlead SW-Development & Head ASMIQ Academy
ASMIQ AG, Geerenweg 2, 8048 Zürich
E-Mail: michael.inden@asmiq.ch
Course offer: https://asmiq.ch/
Blog: https://jaxenter.de/author/minden

Modularization – Project Jigsaw

Hereafter, a simple program will be iteratively transformed into a modularized software and provided with some improvements.

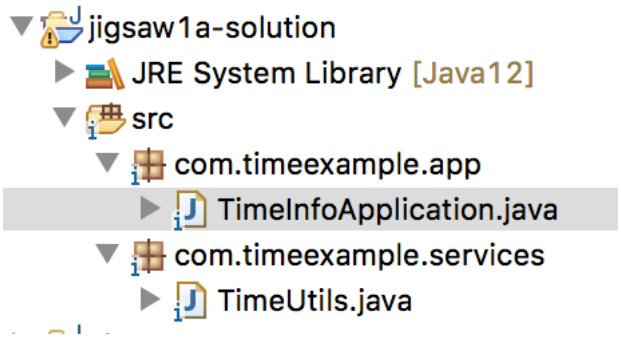
Exercise 1 – Split the application into two modules

Given is a simple application. The functionality is located in **`main()`**, as well as the actual computation in the utility method. This is still made according to a monolithic architecture inside a single class. This application should now be adapted by iteratively applying the modularization approach introduced by JDK 9 with the help of the following exercises.

```
package com.timeexample;

import java.time.LocalDateTime;

public class CurrentTimeExample
{
	public static void main(final String[] args)
	{
		System.out.println("Now: " + getCurrentTime());
	}

	public static LocalDateTime getCurrentTime()
	{
		return LocalDateTime.now();
	}
}
```

Exercise 1a: Structuring for modularization

Extract the following classes from the main class: **`TimeInfoApplication`** and **`TimeUtils`**. Structure the packages to be **`app`** and **`services`**.



Exercise 1b: modularization of the application
Split the application into two modules named **`TimeClient`** and **`TimeServer`**. In Eclipse, use two separate projects; in IntelliJ use a hierarchical project with two modules.

Transfer both packages in the appropriate modules, which are named **`TimeClient`** and **`TimeServer`**. Use a **`module-info.java`** data file for the modularization, where dependencies are correctly specified. Be mindful of the impact of the module dependencies and adapt the configurations of your IDE accordingly. Once you are done, start the modularized application.

Tip 1: during compile, you may encounter errors:

The functionality to create module descriptors is somewhat hidden in Eclipse. You may find it in the context menu of your project under "`configure -> create` **`module-info.java`**"

Tip 2: during compile, you may encounter errors:

Check in the module descriptors for dependencies and permissions, for example "`requires` XYZ" or "`exports com.abc.def`". IDEs provide some quick fixes for this

Tip 3: Use the module-path from the main folder "`jigsaw1b`" at start time and reference both projects:

```
java -p ./TimeServer/bin:./TimeClient/bin \
     -m timeclient/com.timeexample.app.TimeInfoApplication
```

Exercise 1c: generating modular JARs
Generate a JAR (two total) from the compiled sources of each module/subfolder. To this end, create a folder named libs in the main folder, which serves as a goal for modules. Copy the generated JAR from **`TimeServer`** into **`TimeClient`**. The folder should look as follows:

```
├── TimeClient
│   ├── bin
│   │   ├── com
│   │   │   └── timeexample
│   │   │       └── app
│   │   │           └── TimeInfoApplication.class
│   │   └── module-info.class
│   …
└── TimeServer
    ├── bin
    │   ├── com
    │   │   └── timeexample
    │   │       └── services
    │   │           └── TimeUtils.class
    │   └── module-info.class
```

As a result, following addition is expected:

```
├── libs
│   ├── timeclient.jar
│   └── timeserver.jar
```

The software should be able to start with following command:

```
> java –p libs –m timeclient/com.timeexample.app.TimeInfoApplication
```

CAUTION/CAUTION/CAUTION:
Because of an error in the JAR-tool is this process currently not available and will show this error:

```
Error occurred during initialization of boot layer
java.lang.LayerInstantiationException: Package com in both
module timeserver and module timeclient
```

It is however possible to use a Gradle/Maven build and to collect the generated JARs in a folder

Tip: This command helps to generate the JARs when run in their respective project main folders:

```
> jar ––create ––file libs/<myjarname>.jar –C bin .
```

Tip: To copy jars, following command will be useful:

```
> mv libs/timeserver.jar ../TimeClient1/libs
```

GRADLE-BUILD WORK-AROUND

Use a gradle-build as a work-around, and a **`build.gradle`** file composed of the gradle clean assemble command, as well as following lines:

```
apply plugin: 'java'

sourceCompatibility=11
targetCompatibility=11
```

This should be located in the project's main folder. For the **`TimeClient`** project, you will also need to provide the dependencies as specified above:

```
tasks.withType(JavaCompile) {
    options.compilerArgs +=
 ["--module-path",
  "${buildDir}/../../TimeServer1/build/libs"]
}
```

Exercise 2 – Preparing dependencies

Based on the application's split into two modules from the first exercise, generate and visualize a dependency graph. This process is simplified if the gradle-generated JARs are copied in a separate libs folder

Exercise 2a: Listing dependencies
To calculate dependencies, use the **`jdeps`** tool. Its output should look something like this:

```
timeclient
 [file:///Users/michael.inden/eclipse-
workspace/TimeClient2/libs/TimeClient2.jar]
   requires mandated java.base (@12)
   requires timeserver
timeclient -> java.base
timeclient -> timeserver
   com.timeexample.app             -> com.timeexample.services
timeserver
   com.timeexample.app             -> java.io
java.base
   com.timeexample.app             -> java.lang
java.base
   com.timeexample.app             -> java.lang.invoke
java.base
   com.timeexample.app             -> java.time
java.base
timeserver
 [file:///Users/michael.inden/eclipse-
workspace/TimeClient2/libs/TimeServer2.jar]
   requires mandated java.base (@12)
   requires java.logging (@12)
timeserver -> java.base
   com.timeexample.services         -> java.lang            java.base
   com.timeexample.services         -> java.time
java.base
java
```

This is, however, quite unreadable. How can we obtain a summary similar to this one?

```
timeclient -> java.base
timeclient -> timeserver
timeserver -> java.base
```

Exercise 2b: Preparing a graphic representation of dependencies
To generate a graphical representation of these rather confusing informations, use the Graphviz tool (http://graphviz.org) in combination with **`jdeps`** and the **`–dotoutput`** flag.

```
jdeps –dotoutput graphs libs/*.jar
```

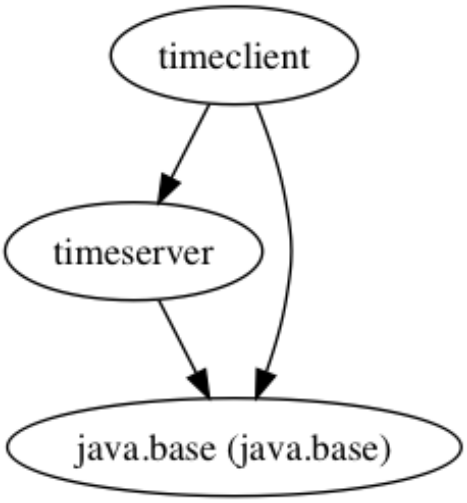
This should generate following data:

```
├── graphs
│   ├── summary.dot
│   ├── timeclient.dot
│   └── timeserver.dot
```

To convert DOT-Graphs in PNG format, following command is useful – in this example for the data named summary.dot:

```
dot –Tpng graphs/summary.dot > summary.png
```

This should generate the following dependency graph:



Exercise 3 – Using a JDK module

Extend the **`TimeUtils`** class, so as to log method calls. Use the `java.util.logging` Logger to this end:

```
package com.timeexample.services;

import java.time.LocalDateTime;
import java.util.logging.Logger;

public class TimeUtils
{
	public static LocalDateTime getCurrentTime()
	{
		Logger.getGlobal().info("getCurrentTime() is called()");
		return LocalDateTime.now();
	}
}
```

What occurs during compilation? How can you fix the reported error easily?

Exercise 4 – Create an Executable Runtime

The before modularized application should be generated as a single standalone executable in a **`runtime_example`** folder.

Exercise 4a: Generating JARs
You may start by creating the JARs, normally with:

```
jar --create --file libs/xyz.jar -C bin/XYZ .
```

Because of an error in the JAR tool, it is currently not possible to use this method. It is however possible to generate JARs in a libs folder with a Gradle/Maven build.

Exercise 4b: Set correct JAVA_HOME during runtime
Check your environment variable JAVA_HOME and set it – as required – to the wanted JDK, in our case JDK 11:

```
export JAVA_HOME=`/usr/libexec/java_home -v "Java SE 11.0.1"`
export PATH=$JAVA_HOME/bin:$PATH
```

Exercise 4c: Generating executables
With the help of **`jlink`**, generate an executable in the **`runtime_example`** folder. The result should look like this:

```
runtime_example/
├── bin
│   ├── MYEXE
│   ├── java
│   └── keytool
├── conf
│   ├── logging.properties
```

Start the application so as to test the operating principle:

```
> ./runtime_example/bin/MYEXECUTABLE
Apr 19, 2019 2:26:58 PM com.timeexample.services.TimeUtils
getCurrentTime
INFO: getCurrentTime() is called()
Now: 2019-04-19T14:26:58.
```

Tip 1: To generate the executable runtime, use the jlink command, as well as the option **`--add-modules TimeClient`**. Also provide the JDK to the module path.

Tip 2: Don't forget the `--launcher` and `-output options`.

Tip 3: Try as follows:

```
jlink -p $JAVA_HOME/jmods:Build/libs:../TimeServer4/build/libs \
      --add-modules timeclient \
      --launcher=MYEXE=timeclient/com.timeexample.app.TimeInfoApplication \
      --output runtime_example
```

Exercise 4d: How big/small is the folder?
Use Windows Explorer, Mac Finder or the command:

```
du -sh runtime_example/
```

Exercise 5 – Solving dependencies with services

So far, programs are structured with modules, but with direct dependencies and therefore strongly coupled. To reduce coupling, you may use Services. To this end, Java provides the **`ServiceLoader`** class, as well as the keywords **`uses`** and **`provides with`** in module descriptors.

In the previous application, the **`TimeClient`** module should be made independent from the **`TimeServer`** module. To this end, an interface may be introduced and supplied in a separate **`TimeService`** module:

Exercise 5a: Generating a new `TimeService` module
Following interface will be used to separate client and server:

```
package com.timeexample.spi;

import java.time.LocalDateTime;

public interface TimeService
{
      public LocalDateTime getCurrentTime();
}
```

Create the related module and start a first build.

Exercise 5b: Conversion of the existing modules

First of all, the **`TimeUtils`** class should implement the interface from the previous exercise and access the service with the help of the **`ServceLoader`** class from the JDK. Use following implementation:

```
package com.timeexample.app;

import java.util.Iterator;
import java.util.Optional;
import java.util.ServiceLoader;

import com.timeexample.spi.TimeService;

public class TimeInfoApplication
{
	public static void main(final String[] args)
	{
		final Optional<TimeService> optService =
                                        lookup(TimeService.class);
		optService.ifPresentOrElse(service ->
		{
			System.out.println("Service: " + service.getClass());
			System.out.println("Now: " + service.getCurrentTime());
		},
		() -> System.err.println("No service provider found!"));
	}

	private static <T> Optional<T> lookup(final Class<T> clazz)
	{
		final Iterator<T> iterator =
                                ServiceLoader.load(clazz).iterator();
		if (iterator.hasNext())
		{
			return Optional.of(iterator.next());
		}
		return Optional.empty();
	}
}
```

`TimeUtils` class should also be adapted as follows:

```
package com.timeexample.services;

import java.time.LocalDateTime;
import java.util.logging.Logger;

import com.timeexample.spi.TimeService;

public class TimeUtils implements TimeService
{
	public LocalDateTime getCurrentTime()
	{
		Logger.getGlobal().info("getCurrentTime() is called()");
		return LocalDateTime.now();
	}
}
```

Ignore compilation errors for now in both modules

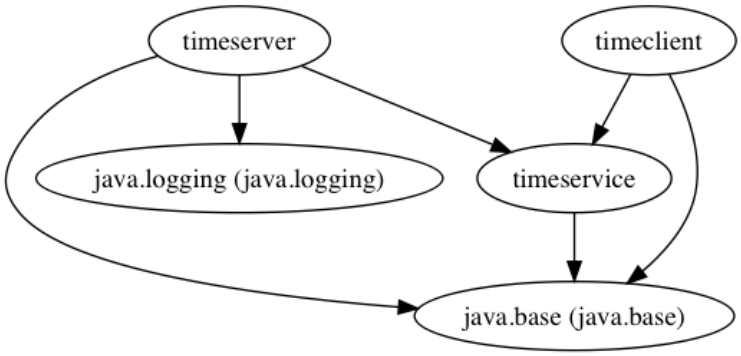
Exercise 5c: adapting dependencies in the module descriptors
Delete all dependencies between the modules in their descriptors and insert them again with all mandatory **`requires`**-instructions, as well as the dependencies between Eclipse projects. Subsequently fix the dependencies in the service context. Start with the **`TimeUtils`** module and use following components:

```
provides com.timeexample.spi.TimeService
with com.timeexample.services.TimeUtils;
```

As well as:

```
uses com.timeexample.spi.TimeService;
```

Exercise 5d: Compile all modules and check they are loosely coupled
Compile each module and adapt build data where necessary to reflect the new dependencies. Collect JARs in the **`libs`** folder again in the **`TimeClient`** main folder. Following dependency graph should be generated:



Tip: Use variations of the following command to fill the **`libs`** folder:

```
cp build/libs/*.jar libs
```

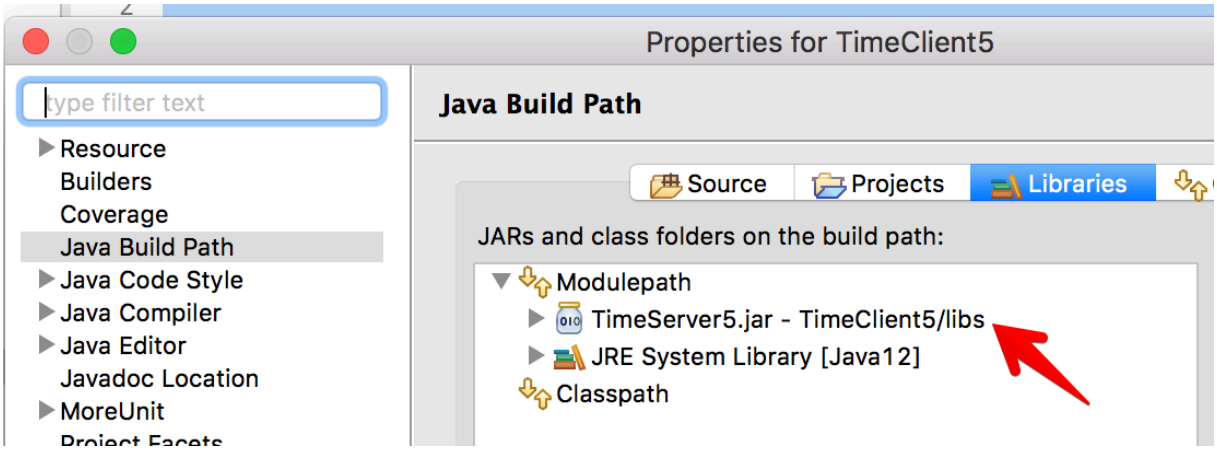
Exercise 5e: Execution with services
Even though the dependency between **`TimeClient`** and **`TimeServer`** modules is obviously not present any longer, the application should still work as intended. Start it

```
Service: class com.timeexample.services.TimeUtils
Apr. 19, 2019 6:08:08 NACHM. com.timeexample.services.TimeUtils
getCurrentTime
INFO: getCurrentTime() is called()
Now: 2019-04-19T18:08:08.913864
```

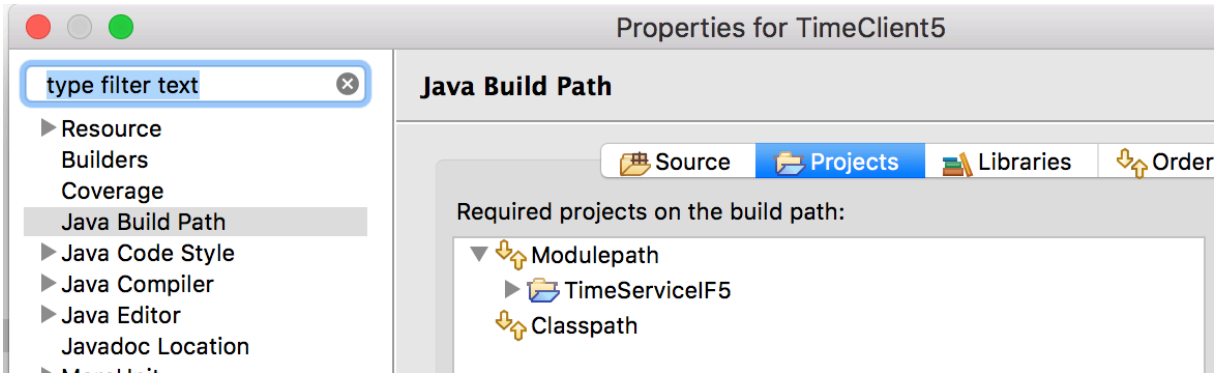
In Eclipse, you may wonder about an error logged at start up:

```
<terminated> TimeInfoApplication (3) [Java
No service provider found!
```

Why do we have this error? After some thought, we can suppose there is a missing dependency in **`TimeClient`** project. At compile time, the module does not require a dependency, but must be able to access the implementation at runtime. Configure your project as follows and start the application again:



Check if your project is only dependent on the interface:



Exercise 6 – Integrating external libraries

Let's assume we want to add a couple of functionalities from Google Guava to our application (for example version *guava-27.0.1-jre.jar*). Suppose we have the library in an **`externallibs`** folder beforehand, to make things easier for Eclipse. We'll take a look at Gradle and Dependencies later when building the project.

Start with the compatibility module, to which you can give a CLASSPATH.

```
javac –d build –cp ../externallibs/guava–27.0.jar \
      –p ../TimeServiceIF5/build/libs $(find src –name '*.java')
```

Although possible, it is usually more meaningful to link the JAR as an automatic module. This allows to merely require the JAR-file in the Module path:

```
javac –d build –p ../externallibs/guava–
27.0.jar:../TimeServiceIF5/build/libs $(find src –name '*.java')
```

To this end, the module descriptor should be adapted as follows:

```
    requires guava;
```

⇨ Name of automatic module 'guava' is unstable, it is derived from the module's file

Guava 27 may be bound by name through a special input in the Manifest as an Automatic module:

```
    Automatic-Module-Name: com.google.common
```

```
    requires com.google.common;
```

The Gradle-Build should be adapted as follows:

```
// bisschen getrickst aber ..
tasks.withType(JavaCompile) {
    options.compilerArgs +=
["--module-path",
"${buildDir}/../../TimeServiceIF5/build/libs:${buildDir}/../externallibs"]
}

repositories
{
      jcenter()
}

dependencies
{
      compile group: 'com.google.guava', name: 'guava', version: '27.1-jre'
}
```

Try to start the application with:

```
    java -p libs:externallibs/guava-27.0.1-jre.jar \
         -m timeclient/com.timeexample.app.TimeInfoApplication
```

This should output the following text:

```
Service: class com.timeexample.services.TimeUtils
Apr. 19, 2019 6:50:36 NACHM. com.timeexample.services.TimeUtils
getCurrentTime
INFO: getCurrentTime() is called()
Now: 2019-04-19T18:50:36.903890
Guava, From, ModulePath
```

YOU HAVE SUCCESFULLY FINSHED MODULARIZATION ☺
