JUG ISTANBUL

JAVA 9 & JAVA 10 WORKSHOP

RESOURCES

https://www.sitepoint.com/ultimate-guide-to-java-9/

https://static.rainfocus.com/oracle/oraclecode17/sess/1485992822413001Yd6N/PF/Cool%20in%20Java%208,%20and%20new%20in%20Java%209.pdf

https://github.com/ozlerhakan/java9-module-examples

https://trishagee.github.io/presentation/real_world_java_9/

https://github.com/CodeFX-org/demo-java-9

https://github.com/java9-modularity/examples

https://guides.gradle.org/building-java-9-modules/

https://www.youtube.com/watch?v=Txsl-K83ygI

https://www.youtube.com/watch?v=Yacu1yUktjY

https://blog.codefx.org/tools/jdeps-tutorial-analyze-java-project-dependencies/

http://tutorials.jenkov.com/java/modules.html

QUICK INFO

JAR FILE : are compressed files that allows packaging compile java files and transfering.

- There is no rule on naming.
- Name could be changed after created.
- All class files in Jar could be reached.
- Any private definition could be accessed by calling Java Reflection setAccessible() method.

MODULE : is a new packaging type coming with Java 9.

- Only permitted packages could be accessed by other modules.
- Depended modules must be specified.
- Usage of Java Reflection needs special permissions.
- Has module-info.java file to define dependencies and export packages to access.

MODULE TYPES

UNNAMED MODULES : A jar file is not located in module path.
AUTO MODULES: A jar file is located in module path but it doesn't include a module-info.java file. Java uses jar file name or specified name in manifest.mf file as module name.
NAMED MODULES: A jar file or JMOD file is located in module path and has a module-info.java file.

MODULE DESCRIPTOR FILE

Module descriptor file is named as module-info.java. It must be located in same folder with root package. Relationship of modules are defined in this file.

Relationship levels:

1. Exported Packages
2. Reflection Permissions
3. Depended Modules
4. Service Providers

CHEAT SHEET : https://zeroturnaround.com/rebellabs/java-9-modules-cheat-sheet/

```
module [MODULE_NAME] {

       exports [PACKAGE];
       exports [PACKAGE] to [SPECIFIC_MODULE_NAME];

       requires [DEPENDED_MODULE_NAME];
       requires transitive  [DEPENDED_MODULE_NAME_TO_OPEN_UP];

       uses [SERVICE_CLASS_NAME];
       provides [SERVICE_CLASS_NAME] with [CLASS_NAME_TO_IMPL] ;

       opens [PACKAGE_NAME_FOR_REFLECTION_ACCESS];
       opens [PACKAGE_NAME_FOR_REFLECTION_ACCESS] to [SPECIFIC_MODULE_NAME];
}
```

JAVA SYSTEM MODULES

Java Modules are located in JAVA_HOME/jmods folder. When you look in the folder, you will see java.base module. java.base module is default module and it is being loaded automatically if not declared in module-info.java.

You will see that there is -incubator- literal on some module names. This kind of modules are still being developed. For example, HTTP Client module.

WORKSHOPS

PREPARATION :

1. https://github.com/JUGIstanbul/java_9_10_workshop
2. You must install Java 9 or later version
3. JUG Istanbul directory structure should be created for project logging, library, and executable applications
    a. If your operating system is windows you should be created c:\jug_istanbul directory. You should be created /Jug_istanbul directory for linux or osx.
    b. The /jug_istanbul/bin, /jug_istanbul/modules, /jug_istanbul/conf and /jug_istanbul/logs directories should be created.

WORKSHOP-1 : INTRODUCTION TO MODULES

TIME : 30 min.
GOAL : Let's create our first module
SCENARIO :

As JUGIstanbul team, we are developing an open-source e-commerce platform. There is an item in project requirements : Use slugified product name for url of both product page and images.

Product Name : Lenovo Y520 i7-7700HQ 16GB 1TB+256SSD GTX1050 Win10 80WK004JTX
Slugified Text : lenovo-y520-i7-7700hq-16gb-1tb256ssd-gtx1050-win10-80wk004jtx

Package written codes as Java 9 Module to be used in other projects.
Module Detail:

Module Name : jugistanbul.slugifier
Module Version : 1.0.0
Package to be exported : jug.istanbul.slugify

GÖREVLER :

1. Let's list the system modules. Execute the command below

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

2. Let's check where these system modules are located in java. Go to JAVA_HOME/jmods folder.

3. Let's inspect module-info.java file defined in our example. Let’s discover which module has been used for which import.
4. Let's compile the application with the following command on the terminal.

```
javac -d out -s src $(find src/main/java -name "*.java")
```

5. In the slugifier module the Slugifier class that contains main method must be run and the sent statement should be seen as slugified.

```
java --module-path out --module jugistanbul.slugifier/jug.istanbul.slugifier.Slugifier
```

WORKSHOP-2 : MODULE BY MODULE COMPILING AND PACKAGING

TIME : 30 min.
GOAL : Module based compiling and packaging
SCENARIO :

As a developer new to Java 9, I’m discovering new things everyday. One of them is that locating application source files in folders named as Module Name.

After learning this feature, I decided to apply modules that I already have. Let’s make changes below for this purpose:

1. Let’s create a folder named as src in workshop-2 folder
2. Let’s create folders below in src folder
    a. jugistanbul.workshop2.module1
    b. jugistanbul.workshop2.module2
3. Let’s copy source files in jug-istanbul/module-1/src/main/java path into src/jugistanbul.workshop2.module1
4. Let’s copy source files in jug-istanbul/module-2/src/main/java path into src/jugistanbul.workshop2.module2
5. Let’s open terminal and execute command below (Please do not copy/paste)

```
javac -d out --module-source-path out
        --module jugistanbul.workshop2.module1, jugistanbul.workshop2.module2
```

6. You should check
    a. that a folder named as out has been created in workshop-2 folder
    b. that jugistanbul.workshop2.module1 and jugistanbul.workshop2.module2 folders have been created in out folder
    c. that there are compiled .class files in jugistanbul.workshop2.module1 and jugistanbul.workshop2.module2 folders
7. Let's create jar file for each module by executing commands below

```
jar --create --file /jugistanbul_fs/modules/jugistanbul.workshop2.module1.jar -C
out/jugistanbul.workshop2.module1

jar --create --file /jugistanbul_fs/modules/jugistanbul.workshop2.module2.jar -C
out/jugistanbul.workshop2.module2
```

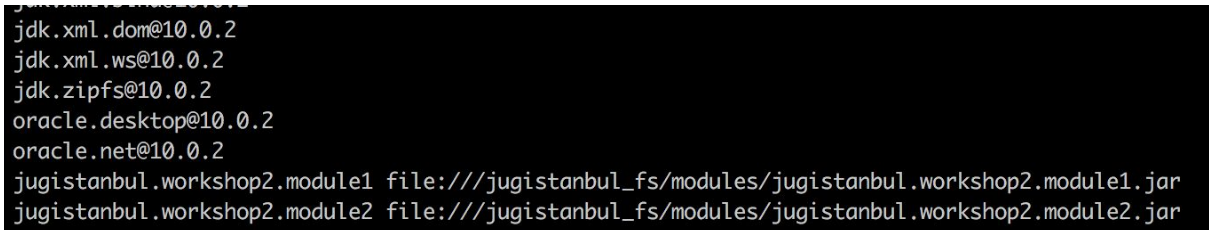
8. Let's check that specified modules have been created in /jugistanbul_fs/modules path
9. Let's check that description of module-info.java without extracting jar file by executing commands below:

```
jar --describe-module --file /jugistanbul_fs/modules/jugistanbul.workshop2.module1.jar

jar --describe-module --file /jugistanbul_fs/modules/jugistanbul.workshop2.module2.jar
```

10. Let's list modules located in /jugistanbul_fs/modules path

```
java --module-path /jugistanbul_fs/modules --list-modules
```



```
jdk.xml.dom@10.0.2
jdk.xml.ws@10.0.2
jdk.zipfs@10.0.2
oracle.desktop@10.0.2
oracle.net@10.0.2
jugistanbul.workshop2.module1 file:///jugistanbul_fs/modules/jugistanbul.workshop2.module1.jar
jugistanbul.workshop2.module2 file:///jugistanbul_fs/modules/jugistanbul.workshop2.module2.jar
```

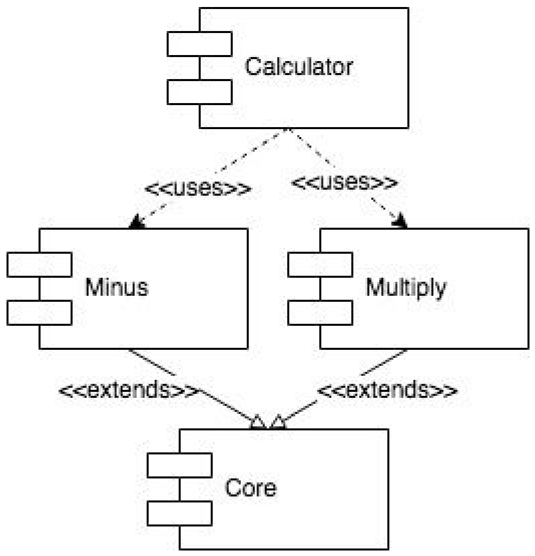
11. You will see that System Modules are being listed with modules that we have created yet. Let's limit the listed modules to specified module by using --limit-modules parameter

```
java --list-modules --limit-modules jugistanbul.workshop2.module1  --module-path
/jugistanbul_fs/modules

java --list-modules --limit-modules jugistanbul.workshop2.module2  --module-path
/jugistanbul_fs/modules
```

WORKSHOP-3 :

Once upon a time, we had created a software that computes complex calculations for a calculator manufacturer. We had used Java 8. After learning that Java 9 is released with module feature, We decided to use this feature. Because we had already designed software as Maven Modules.



While learning Java 9 Module, we figured out that Java 9 still supports loading jar files defined in classpath. Cause of this backward-compatibility, changing source and target to 9 in maven is enough.

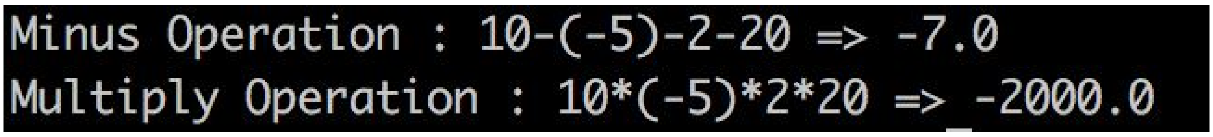
MISSION - 1:
TIME : 30 min.
GOAL : Converting all maven modules to Java 9 modules

1. Let's download calculator project in Workshop-3
2. Let's create module-info.java files and define exports and requires instructions
3. Let's compile and package projects as module. Jar files of modules must be located in /jugistanbul_fs/modules folder
4. Let's execute calculator application by calling command below. There is a main class in jugistanbul-calculator maven modules.

```
java --module-path [MODULE_PATH] --module [MODULE_NAME]/[MAIN_CLASS]
```

- fill the blanks [...] with proper values

5. When you fill correctly the gaps, you must get the result as shown below

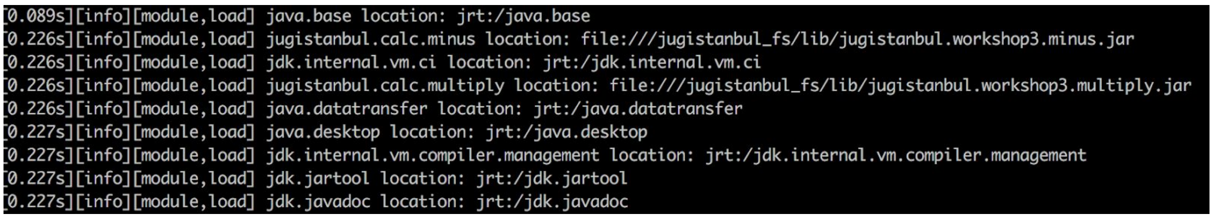


```
Minus Operation : 10-(-5)-2-20 => -7.0
Multiply Operation : 10*(-5)*2*20 => -2000.0
```

7. Let’s verify which modules are being loaded when we execute application. We must use --verbose:module parameter for this purpose.

```
java --module-path [MODULE_PATH] --verbose:module --module
[MODULE_NAME]/[MAIN_CLASS]
```

8. We must take a result as shown below. You will see that listing order is being changed on every executions.

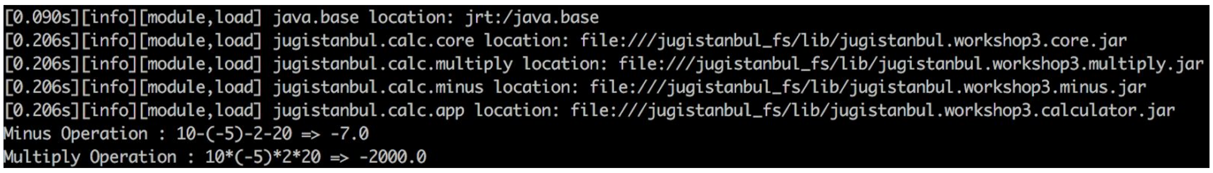


```
[0.089s][info][module,load] java.base location: jrt:/java.base
[0.226s][info][module,load] jugistanbul.calc.minus location: file:///jugistanbul_fs/lib/jugistanbul.workshop3.minus.jar
[0.226s][info][module,load] jdk.internal.vm.ci location: jrt:/jdk.internal.vm.ci
[0.226s][info][module,load] jugistanbul.calc.multiply location: file:///jugistanbul_fs/lib/jugistanbul.workshop3.multiply.jar
[0.226s][info][module,load] java.datatransfer location: jrt:/java.datatransfer
[0.227s][info][module,load] java.desktop location: jrt:/java.desktop
[0.227s][info][module,load] jdk.internal.vm.compiler.management location: jrt:/jdk.internal.vm.compiler.management
[0.227s][info][module,load] jdk.jartool location: jrt:/jdk.jartool
[0.227s][info][module,load] jdk.javadoc location: jrt:/jdk.javadoc
```

9. When we inspect the result, we figure out that many modules that we don’t really need are being loaded. We must use --limit-modules parameter in order to limit.

```
java --module-path [MODULE_PATH] --verbose:module  --limit-modules
jugistanbul.calc.app --module [MODULE_NAME]/[MAIN_CLASS]
```

10. We will get the result shown below:



```
[0.090s][info][module,load] java.base location: jrt:/java.base
[0.206s][info][module,load] jugistanbul.calc.core location: file:///jugistanbul_fs/lib/jugistanbul.workshop3.core.jar
[0.206s][info][module,load] jugistanbul.calc.multiply location: file:///jugistanbul_fs/lib/jugistanbul.workshop3.multiply.jar
[0.206s][info][module,load] jugistanbul.calc.minus location: file:///jugistanbul_fs/lib/jugistanbul.workshop3.minus.jar
[0.206s][info][module,load] jugistanbul.calc.app location: file:///jugistanbul_fs/lib/jugistanbul.workshop3.calculator.jar
Minus Operation : 10-(-5)-2-20 => -7.0
Multiply Operation : 10*(-5)*2*20 => -2000.0
```

11. In order to inspect dependencies of an module, we should use jdeps command. Let’s inspect modules that we have created yet.

```
jdeps --module-path [MODULE_PATH] [FULL_PATH_TO_MODULE_JAR]
```

You read the article about the different usages: https://blog.codefx.org/tools/jdeps-tutorial-analyze-java-project-dependencies/.

MISSION - 2 :

TIME : 15 min.

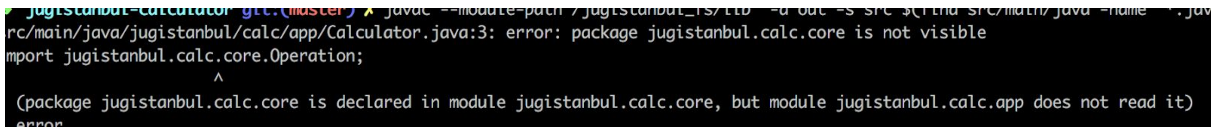
GOAL : We took a feedback from our customer about the code is not flexible. They pointed out that it was hard to assign a math operation to a different calculator button. Because they had to create a reference with type as same as math operation. They asked us to define an abstract class.

However we had already made kind of abstraction. Why can't they use this abstraction?

1. Let's open source code of Calculator class in jugistanbul-calculator maven modules.
2. Let's assign MinusOperation and MultiplyOperation instances to references of Operation.
3. Let's import jugistanbul.calc.core.Operation.
4. Let's compile jugistanbul.calculator module by using terminal
    1. change dir to path of jugistanbul-calculator maven module
    2. execute command below

```
javac --module-path /jugistanbul_fs/lib  -d out -s src $(find src/main/java -name "*.java")
```

5. During compilation, we must have received an error as follows



```
rc/main/java/jugistanbul/calc/app/Calculator.java:3: error: package jugistanbul.calc.core is not visible
mport jugistanbul.calc.core.Operation;
                           ^
 (package jugistanbul.calc.core is declared in module jugistanbul.calc.core, but module jugistanbul.calc.app does not read it)
```

We figure out that jugistanbul.calc.core module is loaded but it is not visible/readable to jugistanbul.calc.app module. There are two ways to make jugistanbul.calc.core readable.:

1. By adding "requires jugistanbul.calc.app module" into module-info.java files of jugistanbul.calc.app module
2. By adding transitive keyword to jugistanbul.calc.app dependency in module-info.java of both jugistanbul.calc.multiply module and jugistanbul.calc.minus module

Let's choose second option:

1. Let's change "requires jugistanbul.calc.core" line to "requires transitive jugistanbul.calc.core" in module-info.java files of both multiply module ve minus module.
2. Let's compile both modules and package them as module.
3. Let's execute command below

```
javac --module-path /jugistanbul_fs/lib  -d out -s src $(find src/main/java -name "*.java”)
```

4. Compilation must have completed successfully

MISSION - 3:

TIME : 15 min.

GOAL : In a meetup about Java 9, we learned that we were able to create a JRE image that includes our runnable application with modules that the application was depended. By creating the image, installation jre to server had not been needed.

Let’s create an image for calculator application:

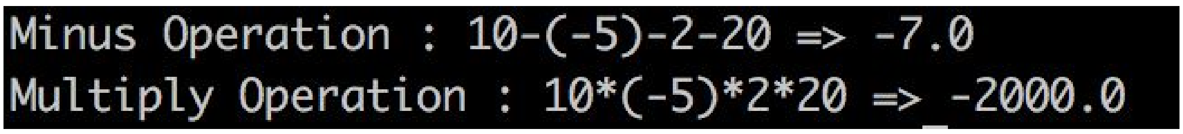
1. Let’s be sure that the modules we created on previous workshop are still in /jugistanbul/lib path.
2. Let’s run the command below to create image

```
jlink --module-path $JAVA_HOME/jmods:/jugistanbul/modules --add-modules
jugistanbul.calc.app --output calculator --launcher
calc=jugistanbul.calc.app/jugistanbul.calc.app.Calculator
```

3. Let’s check that an folder specified with --output parameter must be created.
4. Let’s run image

```
./calculator/bin/calc
```

5. We should see the result below



```
Minus Operation : 10-(-5)-2-20 => -7.0
Multiply Operation : 10*(-5)*2*20 => -2000.0
```

6. Let’s compare files and folders in calculator folder to JDK 9 installation folder
7. You will see that both folders have same folder structure

You can review the different usage of jlink on this article.

WORKSHOP-4 :

TIME : 30 min.

GOAL : Service Modules

SCENARIO:

Let's assume that we are working in software company that has Industrial IOT platform. This platform is collecting sensor datas and processing them to support business decisions. PLC and OPC allow third-party softwares to collect datas. We decided to refactor our software to add a collector for another source. There should be an abstract service layer and collectors to specific sources must implement that service. JDBC driver is a good example.

We have gained the experience about how we create an module. We are ready to convert dataservice project from maven modules to Java 9 Modules.

MISSION :

1. Let's start to convert maven modules to Java 9 Modules.
2. Let's define PLC and OPC modules as service provider.
    a. Let's put the instruction below with correct values into module-info.java.

```
provides [SERVICE_INTERFACE] with [SERVICE_IMPLEMENTER_CLASS];
```

    b. Let's create modules in /jugistanbul_fs/modules folder.

5. Let's review the source code of DataServiceProvider class in iotapp project
    b. Do you know that ServiceLoader class has been existed since Java 1.6. You can see the usage inf Java 1.6 by reading this article.
    c. We should know that usage of ServiceLoader has been become more easy by putting provides...with... and uses… instructions to module-info.java in Java 9.
6. Let's put the definition below into module-info.java of iotapp in order to access dataservice.

```
uses [SERVICE_INTERFACE];
```

7. Let's create iotapp module in /jugistanbul_fs/modules folder.
8. Let's execute main class in iotapp module.
9. If you have time, you can create an image of iotapp. In order to create an image, you should read the info below..

Jlink follows requires instruction while creating an image. But, it doesn’t follow the uses and provides instructions. You must add implementer modules by one by. There is --add-module parameter in order to do this or there is an another way that using --bind-services parameter. Jlink resolves service dependencies automatically if --bind-services parameter is used. But this parameter doesn’t provide an optimal solution. Because it will add all service provider modules that are not deal with the application. In order to inspect/list the required service provider modules, there is --suggest-providers parameter. After deciding which modules are being used, you must add each module by using --add-module parameter.

JLink needs that all packages are created as a Java 9 Module. Because of that, it doesn’t accept Automatic Modules and Unnamed modules.

You can follow the examples in https://github.com/tanerdiler/java9 for other new features coming in JAVA 9.

WORKSHOP-5
GOAL: A near-real world example of getting a Spring Boot application up using Java module set-up.
Code (Includes Spoilers):
https://github.com/JUGIstanbul/java_9_10_workshop/tree/master/spring-boot-workshop

Set-up: There are two modules; one of which is a slightly modified version of the Slugifier. It does not expose the package of “TurkishCharacterConverter”.

The other one is a mere Java 10 Web application initialized using https://start.spring.io. The only dependency used is “Web”. To choose Java 10, one must open the “advanced options” on the page.

The Spring Boot application contains a “SlugifyController”, which processes given string using Slugifier module. There are no sessions, no databases, no whatsoever.

The Challenge: There are two challenges in this case: The first is the relationship between Jigsaw and Maven (or any other build tool) - in fact, there are none. Your build tool gathers dependencies and makes sure they are in the classpath on compile time, and preferably package them together for runtime. The Jigsaw is responsible for determining which module can access which other, and in what way and/or restriction set.

The second challenge is the module-info.java file. Now that we are using Java 9/10 as the runtime, all our code is a big fat anonymous module. Using module-info.java file, we can expose or restrict access to our code in any granularity level we would like (well, maybe not “any” :)). After we edit the module-info.java files, we can compile and run the application.

Well, in most cases; defining compile time module dependencies in module-info.java won’t cut it; because Spring - like most frameworks - relies on reflection to bind component dependencies. This means module-info.java files must also define runtime dependencies.

One last thing, any package we want to enable access via reflection must be exposed by “opens” statement in corresponding module-info.java (like Spring Controllers).

Epilogue: Getting a pre-existing Java 8 app up and running using Java 9+ modules is not an easy task, since it requires repetitive try-and-error tasks writing module-info.java files; but once it is set up, the successive tries will take much less time.

One last word, Spring 2.0+ is better at supporting Java Jigsaw modules.