1. Create a main-class in a module1 that instantiates an object of a Book-class (attributes title, yearOfPublication) in a module2 and prints out the values of that object. Use Jigsaw!

   Add the correct module-information to both projects. Module2 exports the package containing the Book-class, module1 requires module2.

   If you use Eclipse or another IDE, put both modules in different projects. In eclipse add the project of module2 in Project → Properties → Java Build Path → Modulepath

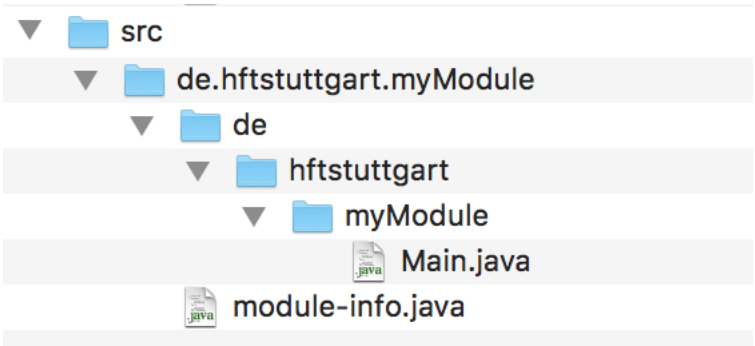
   After successful execution of the Main-class create a modular jar-file for your module1 (Export in Eclipse). Check the results in command-line using the command:

   ```
   jar --file module1.jar --describe-module
   ```

   The result should be something like:

   ```
   de.hftstuttgart.module1 jar:file:///dummy/module1.jar/!module-
   info.class
   requires de.hftstuttgart.module2
   requires java.base mandated
   ```

2. Use the internal class “sun.net.URLCanonicalizer” in a new main-class that uses modules. This is the directory-structure you should create:

   This is the Java main-class:

   ```
   package de.hftstuttgart.myModule;
   import sun.net.URLCanonicalizer;
   public class Main {
     public static void main(String[] args) {
       URLCanonicalizer urlCanonicalizer = new URLCanonicalizer();
       String url = urlCanonicalizer.canonicalize("www.google.de");
       System.out.println(url);
     }
   }
   ```

   This is the module-descriptor:

```
module de.hftstuttgart.myModule {
}
```

Now try to compile that on the command line (be in the path of your project) using:

```
javac src/de/hftstuttgart/myModule/Main.java
```

You should get an error that the package **sun.net** is not visible. This is a deprecated package that is not officially supported and will be removed in the future. To compile it anyway you must include the compiler-option **--add-exports**:

```
javac -d bin
   --add-exports java.base/sun.net=de.hftstuttgart.myModule
   src/de/hftstuttgart/myModule/Main.java
```

Using the above option allows access of the **sun.net**-package by our module. Now it compiles (with only warnings). To run it you also have to include the option **--add-exports**

```
java
   --module-path bin
   --add-exports java.base/sun.net=de.hftstuttgart.myModule
   -m de.hftstuttgart.myModule/de.hftstuttgart.myModule.Main
```

Additional exercises (OSGi)

3. Create a simple OSGi-bundle that displays a welcome-message when started and a goodbye-message when stopped (use the example in the lecture notes as template).

   Please use exactly the names as given here!

   Steps:
   a) Create a new “Plug-in Project” with the name “HelloWorld”
   b) In the “New Plug-in Project wizard” select in “Target Platform → This plug-in targeted to run with” the option “an OSGi framework” and select “standard”.
   c) Copy the code from the lecture notes into the MANIFEST.FM
   d) Create a new Java-Class in the Default-package with the Name “HelloActivator” and copy the code from the lecture notes to that class
   e) Select the project an select “Run as” → “Run Configurations”
   f) On the tab “Bundles” select “Deselect All”. Then select you bundle “HelloWorld (1.0.0.qualifier” and select “Add Required Bundles”
   g) Press “Run” and wait
   h) After “osgi>” is shown type “ss” and search the bundle-id of your HelloWorld-Bundle
   i) Stop the bundle using the bundle-id

   In steps g) and i) you should see the Bundle printing out the messages of your activator.

   Attention: Don’t forget the empty line at the end of the file manifest.mf!

   Attention: If the OSGi-prompt does not show (and you see some exceptions in the console) add the following bundles to your runtime configuration:

   org.eclipse.equinox.console.jar
   org.apache.felix.gogo.shell.jar
   org.apache.felix.gogo.command.jar
   org.apache.felix.gogo.runtime.jar

4. The task is to create a “message of the day” service. This means there is a component you can ask for a message of the day and this component returns a random string.

   a) Create a bundle including an interface-package (including one interface that acts as the interface to the service) and an additional implementation-package (including the implementation-class of your service → there is a template for the implementation in the E-Learning-System).
   b) Add another class to that bundle in the interface-package that creates and registers a “message of the day” service (as OSGi-service). Here you need an activator that registers the service in the start-method.
   c) Create another bundle that acts as client. This client should use the service created in b). Again, use an activator and display the message retrieved by the service in the start-method.

   Attention: Set your export- and import-settings in the manifest-files in a proper way!