Enterprise Java with Spring
Intro to Maven
Lab 1

1 Lab setup

Make sure you have the following items installed

- Latest LTS JDK version (at this point: JDK 21)
- A suitable IDE (Eclipse Enterprise Edition for Java) or IntelliJ IDEA
- Latest version of Maven (at this point: Maven 3.9.9)
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

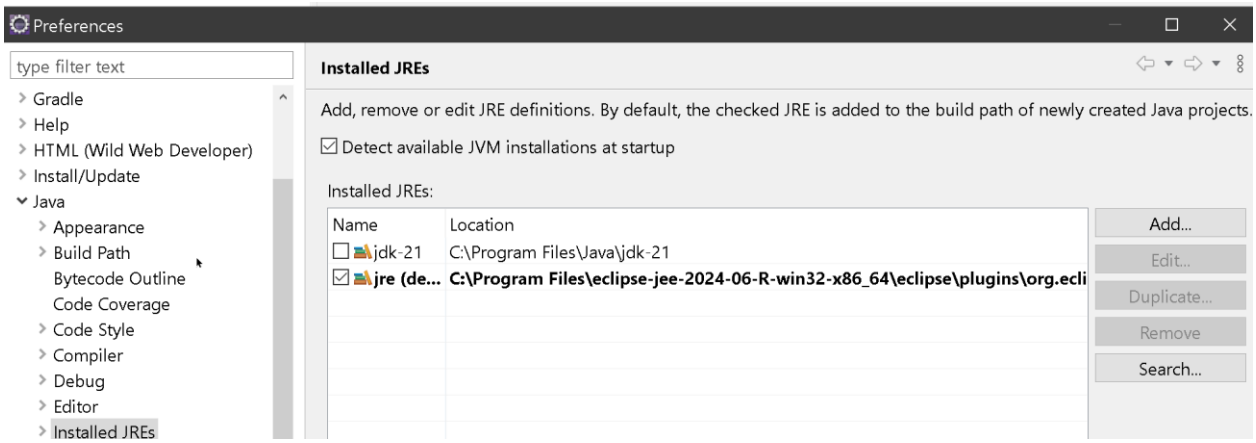
In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder holds the source code and other related files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` subfolder to build up our applications from scratch and you can always fall back on the complete Eclipse project if you encounter any errors while building up the application.

2 Compiler / JRE settings in Eclipse

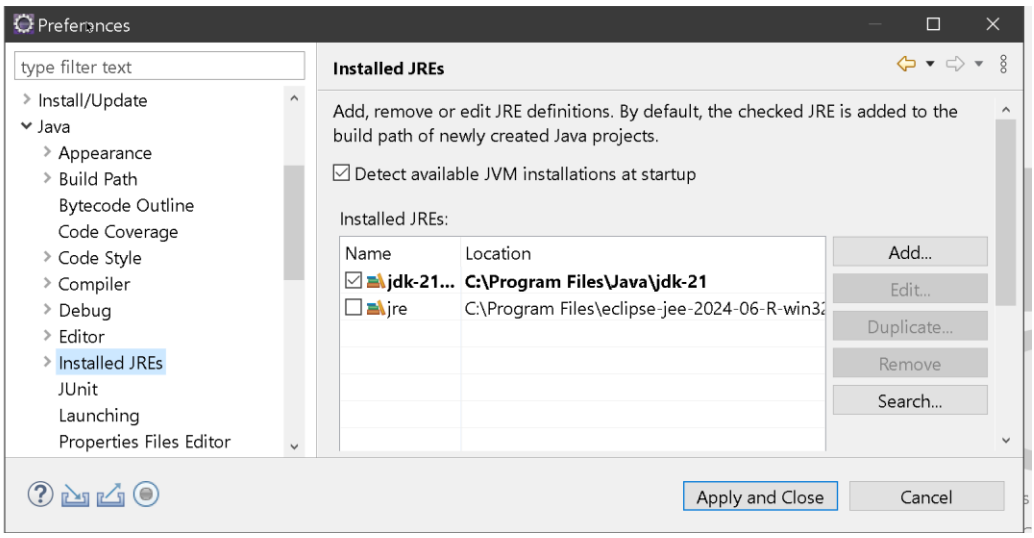
For a new Eclipse installation and also when you change to a new workspace, you may need to double check to ensure the compiler / JRE is set properly to ensure that your programs compile properly and Maven can execute correctly.

Newer versions of Eclipse already include a JRE as part of the installer process and usually do not require any further modification. The JRE version is typically 16 or higher. To avoid potential issues with Maven, it is best to configure Eclipse to use the JDK that you have installed locally rather than the Eclipse built-in JRE.

To do this, go to Window -> Preferences, and in the Dialog Box, Java -> Installed JREs. Eclipse will detect available JVM installations by default at start up and your local JDK should be shown here.



If Eclipse already detects this local JDK, you can directly select it in the Installed JREs dialog box and click Apply and Close.

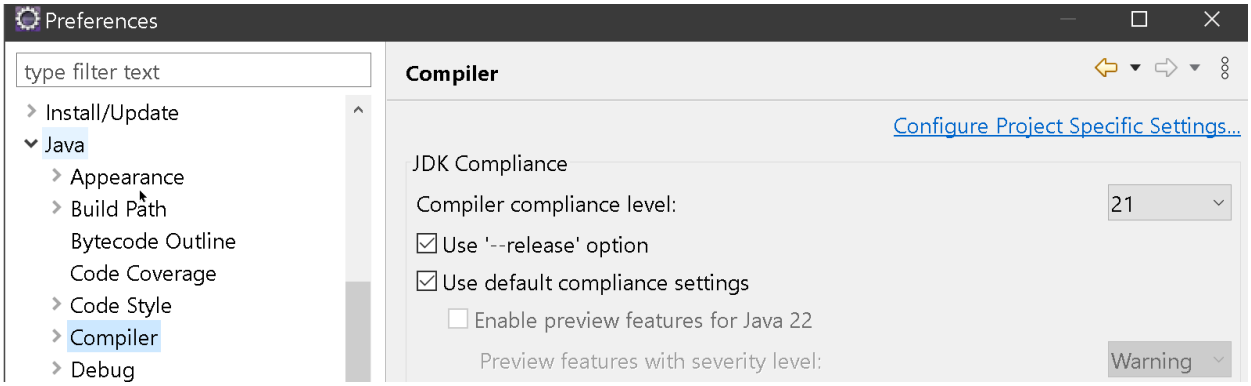


If Eclipse is unable to detect your local JDK, then you need to manually add it yourself.
Click Add. In the JRE Type dialog box, select Standard VM. Click Next.
In the JRE Home entry, click the Directory button and navigate and select the installation directory of the JDK, for e.g: C:\Program Files\Java\jdk-21

If you do not do this on older versions of Eclipse, you might get this particular error message during the Maven build:

```
[ERROR] No compiler is provided in this environment. Perhaps you are
running on a JRE    rather than a JDK?
[INFO] 1 error
```

Next, make sure that the compiler compliance level (accessible in the same Dialog Box from previously, Java → Compiler) is aligned with your local JDK

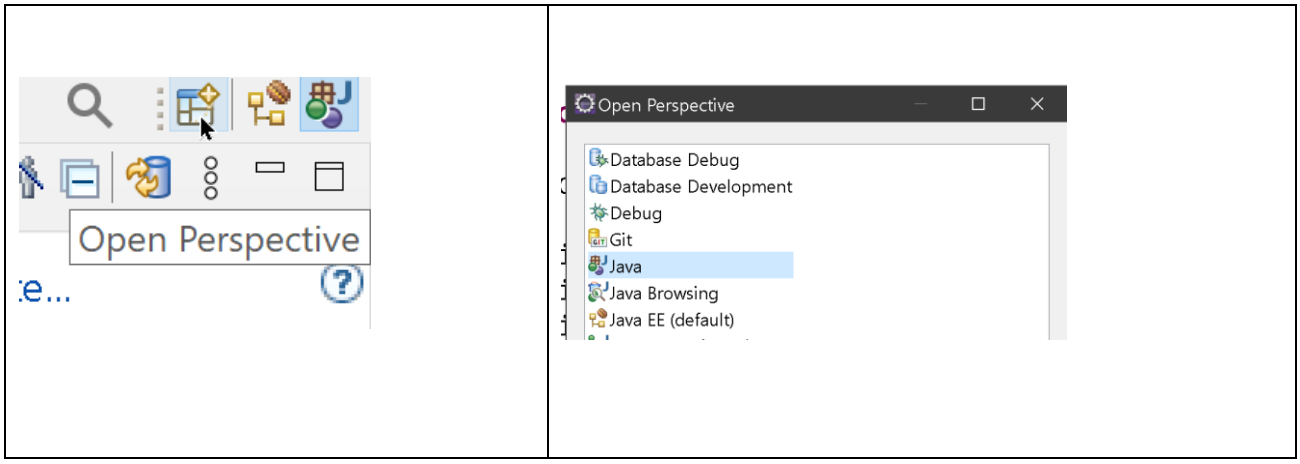


All newly generated projects will then use these settings.

3 Generating and using a JAR

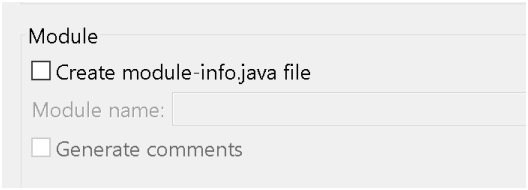
The source code for this lab is found in `demo-jar/changes` folder.

Switch to Java SE perspective, either using the icons at the upper right hand corner or via the menu: Window -> Perspective -> Open Perspective



Create a new Java project (File -> New -> Java Project).
For the project name, type: `SimpleFirstProject`
Ensure that the execution environment JRE is set to the default JRE that you had configured earlier, for e.g. `jdk-21`

Make sure to uncheck the Create module-info.java file option.



Then click Finish.

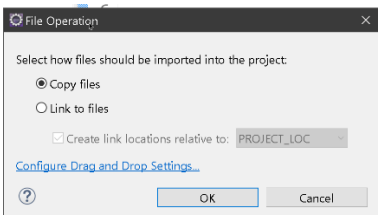
NOTE: The `module-info.java` file in a Java project is a core part of the Java Platform Module System (JPMS), introduced in Java 9 as a way to modularize Java applications. It allows developers to explicitly specify module dependencies and control which packages are exposed to other modules. We will not be using it here for this simple demo.

Create a new package (Right click on the project name, select New -> Package): `com.workshop.operations`

In this package, place these files from `changes`:

`StringOperations.java`
`MainProgram.java`

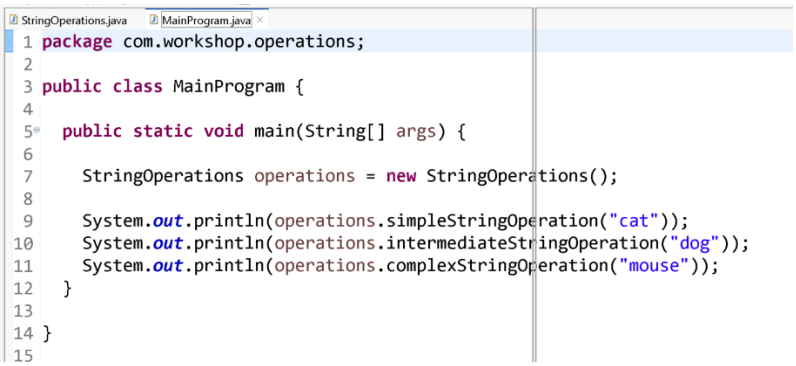
You can simply drag and drop these files from the `changes` folder into the specified package name in the Package Explorer, and select the copy operation:



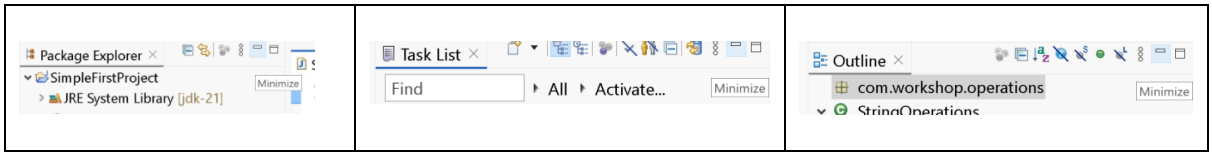
From the main menu, Window -> Editor allows you to customize a few useful options when working with your source code

- Zoom in / Zoom out
- Toggle Word Wrap

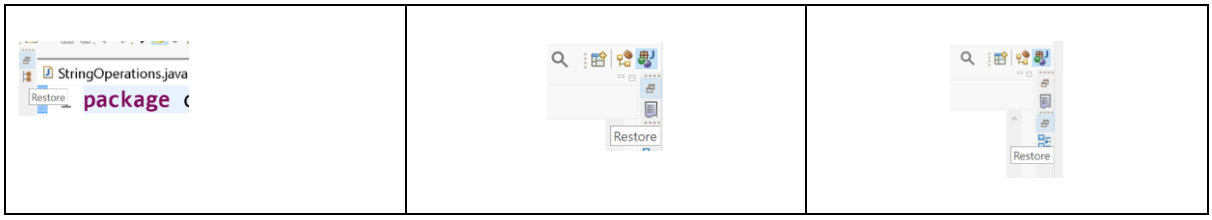
If you want to be able to view multiple editor tabs for different source code files simultaneously side by side, simply click on the tab of an open editor and drag it to the right or left, as shown below.



You can subsequently minimize and restore the different views on the left (Package Explorer) and right (Task List and Outline) to provide more space to view the editors with the source code files.



You can then restore these views again when you need to use them.



You can now run `MainProgram` to verify that it works. With MainProgram as the active editor tab, click anywhere in the editor and from the context menu, select Run As -> Java Application.

Verify that it produces the expected result. The output should appear in the Console view among the stack of views below the main editor area.

```
Problems  Javadoc  Declaration  Console
<terminated> MainProgram [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe  (26 May 2025, 2:00:23 pm – 2:00:2
simpleString operation received input argument : cat
intermediateStringOperation received input argument : dog
complexStringOperation received input argument : mouse
```

We will now create an executable, standalone JAR for this project.

Right click on the project name. From the context menu, select Export. In the Export dialog box, select Java -> Runnable JAR file.

Select a suitable folder as the export destination folder (by default, this will be your worksplace folder) and name the JAR as: `StringOperations.jar`
Note: You can also place this generated JAR in any other folder on your machine.
For the Launch configuration, select `MainProgram – SimpleFirstProject`.
Click Finish.

Open a command prompt or shell terminal and navigate to the destination folder you had exported the JAR to.
If you had placed the JAR in workspace folder earlier, an easy way to do this within the IDE is:
Right click on the project name, from the context menu, select Show in Local Terminal -> Terminal.
In the terminal, type the following to navigate back up to the workspace folder and check for the existence of the newly generated JAR file.

```
cd ..
dir
```

In the command prompt in the destination folder containing the JAR, run the JAR with:

```
java -jar StringOperations.jar
```

Verify that the execution produces the expected result.

Use 7-z (or any other suitable archive app) to view the contents of the JAR, verify that it contains the compiled classes from the project as well as a manifest (MANIFEST.MF) which indicates the main class to run. You can also unzip the JAR to view its contents.

Create another Java project using the same approach as before: `SimpleSecondProject`
Create a new package within this project: `com.workshop.user`

In this package, place this file from `changes`:

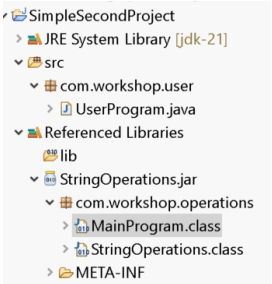
`UserProgram.java`

Notice that the source code here is identical to MainProgram. Java.
However, here the package flags an error as the StringOperations class that is requires (which we call a dependency) is not on its project build path.

To solve this problem, we need to add this class (StringOperations) to the build path, which we can do by simply copying the required class over to this package. However, we already have this class in the JAR file we generated earlier. Since JAR files are the universal method of distributing classes for Java projects, we will place the JAR file in the project build path.

In `SimpleSecondProject`, create a new folder named `lib` (right click on the project name and select New -> Folder). Copy `StringOperations.jar` into this folder. You can double click on it to view the contents in the Editor view via the Eclipse JAR file viewer.

We now need to add this JAR to the build path for this project.
Right click on the project, select Properties -> Java Build Path
Click on the Libraries Tab and select the Classpath entry.
Next click on Add JARs. Select the JAR file that you have just copied into the `lib` folder. Click Apply and Close.
Notice the syntax error flagged earlier in `UserProgram` should disappear as the StringOperations class is now on the project build path.
You can expand the JAR file to see its contents in the Referenced Libraries entry in the Package Explorer.



Verify that you can execute it successfully (right click on UserProgram in the Package Explorer view, select Run as -> Java Application

In this simple example, `StringOperations.jar` is a dependency of `SimpleSecondProject`
The dependency must be placed on the build path as well as the runtime path of `SimpleSecondProject` in order for it to be executed successfully.

We now repeat the process of generating a JAR from this project.
Select a suitable export destination folder (by default the workspace folder) and name the JAR as: `User.jar`
For the Launch configuration, select `UserProgram – SimpleSecondProject`
Click Finish.

A dialog box pops up warning about the repacking of referenced libraries. This indicates that Eclipse will now include the contents of the previous JAR (`StringOperations.jar`) that is placed on the build path of the current project by extracting the package structure of that JAR and placing it into the newly generated JAR. This ensures that the classes of that JAR remain available to this project so that it can be run as a standalone executable JAR.

Use 7-z (or any other suitable archive app) to view the contents of `User.jar`, verify that it contains the compiled classes from this project and from `StringOperations.jar`, as well as the standard manifest (MANIFEST.MF) which indicates the main class to run. You can also unzip the JAR to view its contents.

Open a command prompt or shell terminal in the same way that we did previously (or use the existing terminal that you opened if you placed the JAR file in the workspace folder), navigate to the destination folder and run the JAR with:

```
java -jar User.jar
```

4 Using JARs for popular external libraries (Logback and JUnit)

Logback is a popular logging framework that is a successor to the older log4J project. Logging is a common activity used in many production grade applications.
JUnit is the de-facto library for unit testing in Java.

The source code for this lab is found in `demo-logback-junit/changes` folder.

Switch to Java SE perspective.

Create a new Java project: `SimpleLoggingProject`

Create a new package: `com.workshop.operations`

In this package, place this file from `changes`:

`BasicLoggingDemo`

Notice that there are a variety of syntax errors flagged in the editor as the necessary classes in the import statements are not present on the build path.

In the project, create a new folder `lib`.

Copy the following JARs from the `demo-logback-junit/jars-to-use` folder into `lib`:

```
logback-classic-x.y.z.jar
logback-core-x.y.z.jar
```

`slf4j-api-x.y.z.jar`

We will again add these JARs to the build path for this project.
Right click on the project, select Properties -> Java Build Path
Click on the Libraries Tab and select the Classpath entry.
Next click on Add JARs. Select the JAR files that you have just copied into the `lib` folder. Click Apply and Close.
The syntax errors in `BasicLoggingDemo` should disappear.

Notice that you can expand the JAR files to see their contents in the Referenced Libraries entry in the Package Explorer. The two classes that you just imported in `BasicLoggingDemo` can be located inside `slf4j-api-x.y.z.jar`. These classes in turn use other classes found in the 2 other JAR files in order to execute properly.

Verify that you can execute it successfully (right click, select Run as -> Java Application.
The output you see in the Console view is the standard output that a logging framework typically produces. We will examine logging frameworks in more detail in a subsequent lab.

If you go to the [main download page] for the Logback project and scroll to Binaries in Maven Central topic, you will notice that there are no JARs for you to download directly. Instead this section directs you to the [Maven central repository] in order to download the JAR files that we included in the `lib` folder earlier. We will examine this in more detail in a coming lab.

Create a new package: `com.workshop.test`

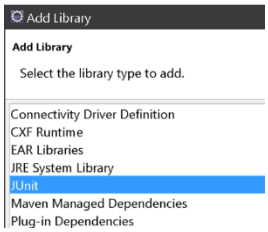
In this package, place these files from `changes`:

`BasicCalculatorTest`

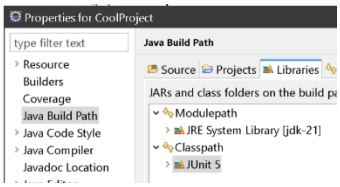
Notice again that there are syntax errors flagged in the class as the required classes from the JUnit library are not on the build class path yet.

The JAR files containing the required classes for the JUnit 5 library is already included with Eclipse, so we do not need to manually add them to the lib folder and then add to the Java Build Path of our project, the way that we did with the JAR file for the Logback library.

To add this JAR files to our Java Build Path:
Right click on the project, select Properties -> Java Build Path
Click on the Libraries Tab and select the Classpath entry.
Select the Add Library button on the right.
In the Add Library dialog box select JUnit and click Next.

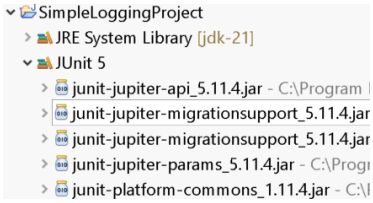


In the next Dialog box, select JUnit 5 and select Finish.



Click Apply and Close in the Properties dialog box.

The previous flagged syntax errors should disappear and an additional JUnit 5 library icon appears in the project hierarchy, which you can expand to see all the relevant JAR files that have now been added to the build path.



To see the JUnit test in action, go to `BasicCalculatorTest` and right click and select Run As -> JUnit Test. The two basic test methods in this simple demo test case (marked with the @Test annotation) should complete successfully and these results in shown in a separate JUnit tab next to the Package Explorer.