Maven Workshop
Lab 2

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 11 (note: labs are tested with JDK 11 but should work with higher versions with no or minimal changes)
- Eclipse Enterprise Edition for Java (or a suitable alternative IDE for Enterprise Java)
- Latest version of Maven
- 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.

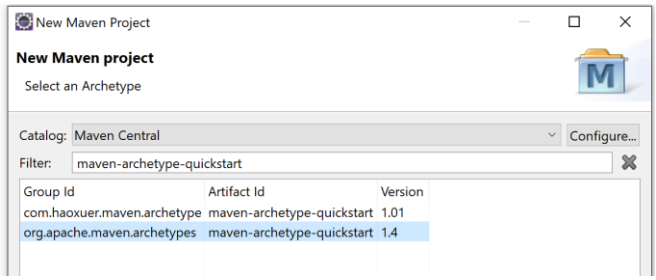
2 Creating a Maven project using an archetype

Maven provides a variety of basic archetypes to generate a template for a project
https://maven.apache.org/guides/introduction/introduction-to-archetypes.html
For a basic Maven project that will run as a command line application, the `maven-archetype-quickstart` archetype is adequate. We will use it here.

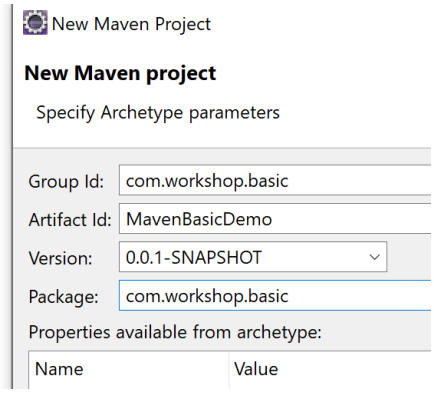
The source code for this lab is found in `maven-basic-demo/changes` folder.

Switch to Java EE perspective

Start with File -> New -> Maven Project. Select Next and type `maven-archetype-quickstart` in the filter. Eclipse may freeze temporarily while it attempts to filter through all the archetypes available at Maven Central. Select the entry with group id: `org.apache.maven.archetypes` and click Next



Enter the following details below and click Finish.



In the newly generated project, open the project POM (`pom.xml`)

Notice that the first 3 elements (the GAV coordinates of the POM) matches the values that you entered earlier.

```
<groupId>com.workshop.basic</groupId>
<artifactId>MavenBasicDemo</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

2.1 Changing Java version

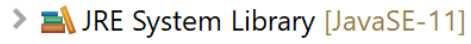
The autogenerated POMs for Maven projects created from archetypes are typically based on older versions of Java. So the first thing we will typically do in a newly generated Maven project is to change the Java version to the correct one. Make this change to the <properties> section in the POM:

```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>
```

The Java version is specified in both the `<maven.compiler.source>` and `<maven.compiler.target>` elements.

Save the POM. To ensure that these changes take effect, right click on the project, select Maven -> Update Project, and click OK in the dialog box that appears. This is something you should always do in Eclipse every time you make a change in the POM.

You should see the JRE system library entry in the project list update to JavaSE-11.



A more updated way of changing the Java version (which is recommended for versions 9 and above) is to configure the `maven-compiler-plugin`

In the <build> section, locate this plugin:

```
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
```

and replace it with this detailed configuration

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <release>11</release>
    </configuration>
</plugin>
```

You can now remove the `<maven.compiler.source>` and `<maven.compiler.target>` elements from the <properties> section.

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Again, to ensure that these changes take effect, right click on the project, select Maven -> Update Project, and click OK in the dialog box that appears.

2.2 Adding in dependencies in <dependencies>

We are now going to add in the JARs that we used dependencies in a previous project.

Replace the existing <dependencies> section with the dependency snippets for the Logback and JUnit libraries:

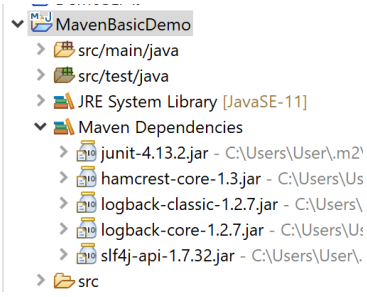
```
<dependencies>

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
  </dependency>

  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.7</version>
  </dependency>

</dependencies>
```

Expand on the Maven dependencies entry in the project. You should be able to see the 5 JAR files that we used in `SimpleLoggingProject` in the previous lab:



The location of these JAR files are stated next to them. When you specify a dependency in your POM, Maven automatically fetches it from the central Maven repository and stores them into the local Maven repository cache on your machine. The default location for this local cache is:

Windows: `C:\Users\<User_Name>\.m2\repository`
Linux: `/home/<User_Name>/.m2/repository`

Mac: `/Users/<user_name>/.m2/repository`

Using File Explorer, you can navigate to the respective directories listed and verify the existence of the JARs there.

Copy `BasicLoggingDemo` from the previous `SimpleLoggingProject` and paste it into the package `com.workshop.basic` in `src/main/java`

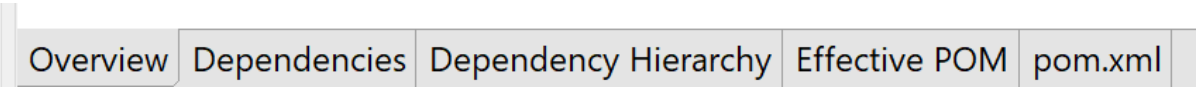
Open `BasicLoggingDemo` in the editor and execute it as usual (Run As -> Java application). Verify that the result is exactly the same as it was in `SimpleLoggingProject`

Copy `TestJUnit` and `TestRunner` from the previous `SimpleLoggingProject` and paste it into the package `com.workshop.basic` in `src/main/java`

Open `TestRunner` in the editor and execute it as usual (Run As -> Java application). Verify that the result is exactly the same as it was in `SimpleLoggingProject`

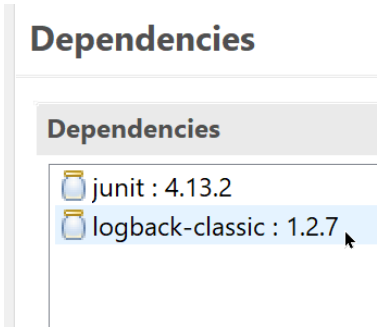
3 Key elements of the POM

When the project POM is open in the Editor view, the bottom part of the editor provides 5 different tabs for 5 different perspectives on the same POM: Overview, Dependencies, Dependency Hierarchy, Effective POM and pom.xml.



The Overview tab allows you to view and edit some (but not all) of the POM elements in a more user-friendly manner. For e.g. you can specify the project GAV coordinates and also the project details, organization, etc, here.

The Dependencies tab shows you all the dependencies currently listed in the <dependencies> section of the POM. These are the direct dependencies. Notice only the artifact-id and the version number is shown. There is no <dependency Management> section currently defined in the POM, so it is empty here.




```
<dependencies>

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
  </dependency>

  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.7</version>
  </dependency>

</dependencies>
```

3.1 Transitive dependencies and dependency mediation

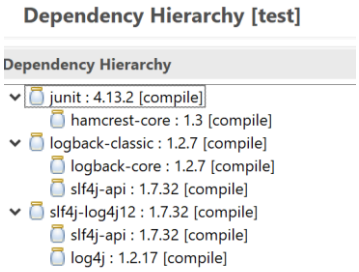
The Dependency Hierarchy tab shows you the direct dependencies and their transitive dependencies. Transitive dependencies are dependencies required by the direct dependencies in order to execute properly. The chain of transitive dependencies can extend to any length (and it will for complex projects such as Spring Boot), but in this simple project, it only extends to a depth of 1. Junit has one transitive dependency, while logback-classic has two.

The resolved dependencies section shows the actual dependencies that will be used in the project build after dependency mediation has been performed. Dependency mediation refers to how Maven resolves version conflicts for a transitive dependency. At this point, there are no conflicts, so there is no need to perform mediation.

We will now introduce a new direct dependency. Switch back to the pom.xml tab and add this <dependency> snippet right at the end of the <dependencies> section and save the change:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.32</version>
</dependency>
```

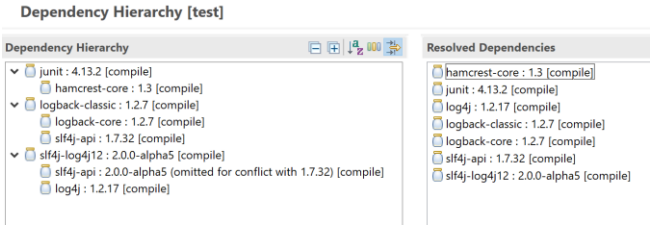
Switch back to the Dependency Hierarchy tab. You can now see that there are now two instances of slf4j-api (since it is a transitive dependency of both logback-classic and slf4j-log4j12 (which are both direct dependencies). However, there is no conflict here because the version is identical in both instances.



Switch back to the pom.xml tab and change the version of the new dependency and save:

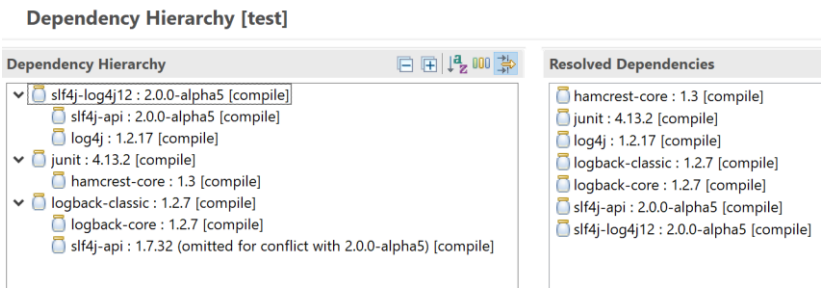
```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>2.0.0-alpha5</version>
</dependency>
```

Switch back to the Dependency Hierarchy tab. You can now see that the two instances of slf4j-api are of different versions and hence a conflict arises (since only one version can be used in the build process). Maven's dependency mediation process decides which one to use and the view will indicate the version that is omitted (2.0.0-alpha5) while the actual version to be used (1.7.32) is shown in the resolved dependencies view.



The way that dependency mediation works in Maven is explained in:
https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#transitive-dependencies

Switch back to the pom.xml tab and move the new dependency snippet right to the top of the <dependencies> section and save. Now back in the Dependency Hierarchy tab, we can now see the situation is reversed: version 1.7.32 is now omitted while version 2.0.0-alpha5 is used.



The way that Maven resolves dependency conflicts will have consequence on whether the application runs correctly or not. Keep in mind that this specific version of logback-classic API was tested and guaranteed to work only against a specific version of slf4j-api (which of course is the reason why that version is included as a transitive dependency).

When Maven uses a different version of slf4j-api instead, the result is unpredictable and depends on a variety of factors such as the difference between the two versions and whether the logback-classic API is using classes form slf4j-api that significantly differ between both versions. Thus, we now use any class from logback-classic in our application code that in turn depends on a class from slf4j-api, the code may not function at all or may produce some unexpected results. Remember to ALWAYS check for transitive dependency conflicts when your project behaves in an unexpected manner.

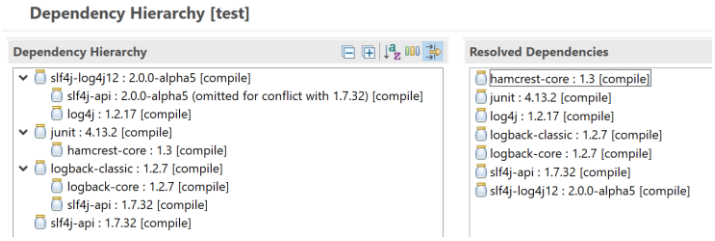
One way to force Maven to use a specific version of a transitive dependency (thereby overriding the results of its dependency mediation process), is to specify that transitive dependency as a direct dependency instead.

Switch back to the pom.xml tab and add in the transitive dependency as a new dependency right at the end of the <dependencies> section and save:

```
    <dependency>
```

```
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.32</version>
    </dependency>
```

Now back in the Dependency Hierarchy tab, we can now see that with 3 instances of slf4j-api in the hierarchy, the version specified by the direct dependency (1.7.32) is now the one that will be used.



You can now remove both additional dependencies that we introduced earlier, so that the <dependencies> section is restored back to its original state, then save:

```
  <dependencies>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
    </dependency>

    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.2.7</version>
    </dependency>

  </dependencies>
```

3.2 Super POM and effective POM

The Super POM is Maven's default POM, which contains basic configuration for the key elements of a project POM in accordance to Maven's philosophy of Configuration over Convention (COC). The configuration specified in the POM for any Maven project is then added onto the super POM to create the effective POM. The configuration in the effective POM is what is actually applied to project in question.

https://maven.apache.org/pom.html#the-super-pom

To view the actual content of the super POM, we simply remove all non-essential elements from our current POM. IN the pom.xml tab, remove the following sections:
`<properties>`
`<dependencies>`
`<build>`
and save:

The project POM should now look like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.workshop.basic</groupId>
  <artifactId>MavenBasicDemo</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <name>MavenBasicDemo</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

</project>
```

Switch over to the effective POM tab and the contents shown here are those of the super POM. Some of the key elements and values here:

- The location of the Maven central repository from which all the dependency JARs are downloaded from is specified: https://repo.maven.apache.org/maven2
- Various directories are specified: directory to hold code for the main application, code for unit tests, compiled classes from these two areas, resources used by the main application, resources used by the unit tests, etc
- The <pluginManagement> section contains a list of <plugin> elements which are used to specify relevant configuration info for the key plugins used in Maven build process. Here we only specify the version numbers to fix them for whenever these plugins are subsequently used (which is the typical use for a <pluginManagement> section).
- The <plugins> section (which is separate from the <pluginManagement> section) contains the list of core plugins and their configurations that Maven will use in various parts of the default lifecycle build process. Notice that their specific goals are specified as well as which phase of the default life cycle they are bound to (https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#built-in-lifecycle-bindings)

3.3 Plugin configuration using <pluginManagement>

An important point to note here is that plugins declared within <plugin> elements that are placed within the larger <pluginManagement> section will NOT take effect in the current project or any inherited child projects. For a plugin to be accessible in order for its goals to be executed, it must be declared in a <plugin> element within a separate <plugins> section OUTSIDE of the <pluginManagement> section, but still within the main <build> section.

The plugins that are declared in the separate <plugins> section OUTSIDE of the <pluginManagement> section are:

- maven-clean-plugin

- maven-resources-plugin
- maven-jar-plugin
- maven-compiler-plugin
- maven-surefire-plugin
- maven-install-plugin
- maven-deploy-plugin
- maven-site-plugin

They are therefore accessible and available to used in the build process in our project POM.

The plugins that are declared in the <pluginManagement> section with only version info specified:

- maven-antrun-plugin
- maven-assembly-plugin
- maven-dependency-plugin
- maven-release-plugin

This means that if you wish to use them in any Maven project, you must explicitly declare them in a separate <plugins> section in the POM. They will automatically inherit the configuration information defined here, unless you override it with new configuration info.

Copy in the `pom.xml` from `changes` into the project to restore the contents back to its previous state.

In the pom.xml tab, we can see that this project POM includes a <pluginManagement> section that contains the same plugins defined in the <plugins> section OUTSIDE of the <pluginManagement> section in the super POM. The purpose of this section here is to simply override the version info provided for these plugins in the super POM.

To see the list of all relevant Maven plugins and their latest versions
https://maven.apache.org/plugins/index.html

You will notice that the versions provided in the project POM (which are autogenerated from the archetype used to create this project) are not necessarily the latest: so you can choose to update them yourself manually here.

3.4 Test and runtime dependency scopes

https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#dependency-scope
Scopes not only specify which classpath a dependency will ultimately be made available on (compile time, test, run time) but also whether that dependency is transitive or not.

The important point to keep in mind here other than knowing when to use a specific scope, is that if any of the dependencies have either the `runtime` or `test` scope, they will not be available for compiling your code. For e.g. if you change the dependency of JUnit in the POM to below and save and perform a Maven -> Update Project:

```
    <dependency>
      <groupId>junit</groupId>
```

```
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
```

Syntax errors will now be flagged in both TestJunit and TestRunner which utilize classes from the JUnit dependency (since it will no longer be available to them at compile time). Also if you check in the Maven Dependencies entry, the 2 JARs associated with this dependency are now shaded, indicating that they are not accessible for compile time use.

Remove the `test` scope from the dependency snippet and replace this with `runtime` scope. The same issue is observed.

Finally, remove the `scope` element altogether (this sets the dependency to the default scope of `compile`). Notice now that the application code compiles fine and the JARs are shown to be available in the Maven Dependencies entry

4 Executing Maven commands

Maven has 2 different invocation modes which can be used when interacting with it through its command line tool `mvn` or via the Eclipse Maven integration. Note that Eclipse ships with its own embedded Maven (M2Eclipse) that does not rely on a local Maven installation. To see the specific version:
Windows -> Preferences -> Maven -> Installations
https://www.eclipse.org/m2e/index.html

Each of the 3 core build lifecycles in Maven (`default`, `clean` and `site`) is defined by a different list of build phases (or stages) in the lifecycle.

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#lifecycle-reference

A particular build phase is performed by executing one or more tasks. Each of these tasks is handled by a plugin goal. A plugin goal may therefore be bound to zero or more build phases. If a goal is bound to one or more build phases, that goal will be called in all those phases.

The bindings for the `clean` and `site` lifecycle phases are fixed. The bindings for the default life cycle phases depending on the `packaging` value.

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#built-in-lifecycle-bindings

4.1 Executing Maven life cycle phases

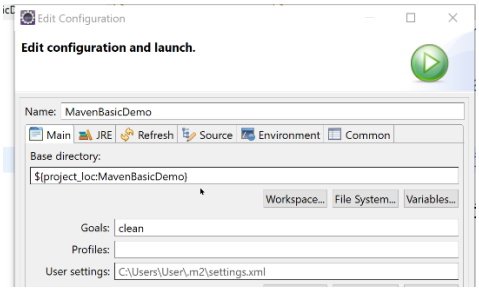
We can invoke Maven by specifying a phase in any of the build life cycles such as `compile` or `package`

Right clicking on the project and select Run As to see the variety of Maven build related options:

Select option 4 - Maven Build to bring up the Edit Configuration dialog box.

A common operation that is performed is to clean up the artifacts (classes, JARs, etc) from a previous build operation. To do that we can use the `clean` phase from the `clean` life cycle.

Type `clean` in the Goals box and click Run.



The output shows the plugin and its related goal that was called to execute this phase (`maven-clean-plugin:3.1.0:clean`) as well as the action of the phase: Deleting G:\code\ee-eclipse-11\MavenBasicDemo\target.
You can ignore any messages related to multiple SLF4J bindings: this is due to the fact that Maven itself uses SLF4J to perform logging, and since we are also using SLF4J library in our project, there are multiple implementations (termed bindings) which are available to use and Maven will automatically select an appropriate one to use.

We can also specify two or more phases to be executed: they will be executed in the order they appear.

Select option 4 - Maven Build to bring up the Edit Configuration dialog box and enter for the goal:
`clean compile`

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#built-in-lifecycle-bindings
When executing any phase in the `default` life cycle, Maven will execute all phases prior to that first before executing the specified phase. So in this case, it will execute the `process-resources` phase which is bound to the `resources` goal of the `maven-resources-plugin`. As there is currently no resources placed in `src/main/resources`, this step is skipped.

Then the compile phase is executed using the compile goal of the maven-compiler-plugin. The compiled bytecode classes are placed in `\target\classes`. If you recall, this was specified in the <outputDirectory> element of the <build> section in the effective POM.

Right click on the Project and select Refresh. Then right click on the `target` folder and select Show in -> System Explorer. In the File Explorer, verify that `\target\classes` contains the bytecode classes for the application code in the appropriate package hierarchy.

Another standard operation that we typically perform is to generate a JAR containing the compiled classes for this project.

Select option 4 - Maven Build to bring up the Edit Configuration dialog box and enter for the goal:
`clean package`

In the console output, you will now see all the phases executed up to and including `package` as well as the plugin goals that were used to execute them. Notice that the unit test code in AppTest.java in src/test/java is compiled and run. This is a basic placeholder test which is autogenerated with each Maven project and which always succeeds - in a real life project, you would create a package hierarchy with complete working unit tests for your application.

Right click on the project and select Refresh. Expand the `target` folder. Notice that it now contains a JAR file in it (`MavenBasicDemo-0.0.1-SNAPSHOT.jar`). The default name of the JAR follows the format : `artifactid-version` (as specified in the GAV coordinates for the project).

Right click on the `target` folder and select Show in -> System Explorer. Notice now that the `target` subfolder actually contains two additional folders (`classes` and `test-classes`) which contain the compiled code from `src/main/java` and `src/test/java` respectively

Copy the single JAR file out to another folder and check its contents with 7-Zip. Notice that

- The class files for the source code in the `src/main/java` directory are included here
- The class files for the source code in the `src/test/java` directory are NOT included here
- The class files for the dependencies (Logback and JUnit) are NOT include here as well

Since the dependencies are not included here, this is NOT a standalone, executable JAR that you can directly run with `java -jar`. However, you can use it as a dependency JAR for another application.

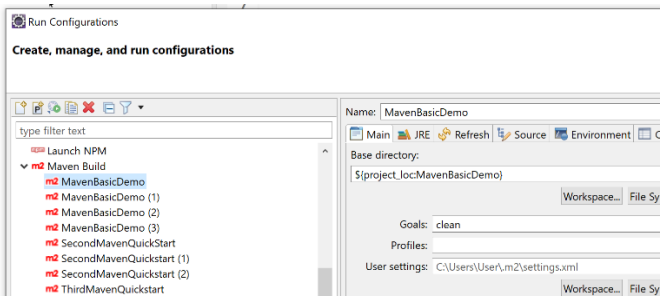
Select option 4 - Maven Build to bring up the Edit Configuration dialog box and enter for the goal:
`clean install`

Notice now that the last goal executed for the phase `install` copies the generated JAR as well as the project POM to the local Maven repository cache. Navigate to the specified folder and verify that the POM as well as the generated JAR is located in the appropriate folder hierarchy given by the `groupId` element. Since Maven will consult the local repository cache first before consulting the remote central Maven repository, we can now use this project as a dependency for a future project.

To repeat any of the commands we issued just now, select option 3 - Maven Build. This brings up a dialog box where we can select any one of the launch configurations we created earlier to run.

We can also use any one of the shortcuts available from the Maven Build menu (for e.g. 5 to clean, 6 to generate sources, 7 to install, etc)

If we choose Run Configuration from the Maven Build menu, we can access the run configurations we created earlier to edit or delete them.



We can also execute these phases using the locally installed Maven from the command line by simply preceding the phases we typed in earlier with the term `mvn`. Note that Maven executions within Eclipse is accomplished via the Maven integration (M2Eclipse) and not the locally installed Maven. We can run the equivalent commands from the command line in the event that executing them from within Eclipse produces unexpected results.

Right click on the project and select Show in Local Terminal -> Terminal. This opens a command line terminal in the root folder of the project.

Type the following command:
```
mvn clean package
```
and verify that the sequence of phase executions is exactly identical to within Eclipse.

4.2 Executing plugin goals directly

There are many plugins whose goals are not related or bound to any of the life cycle phases, and the goals of these plugins can be executed directly in the form of:

```
mvn plugin_identifier:goal_identifier
```

They provide useful functionality to supplement the primary build process.

For e.g., the maven-help-plugin is used to get relative information about a project or the system:
https://maven.apache.org/plugins/maven-help-plugin/
Try the following commands in a terminal in the root folder of the project:

```
mvn help:effective-pom

mvn help:system

mvn help:describe -DgroupId=org.apache.maven.plugins -
DartifactId=maven-compiler-plugin -Dversion=3.8.1
```

The maven-dependency-plugin is used to work with dependencies specified in the project POM:
https://maven.apache.org/plugins/maven-dependency-plugin/

Try the following commands in a terminal in the root folder of the project:

```
mvn dependency:tree

mvn dependency:analyze
```

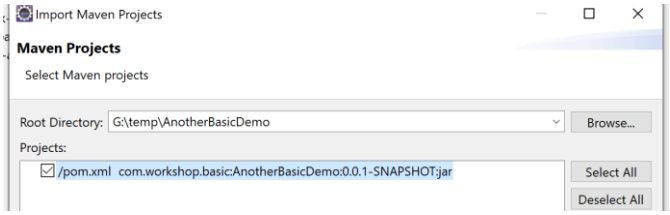
The maven-archetype-plugin is used to generate Maven archetype project templates:
https://maven.apache.org/archetype/maven-archetype-plugin/

We will use this to generate a new Maven project which we will then import into Eclipse.
In a command terminal in a completely different folder (outside of the Eclipse workspace), type this command (ensure that everything goes on a single line before hitting Enter):

```
mvn archetype:generate
-DgroupId=com.workshop.basic
-DartifactId=AnotherBasicDemo
-Dversion=0.0.1-SNAPSHOT
-DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-Dpackage=jar
-DinteractiveMode=false
```

This should generate a Maven project in the project root folder named `AnotherBasicDemo`. Verify that the folder contents are that of a standard Maven project.

Back in Eclipse, select File -> Import. In the Import Dialog box, select Maven -> Existing Maven Projects. Click Next. Select Browse and navigate into `AnotherBasicDemo`. This will automatically select the single project POM in this folder



Click Finish. The project should now appear in the Package Explorer indicating that it has been imported successfully. Note that the project folder is not automatically copied over into the current Eclipse workspace: it remains at its original location.

4.3 Using an existing Maven project as a dependency in another project

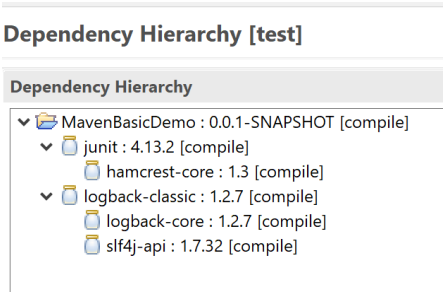
Earlier we had executed the `clean install` phases for the project MavenBasicDemo, which installed the generated JAR and POM for this project into the local Maven repository cache. We can now proceed to use this project as a dependency for the newly imported project.

In the POM of `AnotherBasicDemo`, make the following changes, save and do a Maven -> Update Project.

```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.workshop.basic</groupId>
      <artifactId>MavenBasicDemo</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
  </dependencies>
```

Notice now that all the dependency JARs for MavenBasicDemo are visible in the Maven Dependencies entry in the Project Explorer. They are all now effectively transitive dependencies of the direct dependency of MavenBasicDemo.



Modify the pre-generated App.java in `com.workshop.basic` of `AnotherBasicDemo` to:

```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        BasicLoggingDemo.main(new String[0]);
    }
}
```

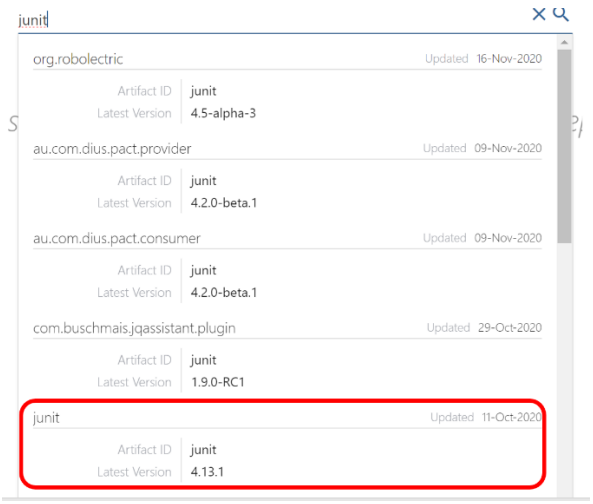
Run it in the usual way and verify that the correct log output statements appear in the console.

5 Searching for Maven dependency coordinates online

Go to the Maven Central Repository Search site : **search.maven.org**

Try to see whether you can hunt down the GAV coordinates for the two dependencies that you have included in your project: JUnit and Logback

Type in the `artifactid` for the dependency that we are looking for: `JUnit` in the search box. Notice that there are many `groupIDs` (representing different organizations or different depts in the same organization) for the same `artifactID`. The actual artifact that we are looking for is circled in red. Click on the Latest Version link to go there.



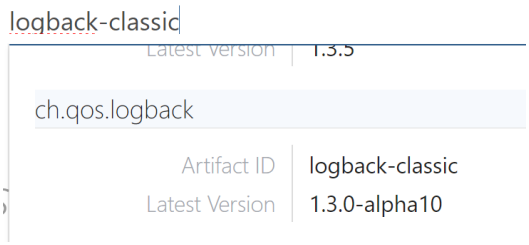
From this area, you can get the GAV coordinates for this artifact to paste in your project POM, download the JARs and also view the project POM. Be careful to check the existence of the `scope` element (if any) when copying and pasting the GAV coordinates into your POM. Many projects such as JUnit are typically intended to be used only with test code, not application code: so if you actually want to use classes from those libraries directly in your application code, make sure you change the `scope` value appropriately (or remove it entirely).

If you look through the project POM for JUnit, you will see that it the transitive dependency Hamcrest is listed there as well. This is how Maven is able to figure out that it needs to also fetch this dependency when it downloads the JUnit dependency JARs from the Maven central repository.

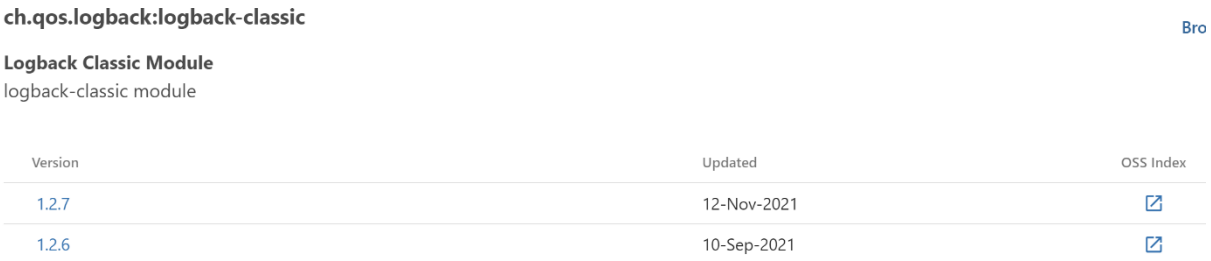
```
<dependencies>
    <dependency>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-core</artifactId>
        <version>${hamcrestVersion}</version>
    </dependency>

    <dependency>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-library</artifactId>
        <version>${hamcrestVersion}</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Repeat this for `logback-classic`.



Click on the `artifactId` itself to get a list of all versions.

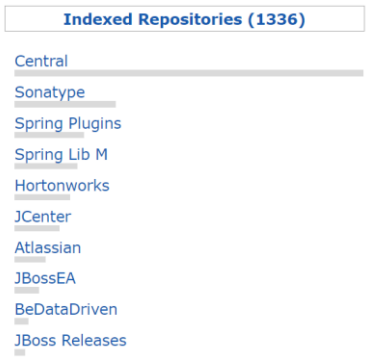


And then click on the specific version that we are interested in (1.2.7)

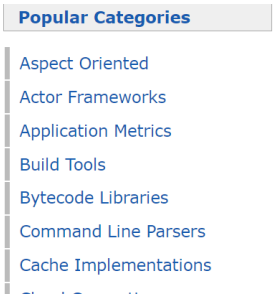
Notice that the project POM lists many dependencies, but only 2 of them have `compile` scope (`logback-core` and `slf4j-api`) so only these 2 are downloaded by Maven as transitive dependencies for `logback-classic`.

In addition to the Maven Central Repository, most organizations also host publicly accessible Maven repositories which may contain dependencies that are not available in the Maven Central repository. The site below indexes all these additional repositories
https://mvnrepository.com

You can see a short clip of the top indexed repositories on the right-hand of the main page. You can see The Maven Central Repository is right there at the top. Notice that the Spring project itself has two public repositories (Spring Plugins and Spring Lib M) which have an extremely high number of indexed JARs. This goes to show how active the Spring framework community is !



The Popular Categories list on the left hand side allows you to quickly determine the most popular projects for a particular type of usage category. This can be very helpful in determining which particular Java framework/library that you want to use in the event there are several candidates possible.



You can also do a search here for the artifact that you are looking for here as well, and the search results are generally more easy to understand than at **search.maven.org**

For e.g. typing `junit` in the search box returns a list of projects listed by their group and artifact IDs, from which you can drill down further by clicking on either one of these IDs:

Clicking on a specific artifact ID gives you the list of artifact versions at different repos, along with their usage statistics. Going to specific version page allows you obtain the corresponding GAV coordinates and download the JAR files.

Another way to determine the Maven GAV coordinates for a dependency that you need to use in your project is to search through the home website dedicated to that project (most major Java libraries and frameworks have a dedicated home website)

For e.g. if we want to use Hibernate (a popular JPA ORM provider), we can find its Maven dependency here
https://hibernate.org/orm/documentation/getting-started/

Logback's Maven dependency is listed here
http://logback.qos.ch/setup.html

If you can't find this on the project home website, you can try doing a search on:

```
xxxx maven dependencies
```

in google search, where *xxx* is the name of the artifact/project you are looking for. Usually the first 5 – 10 links will either navigate you to https://mvnrepository.com/, from which you can then explore further. Or it might redirect you to another website which lists the dependencies that you are looking for.

For e.g. if I type:

```
spring maven dependencies
```

I get links to the following sites that can help me locate the Maven dependency coordinates for the specific artifact I am interested in:

https://mvnrepository.com/artifact/org.springframework/spring-core

Alternatively, this article, gives you information on the dependencies for all the major components of the Spring framework
https://www.baeldung.com/spring-with-maven

6 Online and local repositories

The central Maven repository where the actual dependency JARs for the various project artifacts that you need to download and use in your project at is located at these links:

(newer): https://repo.maven.apache.org/maven2
(older): https://repo1.maven.org/maven2/

Go here and navigate down through any one of the project links to locate the various JAR files and other project artifacts (source code, documentation, project POM, etc)

Go to the default location for the local Maven repository cache on your machine:

Windows: `C:\Users\<User_Name>\.m2\repository`
Linux: `/home/<User_Name>/.m2/repository`
Mac: `/Users/<user_name>/.m2/repository`

Check that you have a package structure corresponding to the GAV coordinates of the dependencies used in MavenBasicDemo, for e.g:

```
C:\Users\<User_Name>\.m2\repository\junit\junit\4.13.2
C:\Users\<User_Name>\.m2\repository\ch\qos\logback\logback-classic\
1.2.7
```

Notice that the folder contains the JAR, source code as well as project POM. If you ever run a build that references these dependencies in any other Maven project in the future, Maven will first check in the local repo cache and use these JARs first. It will only check the central Maven repository if it can't find the required dependency here.

Try and delete any of these folders and then do a Maven -> Update Project on MavenBasicDemo. You will see Maven download all of the relevant dependency artifacts and create the folder anew to place them within it.