Enterprise Java with Spring
Intro to Maven
Lab 2

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.

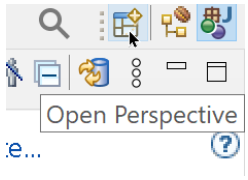
2 Creating a Maven project using an archetype

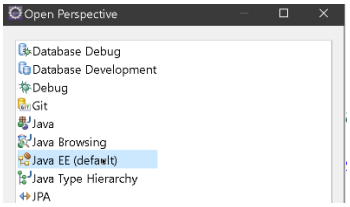
Maven provides a variety of basic archetypes to generate a template for a project

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.

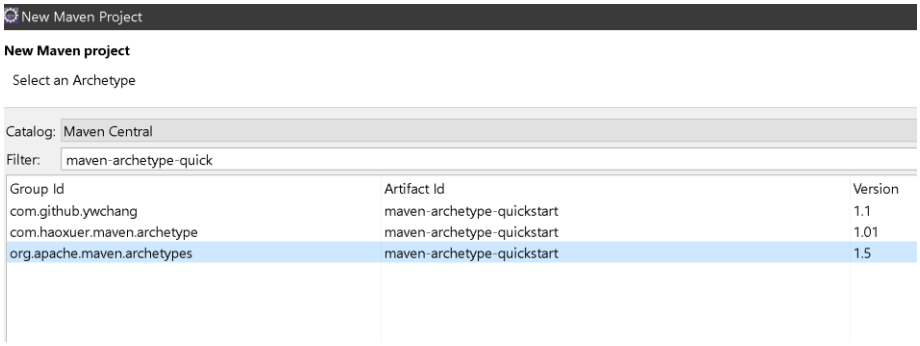
Switch to Java EE perspective, either using the icons at the upper right hand corner or via the menu: Window -> Perspective -> Open Perspective





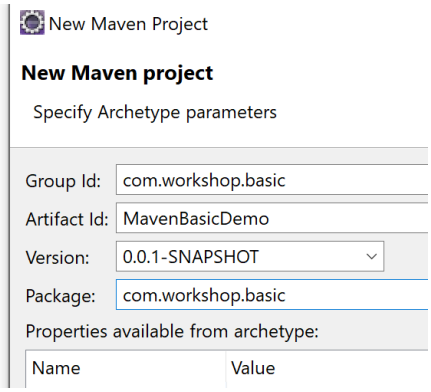
Start with File -> New -> Maven Project.
Select your current workspace location (if it is not already pre-selected by default), and then make sure the checkbox for the Use Default Workspace location is ticked.

Select Next and choose the Maven Central.
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 in the New Maven project dialog box (you can accept all the other values) and click Finish.

Group Id: `com.workshop.basic`
Artifact Id: `MavenBasicDemo`
Version: *`accept default (0.01-SNAPSHOT)`*
Package: `com.workshop.basic`



If this is the first time you are generating a Maven project, the creation of this project will download the required artifacts from Maven Central Repository (https://repo.maven.apache.org/maven2/), and messages to that effect will appear in the Console view in the stack of views at the bottom.

The Maven project creation will by default run interactively, so at some point, you will be asked to confirm the properties configuration in the Console view:

```
Confirm properties configuration:
javaCompilerVersion: 17
junitVersion: 5.11.0
groupId: com.workshop.basic
artifactId: MavenBasicDemo
version: 0.0.1-SNAPSHOT
package: com.workshop.basic
Y
```

Click in the Console view next to the Y and press enter. This should complete the installation process. You should see messages to this effect:

```
[INFO] [1m------------------------------------------------------------------------[m
[INFO] [1;32mBUILD SUCCESS[m
[INFO] [1m------------------------------------------------------------------------[m
[INFO] Total time:  20.588 s
[INFO] Finished at: 2025-05-28T07:13:51+08:00
[INFO] [1m------------------------------------------------------------------------
```

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.

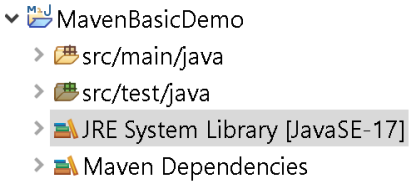
```
<groupId>com.workshop.basic</groupId>
<artifactId>MavenBasicDemo</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

2.1 Changing Java version

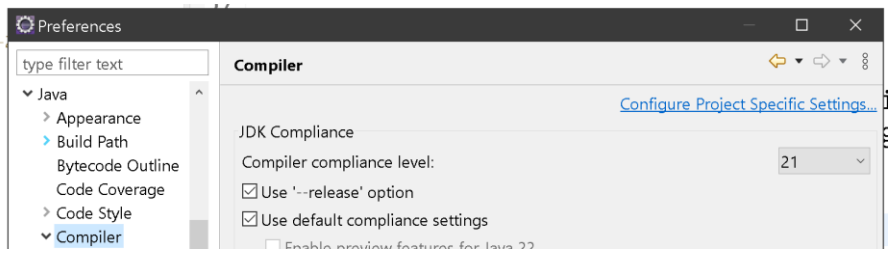
The autogenerated POMs for Maven projects created from archetypes typically reference older versions of Java. For e.g. you will see a snippet here similar to this:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.release>17</maven.compiler.release>
</properties>
```

And the Maven project will correspondingly indicate that any source code files that are within it will be compiled again this particular release of Java.



The Java version that the Maven project will compile against may be different from the setting that you might already have on your Eclipse IDE earlier (Windows -> Preferences -> Java -> Compiler), any source code files within the Maven project will be compiled against the Maven Java version setting and not the Eclipse IDE setting.

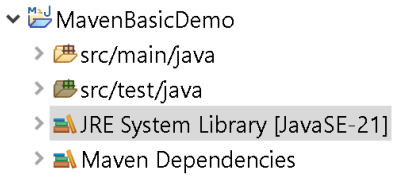


The first thing we will typically do in a newly generated Maven project is to change the Java version to the correct one that we intend to use for our application.
For e.g. if you Java 21 installed locally and you would like the code compiled against this version instead, then change the snippet to:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.release>21</maven.compiler.release>
</properties>
```

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-21.



An alternative way that is also frequently used to change the Java version (which is typical for Java 11 and above) is to configure the `maven-compiler-plugin`

In the <build> section, locate this plugin:

```
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.13.0</version>
        </plugin>
```

and replace it with this detailed configuration

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.13.0</version>
    <configuration>
        <release>21</release>
    </configuration>
</plugin>
```

You can then remove the `<maven.compiler.release>` element from the <properties> section.

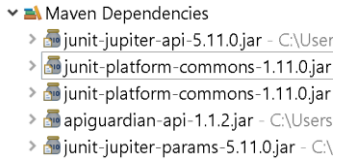
```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
```

To properly indent the XML elements in the POM after making changes, you can select the entire file (Ctrl-A), right click to bring up context menu, Source -> Format.

Save the POM and 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 project dependencies in the <dependencies> section

In the `<dependencies>` section of the POM, you should already see some default dependencies provided as part of the autogenerated project. These dependencies are the classes required for the JUnit framework and some of them are imported in `AppTest.java` in `src/test/java` of the current Maven project.
You should also be able to see the JAR files for these dependencies listed in the Maven dependencies section of the project.



These dependencies are the JAR files containing the minimum core classes required to create and run a basic JUnit 5 test. Notice that they are a subset of the JAR files in the JUnit 5 library for the SimpleLoggingProject we created earlier.

We are now going to add in the JARs for the Logback library that we used as dependencies in a previous project (SimpleLoggingProject). Add at the end of the existing `<dependencies>` section (below the dependency for `junit-jupiter-params` with the scope of `test`) the following dependency snippet for the Logback library JARs:

```
<dependencies>

  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.5.16</version>
  </dependency>

</dependencies>
```

To properly indent the XML elements in the POM after making changes, you can select the entire file (Ctrl-A), right click to bring up context menu, Source -> Format.
Save the POM, and update the project by right clicking on the project name, select Maven -> Update Project, and click OK in the dialog box that appears.

Expand on the Maven dependencies entry in the project. You should be able to see the 3 JAR files that we used in `SimpleLoggingProject` in the previous lab for the Logback library, along with the JARs for the JUnit library:

The location of these JAR files are listed next to them. Notice that the JUnit related JARs are displayed with a shaded icon – which indicates their particular scope: `test` that is different from the Logback JARs. We will discuss this later.

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`

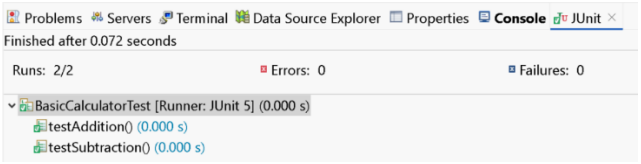
Using File Explorer, you can navigate to the respective directories listed and verify the existence of the JARs there.

Copy `BasicLoggingDemo` from the previous `SimpleLoggingProject` and paste it into the package `com.workshop.basic` in `src/main/java` . You can copy and paste by right clicking on the respective items using the context menu, DO NOT drag and drop as this will move instead of copy the file.

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`

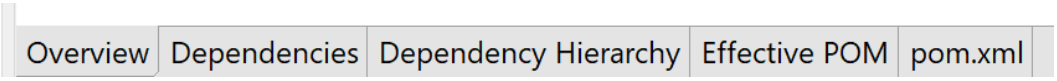
Copy `BasicCalculatorTest` from the previous `SimpleLoggingProject` and paste it into the package `com.workshop.basic` in `src/test/java`

Open `BasicCalculatorTest` in the editor and execute it as a JUnit Test (Run As -> JUnit Test). Verify that the tests complete successfully as shown in the JUnit tab in the views below, exactly the same as 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 Maven project GAV (GroupId, ArtifactId, Version) coordinates and also the project details, organization, etc, here.

3.1 Dependencies and Dependency Management

There are two main sections in a POM file that specify dependency related info, typically located after the `<properties>` top level section

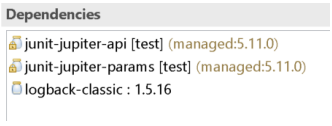
a) The top level `<dependencies>` section
b) The top level `<dependencyManagement>` section

Note that these two top level sections each contain standalone `<dependency>` elements, but these elements have different meanings and purposes from each other.

The top level `<dependencies>` section contains all the dependencies whose artifacts (JAR files) will be downloaded and included in the project build path. These JAR files can be viewed from the Maven dependencies section in the project structure.

These dependencies are direct dependencies, which means their classes are utilized directly by the source code in the project.

The right portion of the Dependencies tab shows you all the dependencies currently listed in the top level `<dependencies>` section of the POM. Notice only the artifact-id and the version number is shown.




```
        <dependencies>
                <dependency>

        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-
api</artifactId>
                        <scope>test</scope>
                </dependency>
                <!-- Optionally: parameterized tests
support -->
                <dependency>

        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-
params</artifactId>
                        <scope>test</scope>
                </dependency>

                <dependency>
                        <groupId>ch.qos.logback</groupId>
                        <artifactId>logback-
classic</artifactId>
                        <version>1.5.16</version>
                </dependency>


        </dependencies>
```


Notice that the 2 JUnit dependencies are marked as `managed`, which means that their version info and other settings are inherited from the POM's <`dependencyManagement`> section. The `test` scope means that these dependencies are only used or made available to the application during the test phase of the Maven life cycle (to be discussed later)

The top level <`dependencyManagement`> section does not contain direct dependencies. Instead, it references a Bill of Materials (BOM), which is imported as another POM file (nested into the current POM). The purpose of this BOM is to specify a list of dependencies with their versions for the purpose of standardizing these version numbers when imported into other Maven projects (such as the existing one), so that:

- We can omit version info in the dependencies section
- All modules refer to the same versions, avoiding version mismatches.

The BOM above sets the version number as 5.11.0 for the other direct dependencies in the top lelvel <dependencies> section (in this case, it will be `junit-jupiter-api`and `junit-jupiter-params`). This is why we do not explicitly declare the version number for them, unlike the case for the `logback-classic` dependency, where we explicitly declare the version as 1.5.16

The actual contents of the POM for the BOM above [is accessible here]():
You can see that this POM only has a <`dependencyManagement`> section. The <dependency> elements listed in this POM are not actual direct dependencies which will be used in a project, but are provided as a way to explicitly set the version number for the actual direct dependences that will be used in the project POM. For e.g. we have two <`dependency`> elements that explicitly sets the version number as 5.11.0 for the 2 JUnit dependencies (`junit-jupiter-api` and `junit-jupiter-params`)

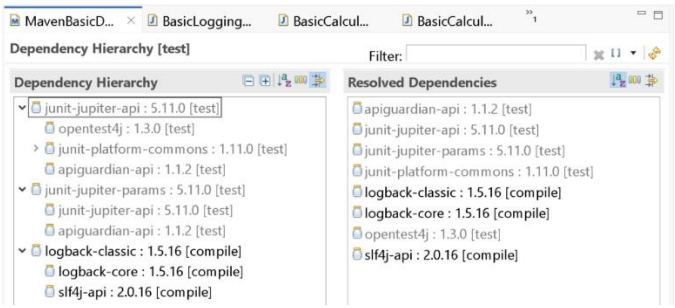
```
….
….

<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-api</artifactId>
<version>5.11.0</version>
</dependency>

….
….


<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-params</artifactId>
<version>5.11.0</version>
</dependency>
…..
…..
```

The fact that a dependency appears in a BOM does not mean that the corresponding artifact for that dependency will be automatically included into the project POM that references that BOM (such as in this case). The Maven Project POM that references this BOM must still explicitly declare the dependencies that is needed by the project source code in the top level <`dependencies`> section.

We will talk about how to obtain access the Maven dependencies

3.2 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). In this project, `logback-classic` and `junit-jupiter-params` have 2 transitive dependencies each, while `junit-jupiter-api` has 3 transitive dependencies (where one of those dependencies `junit-platform-commons` itself has another transitive dependency of its own).

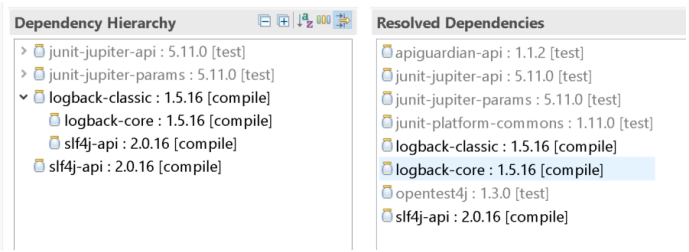


The resolved dependencies section shows the actual dependencies that will be used in the project build after dependency mediation has been performed. Dependency mediation refers to how Maven resolves version conflicts for a transitive dependency. At this point, there are no conflicts, so there is no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

```
<dependency>
	<groupId>org.slf4j</groupId>
	<artifactId>slf4j-api</artifactId>
	<version>2.0.16</version>
</dependency>
```

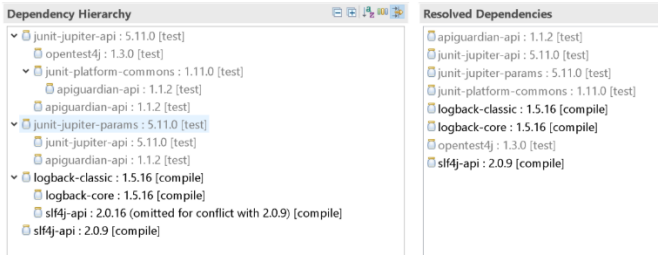
Switch back to the Dependency Hierarchy tab. You can now see that there are now two instances of `slf4j-api : 2.0.16`. One is a direct dependency (the snippet we just introduced) and the other is transitive dependency of `logback-classic`. However, there is no conflict here because the version is identical 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-api</artifactId>
	<version>2.0.9</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 (2.0.16) while the actual version to be used (2.0.9) is shown in the resolved dependencies view.



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` (1.5.16) was tested and guaranteed to work only against a specific version of `slf4j-api` (2.0.16) - 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
- whether the `logback-classic` API is using classes form slf4j-api that significantly differ between both version

Thus, when 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.

If you check the class that uses classes from `logback-classic` in our application code (BasicLoggingDemo), there is no syntax error and if you try running it as a Java application in the usual way, it runs fine without any issues.

This indicates that there is no significant difference between these 2 versions of the `slf4j-api` library `2.0.16` and `2.0.9`, with respect to how it is being used in BasicLoggingDemo.

Switch back to the pom.xml tab and change the version of the new dependency to an extremely early version and save and update the project:

```
<dependency>
	<groupId>org.slf4j</groupId>
	<artifactId>slf4j-api</artifactId>
	<version>1.2</version>
</dependency>
```

Now, if you check the class that uses classes from `logback-classic` in our application code (BasicLoggingDemo), there is now a syntax error related to the class being imported and you will no longer be able to run it as Java application.

Switch back to the `pom.xml` tab and remove this latest new dependency that we introduced so that the actual correct version of `slf4j-api` is utilized as a transitive dependency of `logback-classic`. Save and update the project.

Now the correct version of `slf4j-api` should be restored in the dependency hierarchy view `(2.0.16)`. If you check the class that uses classes from `logback-classic` in our application code (BasicLoggingDemo), the syntax error should disappear and you will be able to run it as a normal Java application as before.

The key take away here is to remember to **ALWAYS** check for transitive dependency conflicts when your project behaves in an unexpected manner. This is particular true when you integrate external, standalone Java libraries into your Spring framework projects themselves which involve many dependencies.

3.3 Test and runtime dependency scopes

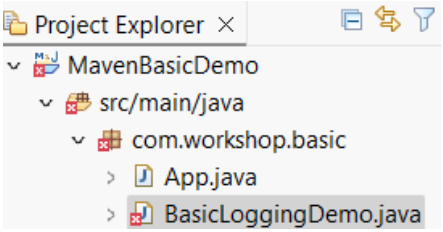
Scopes specify which classpath a dependency will ultimately be made available on (compile time, test, run time) as well as whether that dependency is transitive or not.

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 source code.

To illustrate, if you change the dependency of `logback-classic` in the POM to below and save and perform a Maven -> Update Project:

```
<dependency>
	<groupId>ch.qos.logback</groupId>
	<artifactId>logback-classic</artifactId>
	<version>1.5.16</version>
	<scope>test</scope>
</dependency>
```

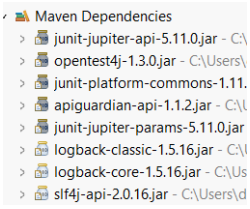
If you check in the Maven Dependencies entry, the 2 JARs associated with this dependency are now shaded, indicating that they are not accessible for compile time use. At the same time, new errors are flagged in BasicLoggingDemo related to the import of classes from this dependency, as these are no longer accessible for compile time use.



Remove the `test` cope from this dependency in the POM, to restore it back to the default dependency of `compile` and save:

```
<dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.5.16</version>
</dependency>
```

The previous compile time errors that were flagged in BasicLoggingDemo now disappear because the classes in this dependency are now available for us at compile time. The dependencies are also now shown as normal (unshaded) in the Maven dependencies list section.



Currently the two dependencies as `junit-jupiter-api` well as `junit-jupiter-params` are available under the `test` scope and therefore they are only accessible to classes in `src/test/java` (which is the default base directory for all code that is to be executed during the test phase of a Maven build).

Move the single class BasicCalculatorTest from `com.workshop.basic` in `src/test/java` to the same package name in `src/main/java` by dragging and dropping it there. Notice now there are errors flagged in this class now for the import of classes from these two dependencies as they are not available to any classes in `src/main/java`

Drag and drop the class BasicCalculatorTest back to its original location in `com.workshop.basic` in `src/test/java`. The errors should now disappear

4 Searching for Maven dependency coordinates online

4.1 Official Maven Central Repository Search

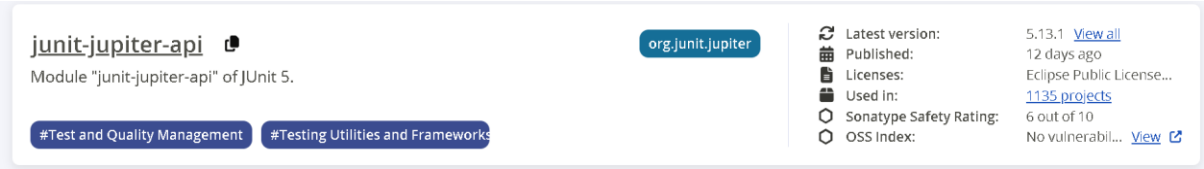
Go to the official Maven Central Repository Search site (currently hosted by Sonatype):

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

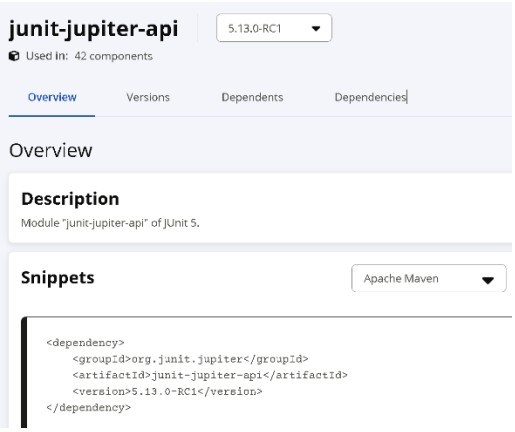
Type in the `artifactid` for the dependency that we are looking for: `JUnit` in the search box and press enter.

There are many artifacts with this particular term in their names. To determine the actual artifactID we are looking for, we can check the official documentation page for this particular Java project (JUnit5) in the section on 10.2 Dependency Metadata. Here we can see the 3 main groups: Platform, Jupiter and Vintage. The particular group we are looking for is 10.2.2 JUnit Jupiter

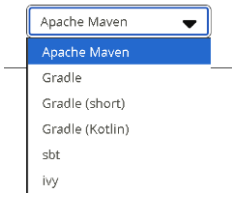
The actual artifact that we are looking for is `junit-jupiter-api` in `org.junit.jupiter`. Note that you may have to browse through several pages to find this particular artifact.



If we click on this entry summary, we will be redirected to the main page for this artifact. From this area, you can get the Maven snippet with the appropriate GAV coordinates for this artifact to paste in your project POM (using Copy to Clipboard), view all the versions available in the history of this project, and also the dependents (the projects that use this particular artifact as a dependency) and the dependencies that this artifact relies on.



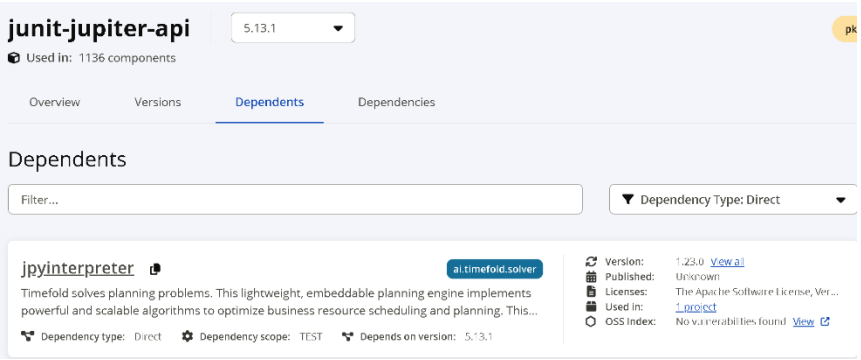
Notice that the snippets drop down list reference a few other build tools that are alternative options to Maven: Gradle is the most popular one and you may see it used in Spring Boot projects.



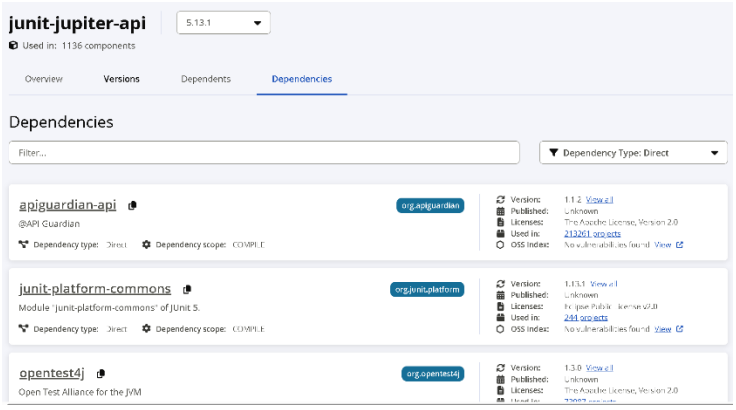
On the versions page we can view all the different versions in the versioning history of this particular project.

If you click on the Browse link, you will be redirected to the actual Maven repository site where the actual JARs for this dependency are stored and from which you can download the JAR directly if you wish (Maven already does this automatically for you in the background when you specify the dependency in your project POM).

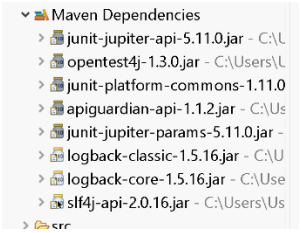
The Dependents page shows all other Maven Java projects that use this particular project (JUnit 5) dependency (`junit-jupiter-api`) in their project POM. As you can see from the list, there are many projects that utilize JUnit 5, which is the most popular unit testing library in Java.



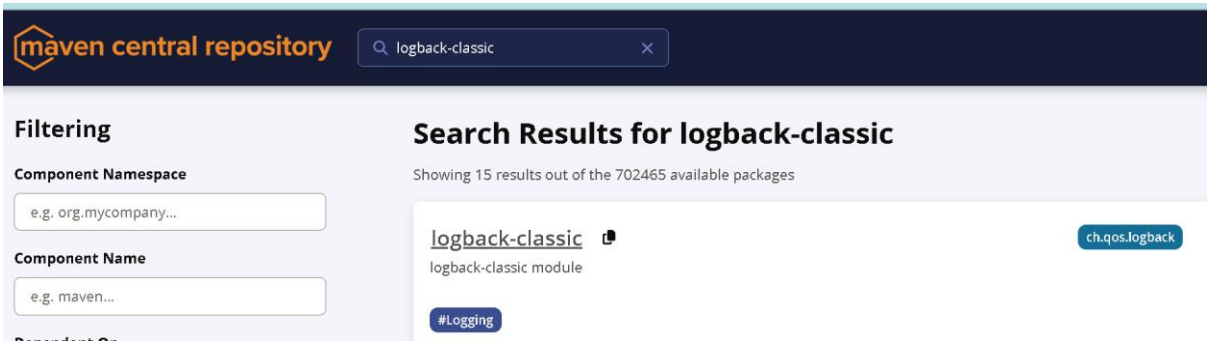
Finally the Dependencies page show the Maven projects that this particular project (JUnit 5) relies on.



As you can see, there only a few here, and the JARs for these dependencies are automatically downloaded by Maven when you specify this dependency (`junit-jupiter-api`) in the project POM, as can be seen in the Maven dependencies entry of your project in Eclipse.



Repeat this process and search for the other dependency in the project POM: `logback-classic`.

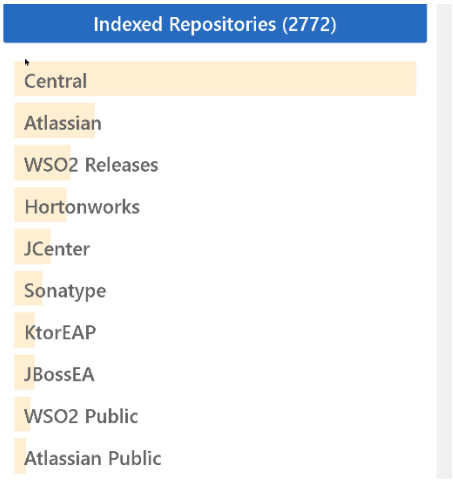


You can quickly glance through the Versions, Dependents and Dependencies pages in the same way as before.

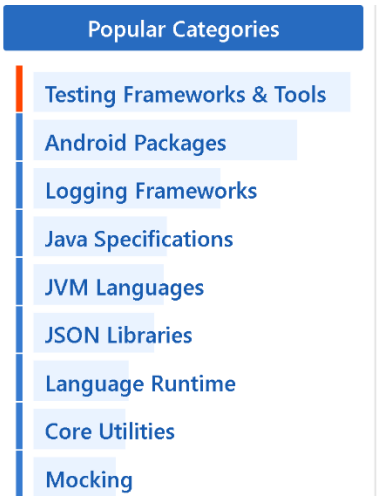
4.2 Maven Repository

In addition to the Maven Central Repository search, most organizations and companies also host publicly accessible Maven repositories which may contain dependencies that are not accessible in the Maven Central search site.
The Maven Repository site indexes all these additional repositories

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.



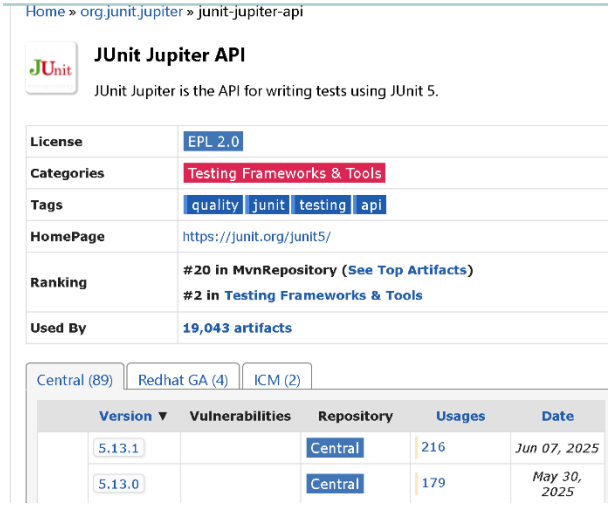
The Popular Categories list on the left hand side allows you to quickly determine the most popular projects for a particular type of usage category. This can be very helpful in determining which particular Java framework/library that you want to use in the event there are several candidates possible.



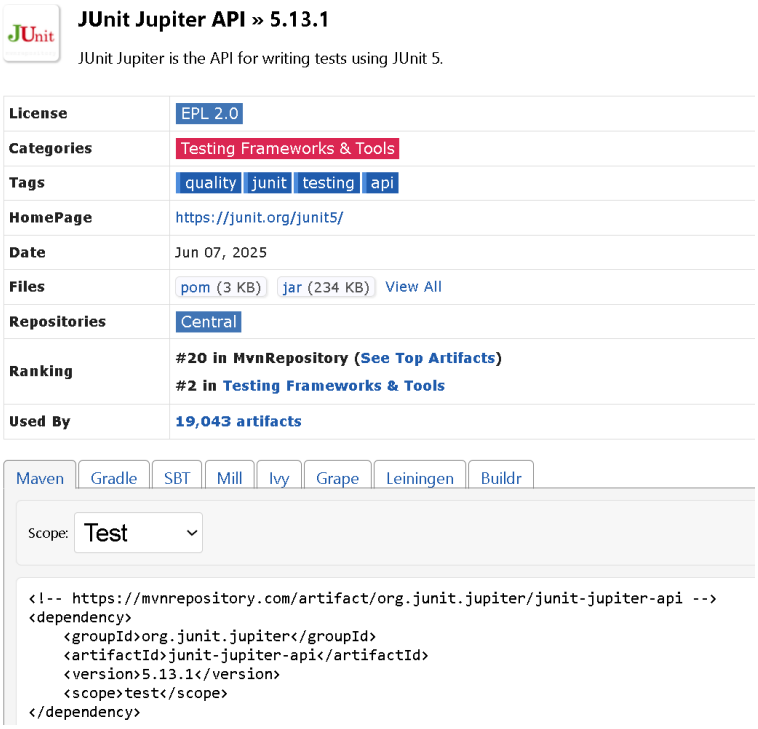
You can also do a search here for the artifact that you are looking for here as well, and the search results are generally more easy to understand than at the official Maven Central Repository Search site (so you will probably end up using this site more often to search for your relevant project dependencies)

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 entry gives you the list of artifact versions at different repos, along with their usage statistics. For e.g. clicking on `org.junit.jupiter >> junit-jupiter-api` gives you the main page for the project, from which you can double click on the specific version to zoom in further.



Going to specific version page allows you obtain the corresponding GAV coordinates as well as download the JAR file corresponding to this dependency if required:



JUnit Jupiter API » 5.13.1

JUnit Jupiter is the API for writing tests using JUnit 5.

License	EPL 2.0
Categories	Testing Frameworks & Tools
Tags	quality junit testing api
HomePage	https://junit.org/junit5/
Date	Jun 07, 2025
Files	pom (3 KB) jar (234 KB) View All
Repositories	Central
Ranking	#20 in MvnRepository (See Top Artifacts) #2 in Testing Frameworks & Tools
Used By	19,043 artifacts

Maven | Gradle | SBT | Mill | Ivy | Grape | Leiningen | Buildr

Scope: Test

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.13.1</version>
    <scope>test</scope>
</dependency>
```

Another way to determine the Maven GAV coordinates for a dependency that you need to use in your project is to search through the official 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 browse through the project website documentation to locate its Maven dependency in the getting started guide.

If you can't find this on the project home website, you can use ChatGPT (or some other AI tool) with a prompt similar to the following:

```
Provide the Maven dependency snippet to be included in a POM file
for a Java Maven project that needs to include xxxxx
```

where `xxx` is the particular Java project, for e.g. Hibernate, Spring Boot, JUnit

5 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)

For e.g. the JAR dependency for Junit 5 that we used in the earlier Maven project was actually downloaded from this location:

https://repo1.maven.org/maven2/org/junit/jupiter/junit-jupiter-api/5.13.1/

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`

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\org\junit\jupiter\junit-jupiter-api\5.11.0
```

```
C:\Users\<User_Name>\.m2\repository\ch\qos\logback\logback-classic\1.5.16
```

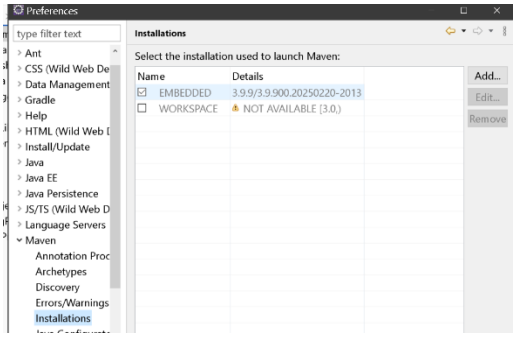
Notice that the folder contains the JAR, source code as well as project POM.

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 attempt to locate the required dependency JARs for that project POM in these directories first. It will only check the central Maven repository if it can't find the required dependency here.

Try and delete any of these 2 folders on your machine 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.

6 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. 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



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.

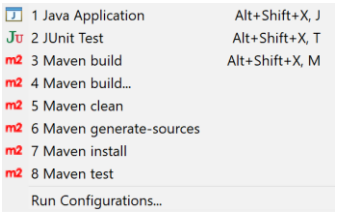
A particular build phase is performed by executing one or more tasks. Each of these tasks is handled by a plugin goal. A plugin goal may therefore be bound to zero or more build phases. If a goal is bound to one or more build phases, that goal will be called in all those phases.

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

6.1 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`

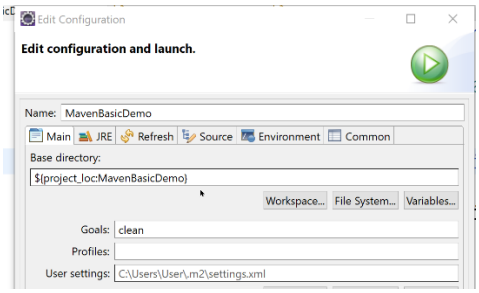
Right clicking on the project and select Run As to see the variety of Maven build related options:



Select option 3 - Maven Build to bring up the Edit Configuration dialog box. If you are doing this for the first time, a build operation will run automatically first. So you will need to right click again on the project and select this option another time.

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:3.4.0:clean`) as well as the action of the phase: `Deleting G:\code\ee-eclipse-11\MavenBasicDemo\target`. This is the folder containing the compiled bytecode classes for the project and now if you check it in the Project Explorer it will be totally empty.
We can also specify two or more phases to be executed: they will be executed in the order they appear.

Select option 3 - Maven Build to bring up the Edit Configuration dialog box and enter for the goal: `clean compile`, then click Run.

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 `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`.

Right click on the Project and select Refresh.
Then right click on the `target` folder and select Show in -> System Explorer (we need to do this because the classes folder by default will be hidden in the Project Explorer view in Eclipse). Now in File Explorer, verify that `\target\classes` contains the bytecode classes for the application code (both from `src/main/java` as well as `src/test/java`) 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 3 - Maven Build to bring up the Edit Configuration dialog box and enter for the goal: `clean package`, and click run.

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. The plugins may also additionally be downloaded from Maven Central repo first if this is the first time you are executing them.

Notice that the unit test code in AppTest (the default placeholder test for the archetype Maven project) as well as BasicCalculatorTest (which we created) in `src/test/java` is compiled and run, and the results of the tests are reported back before the JAR file is generated.
In a real life project, you would create a package hierarchy with complete working unit tests for your application in the `src/test/java` folder.

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 included** 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. That application must also additionally contain the dependencies for Logback and JUnit, since they are not included here.

To build an executable JAR that includes all the dependencies that the main application code base requires, we need to create an uber JAR or a fat JAR. We will see how to do this properly later on with Spring Boot.

Select option 3 - Maven Build to bring up the Edit Configuration dialog box and enter for the goal: `clean install`, and click run

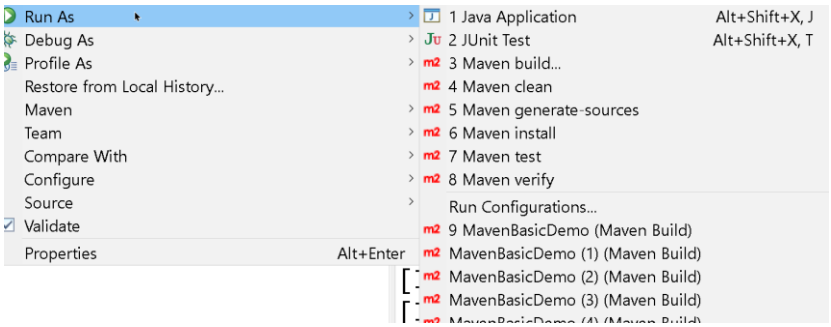
Notice now that the last goal executed for the phase `install` copies the generated JAR as well as the project POM to the local Maven repository cache.

```
[INFO] [1m--- [0;32minstall:3.1.2:install[m [1m(default-install)[m @ [36mMavenBasicDemo[0;1m ---[m
[INFO] Installing G:\code\ee-eclipse-2023-03\MavenBasicDemo\pom.xml to C:\Users\User\.m2\repository\com\workshop\basic\MavenBasicDemo\0.0.1-SNAPSHOT\MavenBasicDemo-0.0.1-SNAPSHOT.pom
[INFO] Installing G:\code\ee-eclipse-2023-03\MavenBasicDemo\target\MavenBasicDemo-0.0.1-SNAPSHOT.jar to C:\Users\User\.m2\repository\com\workshop\basic\MavenBasicDemo\0.0.1-SNAPSHOT\MavenBasicDemo-0.0.1-SNAPSHOT.jar
[INFO] [1m------------------------------------------------------------------------[m
[INFO] [1;32mBUILD SUCCESS
```

This makes the project now accessible as a dependency itself for other new Maven projects that can use it, as we will see in a future lab session.

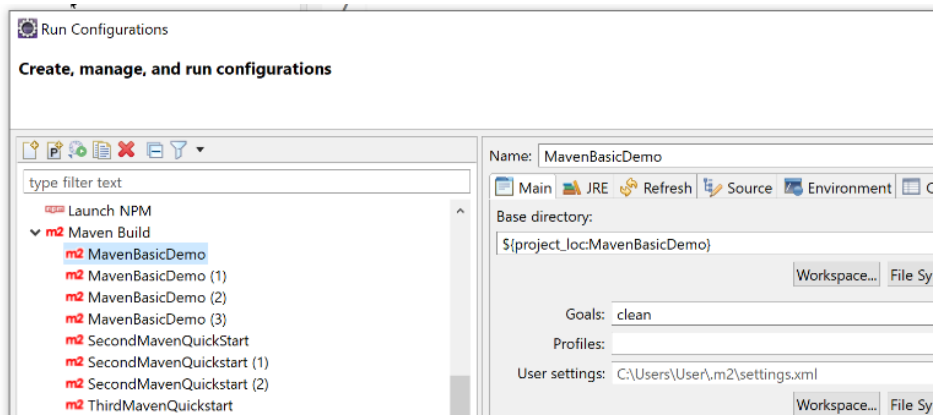
Navigate to the specified folder and verify that the POM as well as the generated JAR is located in the appropriate folder hierarchy given by the `groupId` element. Since Maven will consult the local repository cache first before consulting the remote central Maven repository, we can now use this project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

At this point of time, you will have executed a number of Maven commands within a specific configuration. From the Run as context menu available from right clicking on the project, you should be able to see the list of those configurations below the Run Configurations option and click on them to run any of them again.



We can also use any one of the shortcuts available from the Maven Build menu (for e.g. 4 to clean, 6 to install, etc)

Finally, we can select Run Configurations to access these configurations we created earlier to either edit them and run them again or delete them entirely.



We can also execute these Maven 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:
```
mvn -v
```
To check the version of the locally installed Maven and its installation directory (as given by the MAVEN_HOME environment variable)

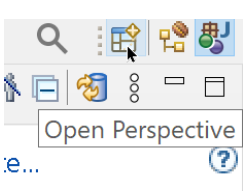
Type:
```
mvn clean package
```
and verify that the sequence of phase executions is exactly identical to within Eclipse.

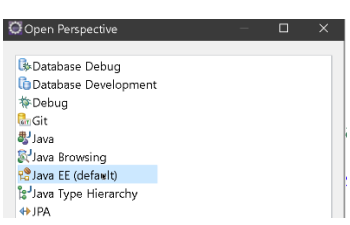
6.2 Using an existing Maven project as a dependency in another project

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 will now create another basic Maven project using the `maven-archetype-quickstart` as before.

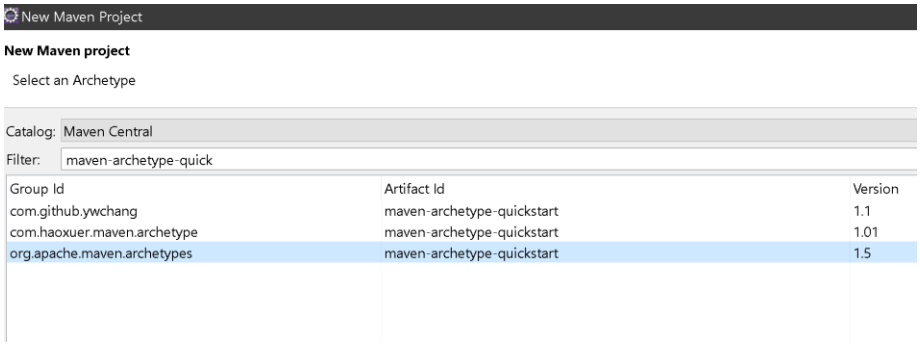
Switch to Java EE perspective, either using the icons at the upper right hand corner or via the menu: Window -> Perspective -> Open Perspective





Start with File -> New -> Maven Project.
Select your current workspace location (if it is not already pre-selected by default), and then make sure the checkbox for the Use Default Workspace location is ticked.

Select Next and choose the Maven Central.
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 in the New Maven project dialog box (you can accept all the other values) and click Finish.

Group Id: `com.workshop.another`
Artifact Id: `AnotherBasicDemo`
Version: *`accept default (0.01-SNAPSHOT)`*
Package: `com.workshop.another`

If this is the first time you are generating a Maven project, the creation of this project will download the required artifacts from Maven Central Repository (https://repo.maven.apache.org/maven2/), and messages to that effect will appear in the Console view in the stack of views at the bottom.

The Maven project creation will by default run interactively, so at some point, you will be asked to confirm the properties configuration in the Console view:

```
Confirm properties configuration:
javaCompilerVersion: 17
junitVersion: 5.11.0
groupId: com.workshop.another
artifactId: AnotherBasicDemo
version: 0.0.1-SNAPSHOT
package: com.workshop.another
Y
```

Click in the Console view next to the Y and press enter. This should complete the installation process. You should see messages to this effect:

```
[INFO] [1m------------------------------------------------------------------------[m
[INFO] [1;32mBUILD SUCCESS[m
[INFO] [1m------------------------------------------------------------------------[m
[INFO] Total time:  20.588 s
[INFO] Finished at: 2025-05-28T07:13:51+08:00
[INFO] [1m------------------------------------------------------------------------
```

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

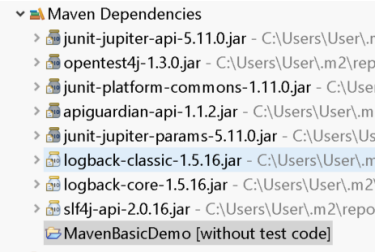
Notice that the first 3 elements (the GAV coordinates of the POM) matches the values that you entered earlier.

In the POM of `AnotherBasicDemo`, add the following dependency that references the MavenBasicDemo project dependency that we created earlier, save and do a Maven -> Update Project.

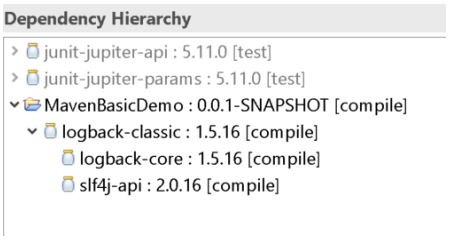
```
<dependencies>

  …. Existing org.junit.jupiter dependencies…..

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

</dependencies>
```

Maven will first search for this dependency in the local Maven repository cache, and it should be able to find it there since we placed the JAR for MavenBasicDemo in this cache in the earlier lab session when we executed the phases: `clean install`

Notice now that all the dependency JARs for MavenBasicDemo are visible in the Maven Dependencies entry in the Project Explorer. The local artifact MavenBasicDemo itself appears as an entry at the bottom.



We can also see that the Logback related JARs are shown as transitive dependencies of the direct dependency of MavenBasicDemo.



Modify the pre-generated `App.java` in `com.workshop.another` of `AnotherBasicDemo` in order to access a class from `MavenBasicDemo`

```
package com.workshop.another;

import com.workshop.basic.BasicLoggingDemo;

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.

When you are done, you can close all the source code files from this project as well as the project itself in the IDE to prevent accidentally opening the wrong files in future projects.