

SENG202 – Software Engineering Project Workshop

2020

Tutorial 1 – Project Build Automation with Maven

1. Introduction

This tutorial will walk you through setting up a project using Maven. Maven is a build automation and dependency management utility which is required for this course. It is popular in industry and understanding how to use it (and other similar tools) will be very useful.

Each student should complete this individually. Later you can nominate one (blank) project to use as your team's official SENG202 project or create a new one by running through the steps again. We will begin with some background information on Maven (mainly based on Maven's official documentation¹). **If you skip this part, do not be surprised if the remainder of this tutorial leaves you feeling confused.**

Objectives:

- Understand what Maven is and the problems it can solve;
- Make sure Maven is installed on your development machines;
- Become comfortable with setting up a project with Maven;
- Learn how to import a Maven project into an IDE;
- Understand how dependency management in Maven works;
- Use Maven to build a project from source (produce an executable);
- Complete the Maven quiz on the Quiz Server. The quiz will be open in the second half of the tutorial time, and close at the end of the tutorial.

2. Maven and the Build Lifecycle

Maven began as an attempt to support the build processes of big and complex systems, usually compound of several projects, each of them with their own build files. The aim was to have a standard way to build projects, a clear definition of what the project consisted of, an easy way to publish project information, and a way to share executable files across several projects. Based on the concept of a project object model (POM)², Maven will be used to manage your project's build, reporting and documentation.

The Build Lifecycle

The source code commonly written by software engineers is not very useful by itself. It must be compiled into executable binary instructions and linked to any dependencies (e.g. libraries) which the programmer has imported. There are also many other jobs that professional engineers need to perform when they want to

¹Apache Maven Project (<http://maven.apache.org/>)

²The POM file (<http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html#the-pom>)

convert their work into a real executable program. All these jobs (including compilation, linking, automated testing, and others) form what is known as the “build process”.

Maven is a build automation tool, which means that it carries out these tasks automatically (software engineers love writing tools to do their work for them). Maven is based around the central concept of a build lifecycle. This means that the process for building and distributing a particular artefact (or a whole project) is clearly defined. Developers only need a small set of commands to build any Maven project, and the [POM](#)³ will ensure they get the desired results.

There are three built-in build lifecycles in Maven: **default**, **clean** and **site**. The **default** lifecycle handles your project deployment, the **clean** lifecycle handles project cleaning, while the **site** lifecycle handles the creation of your project's site documentation. The **default** Maven build lifecycle consists of the following phases⁴:

- **Validate:** Ensure that all the information needed to build the project has been provided and that there are no major problems with the structure of the project itself.
- **Compile:** Convert the project source code into executable binary files.
- **Test:** Run the project's unit tests (and make sure they all pass).
- **Package:** Wrap the compiled binaries in a .jar file, for easier distribution.
- **Verify:** Run developer-defined final checks on the package.
- **Install:** Copy the package to a local location (on the same computer as the packaging took place).
- **Deploy:** Copy the package to a remote location, where it may be accessed by other developers.

This means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository. One way of customising Maven's behaviour is by specifying plugins for it to use during different phases.⁵ They are essentially just chunks of code that run at a time of the developer's choosing.

3. Dependency Management

Virtually all software projects have dependencies; usually imported libraries. Dependency management can be a huge problem for software engineers, particularly when working with large and complex projects. Manual dependency management involves:

- Downloading libraries (i.e. finding them on the internet);
- Installing libraries into the project directory;
- Periodically updating those libraries (which requires repeating the previous two steps);
- Telling co-workers which libraries to download, being sure to use the right version of a library for the project, using the right version of one library with the right version of another library, and;
- Making sure that all developers are using the same version of the same libraries for the same projects.

³ [Introduction to the POM](http://maven.apache.org/guides/introduction/introduction-to-the-pom.html) (<http://maven.apache.org/guides/introduction/introduction-to-the-pom.html>)

⁴ [Apache Maven Project](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html) (<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>)

⁵ [A Build Phase is Made Up of Plugin Goals](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#a-build-phase-is-made-up-of-plugin-goals) (<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#a-build-phase-is-made-up-of-plugin-goals>)

Hopefully you can see how things can start to go wrong. Maven lets you list your project's dependencies (and their version numbers) using a predefined format. Here's an example for the Joda Money library⁶:

```
<dependency>
  <groupId>org.joda</groupId>
  <artifactId>joda-money</artifactId>
  <version>1.0.1</version>
</dependency>
```

Snippet 1: Joda Money library dependency block

You can use Maven to “resolve” your project's dependencies by downloading any versions of the libraries specified for the project which your computer may be missing. For example, if your computer had version 0.9.1 of the Joda Money library installed but your project's Maven settings indicated that it needed version 1.0.1 then Maven would download the newer version and you would end up with two versions of the same library (the same would apply if you had version 1.0.1 installed but the project required version 0.9.1). This makes it much easier to set a project up on a new computer. Rather than having to work through a list of libraries and download them all individually, you can simply run Maven and it will do the work for you.

This creates an element of **repeatability**, which is one of Maven's greatest strengths. By formally specifying and automatically executing the build process Maven can reliably produce identical executable programs on different computers at different times and under different conditions. This has many applications. Teams of developers can share their Maven configuration files so that if one person adds code that requires a certain library the others don't have to worry about locating, downloading, and installing the correct version of that library. Maven can also be useful when companies need to supply different customers with different versions of the same product, and for repeatedly testing a product with a continuous integration system like GitLab CI (or Jenkins), as you'll see later in the course.

Most popular IDEs offer support for Maven, allowing you to do even less work. Although you do still need to understand how everything works, because one day it **will** fall apart around you.

4. Setting up a Project with Maven

Installing Maven

Make sure you have Maven installed on your development machine (or machines). It should already be installed in the university lab computers. The simplest way to do this is to type `mvn --version` into the command line. If you get a “command not found” error, please, let one of your tutors aware of this as soon as possible. For your home computer, you might need to [download](#)⁷ and [install](#)⁸ Maven. The process for doing this varies with operating system. We recommend searching for a tutorial suited to your particular environment. It's also a good idea to try and use the same version of Maven as the rest of your team, as this reduces the number of inconsistencies between your environments.

⁶ www.joda.org/joda-money

⁷ <http://maven.apache.org/download.cgi>

⁸ <http://maven.apache.org/install.html>

Creating a Maven Project

Now we'll use Maven to create a "scaffold" for your new project. This just involves automatically creating a bunch of folders and a few configuration files, as you'll see. You could do this manually, but that would take ages.

In the command line navigate to a suitable directory (this is where you'll create your project directory - you can always move it later, of course). Paste the following command⁹ into the shell and edit it as indicated. **Make sure that the command pastes into the shell correctly** (sometimes formatting errors occur and need to be manually corrected).

```
mvn archetype:generate -DgroupId=seng202.team[teamnumber] \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false \
    -DartifactId=[your_project_name]
```

Snippet 2: Maven command for create a new project

The directory structure that Maven uses may look complex at first but it's intuitive once you see it laid out. It's designed to scale for large and complex projects, but it'll also work well for your needs. Open the new directory created by Maven in a file browser and have a look around. The `src` folder is for your source code and contains sub-directories for your test and production code. The `pom.xml` file is particularly important, as that's where you configure Maven.¹⁰

Introducing the Maven POM file

The `pom.xml` file is the core of a project's configuration in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. In the previous step you executed the Maven goal `archetype:generate` with a set of parameters¹¹ to create a simple Maven project based upon the "generate" archetype¹² and the resulting `pom.xml` output can be seen in the code snippet below.

The POM file can become large and its complexity can seem daunting, but it is not necessary to understand all the intricacies in order to use it effectively. However, it is important to understand the formatting of the file and be aware of the opening/closing `xml` tags.

⁹ See <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html> for details.

¹⁰ Maven applies the instructions in each POM file to the directory it finds it in. This POM file is in the root directory of the project, so it is used for project-wide settings. You can define other POMs, but you probably won't need to for this project.

¹¹ A list of goals can be found at <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle> Reference

¹² Refer to <https://maven.apache.org/archetype/index.html> for further information about Maven Archetype.

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>seng202.team1</groupId>
  <artifactId>MyProjectName</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>MyProjectName</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Snippet 3: POM file created by the `mvn archetype:generate` command.

Importing a Maven Project into your IDE

IntelliJ

IntelliJ has built-in support for Maven (although if you turned the Maven plugin off when you were setting it up you'll need to re-enable it¹³). To import your project:

- Launch IntelliJ and choose `Import Project` from the start up window.
- Select the project you created with Maven in the previous section of this tutorial.
- Select `Import project from external model` and make sure that **Maven** is highlighted.
- Navigate through your project folder and open the POM file. You should then see something similar to Figure 1.

Eclipse

Some versions of Eclipse have support for Maven built in. If you find that yours does not you should search for an install a suitable Maven plugin for Eclipse. To import your project:

- Launch Eclipse and choose `File > Import`.
- Select `Maven > Existing Maven Projects` from the list. Then press `Next`.
- Browse for the project you created in Maven in the previous section of this tutorial and click `Open`.
- Ensure that your project is selected in the `Projects` pane and then click `Finish`.
- Navigate through your project folder and open your POM file. You should then see something similar to Figure 2.

¹³ Go to File > Settings > Plugins -> Maven and click Enable.

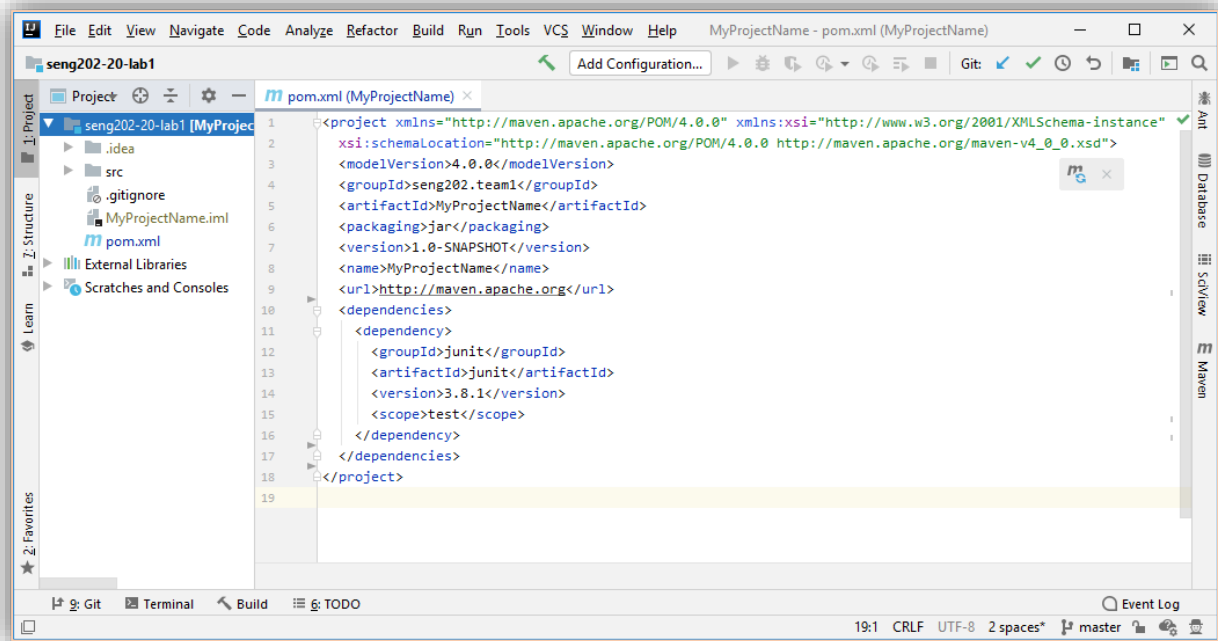


Figure 1: Maven Project Imported into IntelliJ IDE

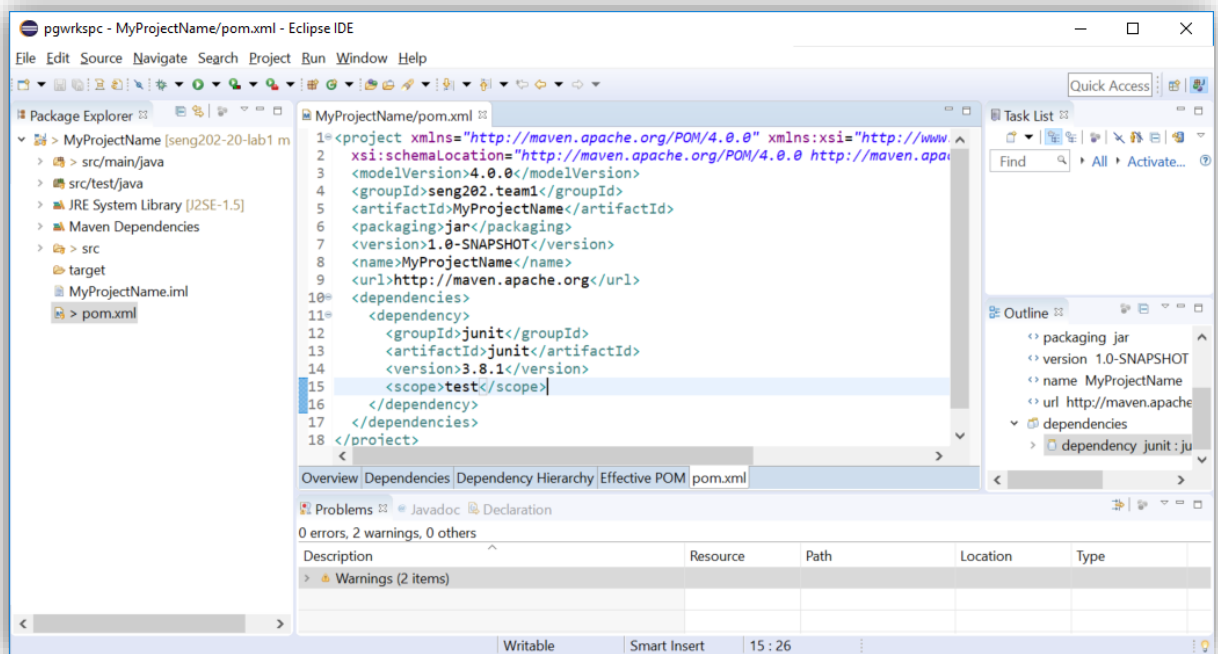


Figure 2: Maven Project Imported into Eclipse IDE

Setting up Java version compatibility

Add the below snippet (Snippet 4) to your POM file¹⁴ to setup the Java version which will be used in your project. In SENG202 all projects must use Java version 11. You can check the Java version on your device by typing `java --version` into a terminal. Make sure to have the same version on all devices used to work on your SENG202 project.

```
<!-- Java version setup -->
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
</properties>
```

Snippet 4: Project's source and target Java version setup

5. Dependency management with Maven

Verifying the Project Scaffold

Now that you've imported your new Maven project into your IDE you should make sure you can run it.

IntelliJ

Locate `App` within the Project Window (`src/main/java/seng202/team[your_team_number]/App`), right click on the file name and select: **Run/Run 'App.main()'**.

Eclipse

Locate `App.java` within the Package Explorer, right click on this file and select: **Run As -> Java Application**.

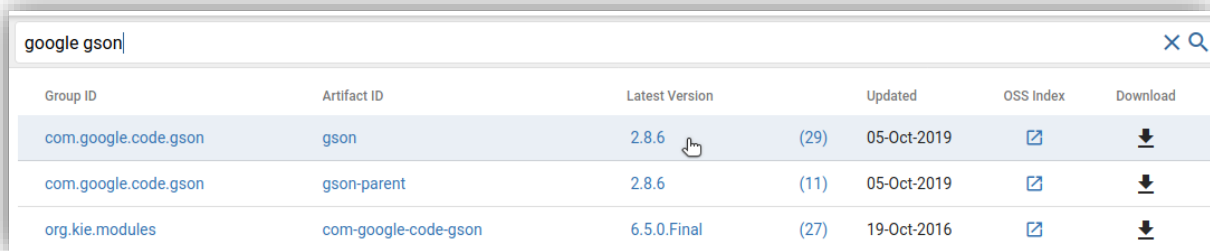
The default `App.java` file Maven created when it scaffolded the project will print "Hello World!", so make sure you see that output before proceeding.

Searching for a Dependency

Now imagine that we want to use the popular Gson library (this is used to convert between Java and JSON objects). Rather than manually downloading the Gson .jar file and copying it into our project directory we'll add it to Maven as a dependency. **Maven Central** (<http://search.maven.org>) is a repository containing thousands of libraries built with Maven. This makes it extremely easy to add them to your project.

Search Maven Central for "**google gson**" and select (*click on the version number to select it*) the most recent version, which up to the date of this tutorial release, was the version 2.8.6 as shown on Figure 3.

¹⁴ Before the dependencies block.



Group ID	Artifact ID	Latest Version	Updated	OSS Index	Download
com.google.code.gson	gson	2.8.6	(29) 05-Oct-2019	[OSS Index]	Download
com.google.code.gson	gson-parent	2.8.6	(11) 05-Oct-2019	[OSS Index]	Download
org.kie.modules	com-google-code-gson	6.5.0.Final	(27) 19-Oct-2016	[OSS Index]	Download

Figure 3: Maven Central: `google gson` search outcome

Adding a Dependency

On the page that opens you should see a snippet enclosed in `<dependency>` tags (Figure 4). This is what you need to include into your project's POM file in order to add the Gson library version 2.8.6 as a dependency (which will in turn tell Maven to download it whenever necessary) of your project.



Gson

com.google.code.gson:gson
2.8.6

```
<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.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.6</version>
</project>
```

Apache Maven

maven.apache.org

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.6</version>
</dependency>
```

Figure 4: Maven Central: `google gson` dependency block

Copy/paste this from the website into your project's **pom.xml** file (it should go inside the `<dependencies></dependencies>` tags, as in the screenshot below).

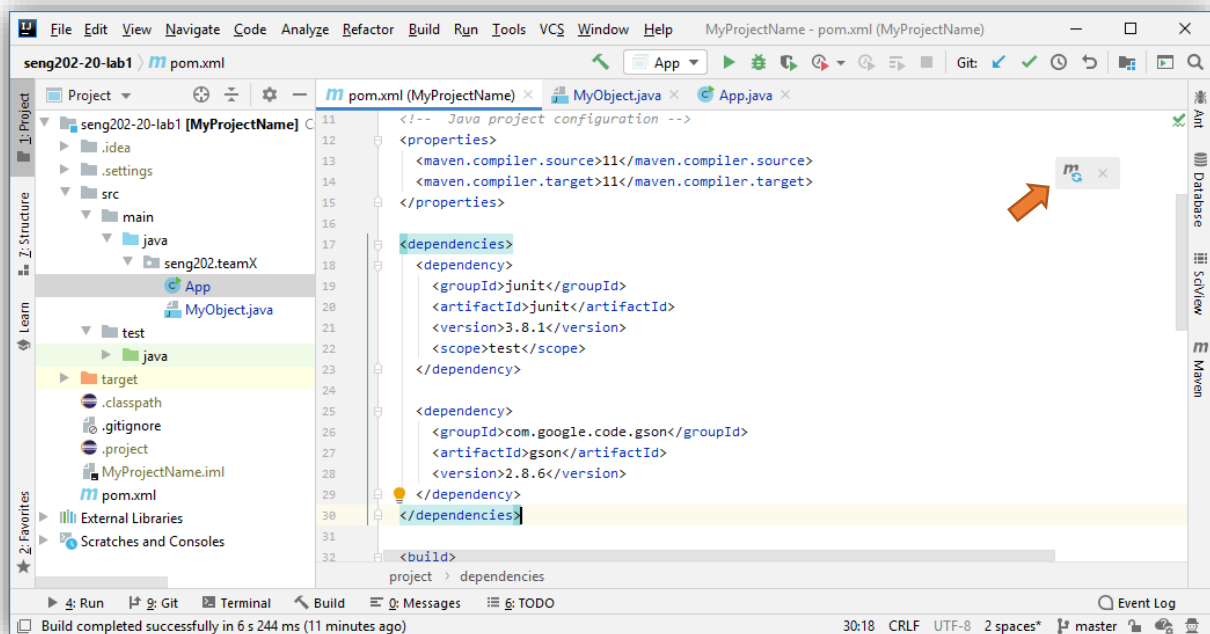


Figure 5: POM file with the `google gson` dependency block

Your IDE should now detect that you've made changes to the **POM** file and update the project accordingly (in our case, that just means resolving the newly added dependency)¹⁵. After this you should be able to use **Gson** in your project.

Note that Eclipse will by default display POM files in a custom interface. We recommend interacting directly with the raw XML¹⁶ so click the "pom.xml" tab to switch views.

Verifying the New Dependency

In order to verify if the added dependency is working fine, let's create a dummy class named **MyObject** (Figure 6) which we will use to test the Gson library and refactor the **App** class (Figure 7) class in order to perform the test. Run the program and check if you got the same output as shown above.

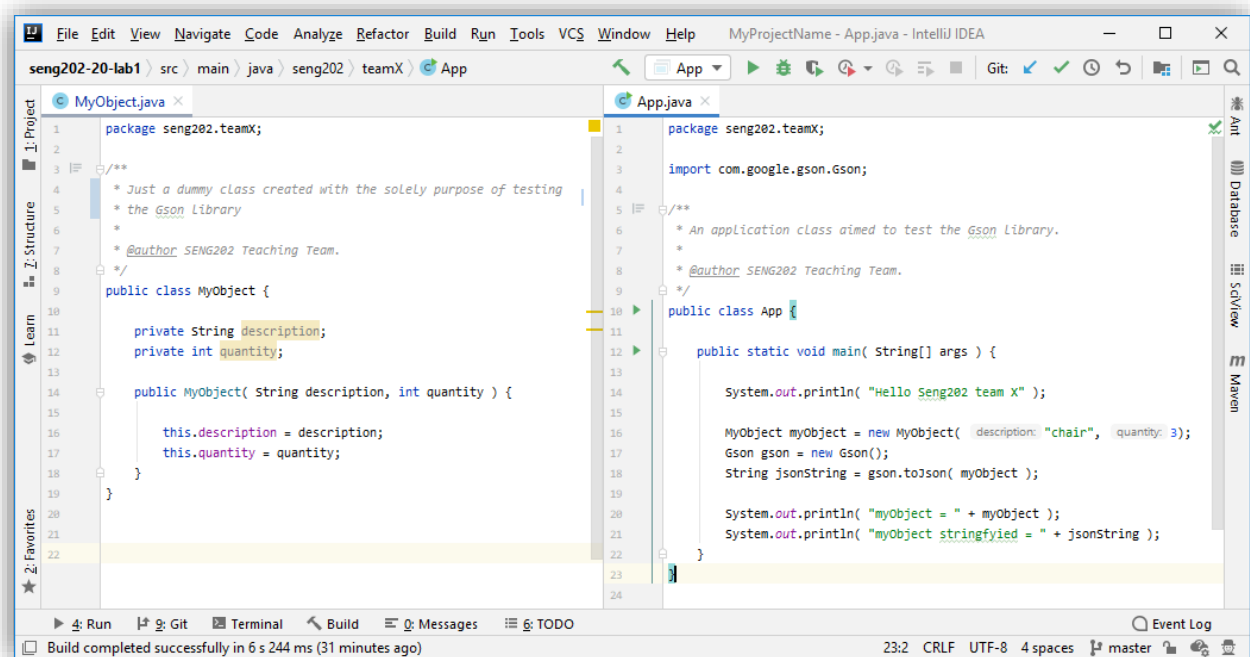


Figure 6: MyObject and App classes

Make sure everything was worked fine before moving to the next sections.

Version Management

If Gson (or another of your project's dependencies) was updated and you wanted to use a feature from the new version, all you would need to do is to edit the version number for that dependency in your project's POM file. Your IDE would notice the change and ask Maven to resolve the new version requirement, i.e. to reload the dependencies.

¹⁵ In case the reload process does not automatically start, you can manually invoke the process clicking on the icon highlighted in Figure 5.

¹⁶ It's good to see what things really look like without relying on an interface, and a diet of raw XML builds character.

Sharing Maven Configurations

In a team project, your POM file would be added to your Git repository (or equivalent). Any changes made to the POM by one member of the team would thus be shared with all other members. If this doesn't mean much to you don't worry. We will introduce Git in another tutorial.

Manually Adding Dependencies

It is possible to add dependencies which are not available on Maven Central, but we do not encourage doing so for SENG202. Manually adding jar files to your project is usually more trouble than it's worth. If you really want to do this then please talk to one of your tutors so that you can together find the best possible solution for the case.

6. Creating an Executable Jar

In the previous section (Dependency Management with Maven) we looked at how useful Maven is for handling dependency management. That's great for collaborating with other engineers, but what about when we deploy our product for customers to use? We don't want them to have to install Maven, much less have all sorts of third party jars downloaded to their system. It's better to give customers (and markers) a single runnable file (usually .jar) which contains all our source code and dependencies. Fortunately, there's a Maven plugin which will do this for us!

Paste the following block into your POM file (put it within the <project></project> tags) and edit it to include your team number where indicated. This tells Maven to use the **Shade**¹⁷ plugin to produce an '**UberJar**' or '**FatJar**' file¹⁸ during the package phase.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.2.4</version>
      <configuration>
        <transformers>
          <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
            <manifestEntries>
              <Main-Class>seng202.teamX.App</Main-Class>
              <X-Compile-Source-JDK>11</X-Compile-Source-JDK>
              <X-Compile-Target-JDK>11</X-Compile-Target-JDK>
            </manifestEntries>
          </transformer>
        </transformers>
      </configuration>
      <executions>
        <execution>
          <phase>package</phase>
```

¹⁷ <https://maven.apache.org/plugins/maven-shade-plugin/>

¹⁸ An UberJar/FatJar is an executable file containing your code plus the all the dependencies.

```

        <goals>
          <goal>shade</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>

```

Snippet 5: Shade plugin added to the POM file.

Packaging the Project and its Dependencies

In the command line navigate to your project directory and type `mvn package`. The build process should start and successfully build your project as shown on Figure 7. As a result, Maven should have created an executable `jar` at `[your_project_name]/target/[your_project_name]-1.0-SNAPSHOT.jar` (Figure 8). Run it by navigating to its folder in the command line and entering the command `java -jar [your_project_name]-1.0-SNAPSHOT.jar`. You should see the text “Hello World!” in the shell (Figure 9).

```

Terminal: Local x +
/home/patricia/prgm_wokspc/uocnt/seng202/2020-lab1/MyProjectName/target/MyProjectName-1.0-SNAPSHOT-shaded.jar
[INFO] Dependency-reduced POM written at: /home/patricia/prgm_wokspc/uocnt/seng202/2020-lab1/MyProjectName/dependency-reduce
d-pom.xml
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.121 s
[INFO] Finished at: 2020-07-10T19:39:47+12:00
[INFO] -----

```

Figure 7: `mvn package` outcome, build success

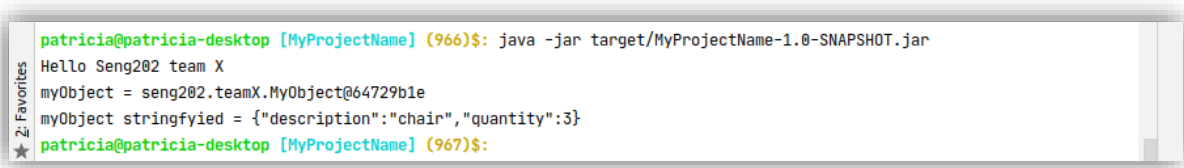
```

patricia@patricia-desktop [MyProjectName] (965)$: ls -al target/
total 644
drwxrwxr-x 9 patricia patricia 4096 Jul 10 19:39 .
drwxrwxr-x 6 patricia patricia 4096 Jul 10 19:39 ..
drwxrwxr-x 3 patricia patricia 4096 Jul 10 04:19 classes
drwxrwxr-x 3 patricia patricia 4096 Jul 10 04:18 generated-sources
drwxrwxr-x 3 patricia patricia 4096 Jul 10 04:19 generated-test-sources
drwxrwxr-x 2 patricia patricia 4096 Jul 10 19:38 maven-archiver
drwxrwxr-x 3 patricia patricia 4096 Jul 10 19:38 maven-status
-rw-rw-r-- 1 patricia patricia 308619 Jul 10 19:39 MyProjectName-1.0-SNAPSHOT.jar
-rw-rw-r-- 1 patricia patricia 308619 Jul 10 19:38 original-MyProjectName-1.0-SNAPSHOT.jar
drwxrwxr-x 2 patricia patricia 4096 Jul 10 19:38 surefire-reports
drwxrwxr-x 3 patricia patricia 4096 Jul 10 04:19 test-classes

```

Figure 8: `mvn package` outcome, generated files

Note that two `.jar` files were created, one with the third parties dependencies (`MyProjectName-1.0-SNAPSHOT.jar`) and one without those dependencies (`original-MyProjectName-1.0-SNAPSHOT.jar`). For your project submissions, you should always use the first one.



```
patricia@patricia-desktop [MyProjectName] (966)$: java -jar target/MyProjectName-1.0-SNAPSHOT.jar
Hello Seng202 team X
myObject = seng202.teamX.MyObject@64729b1e
myObject stringfyied = {"description":"chair","quantity":3}
patricia@patricia-desktop [MyProjectName] (967)$:
```

Figure 9: Runnable file result

Summarizing what has just happened, the ``mvn package`` command told Maven to complete the “package” phase of the build lifecycle. To do this Maven must also execute any phases which precede it in the build lifecycle (see the list at the start of this tutorial). You can also invoke commands like ``mvn deploy`` or ``mvn test`` to trigger different phases (other useful Maven commands are described in the next section).

If someone asks you to build a project with Maven this is probably what they mean. They want you to use Maven to produce an executable package with ``mvn package`` or a similar command.

Note that IntelliJ has a built-in interface for interacting with Maven (View > Tool Windows > Maven Projects). You can use this if you would prefer to avoid the command line.

7. Useful Maven Commands

- ``mvn package``
Take the compiled code and package it in its distributable format, such as a JAR. *If someone asks you to build a project with Maven this is probably what they mean.*
- ``mvn install``
Install the package into the local repository, for use as a dependency in other projects locally.
- ``mvn deploy``
Copies the final package to the remote repository for sharing with other developers and projects.
- ``mvn test``
Executes the unit tests that are defined in project’s directory.
- ``mvn clean``
Removes files generated at build-time in a project’s directory.

It is possible to also combine commands such as ``mvn clean install`` which will execute the clean command followed by the install command.