The Maven Build System

Features of build systems

Any build system needs to provide a core set of features which should include the following:

- Artefact generation - we want our executable program at the end, just like in Make
- Unified testing - we need a unified way to run some tests on our code, we'll dive into more details on what testing is about next week
- Dependency resolution - modern software development is all about not reinventing the wheel, Maven provides a simple and declarative way of getting wheels invented by someone else on the internet (license permitting of course!)

You many be wondering, why can't we just stick with IntelliJ's project structure? From the outset, it seems to support adding dependencies and you can run your program with relative ease. However, IntelliJ's project structure is not portable; you can't run your project on another computer unless it has IntelliJ installed.

Let's discuss how Maven provides the portable aspect in the follow sections.

Convention over configuration

One of the most important aspect of Maven is that it *requires* a specific file structure and a specific set of files for your project work; the core philosophy of *Maven* is convention over configuration.

Per convention , Maven projects uses this directory structure:

```
├──.mvn - Maven wrapper binaries
├──mvnw - Maven wrapper for *nix
├──mvnw.cmd - Maven wrapper for Windows
├──pom.xml - Maven "Project Object Mode", or build configuration (mandatory)
├──README.md - include any other files for humans, maven don't care about these
└──src - source directory, contains all (only) source file (essentially mandatory)
    ├──main - application sources (essentially mandatory)
    │   ├──java - java source files
    │   └──resources - resources needed by other source files (optional)
    └──test - test sources (optional)
        ├──java - java test sources
        └──resources - resources needed by tests only
```

Note: Changing the overall directory structure will almost certainly cause the project to not compile, as expected.

Maven will perform compilation tasks with respect to this directory structure and handle all the intricacies of setting the classpath for you (it's more complicated than it should be because of how each platform handles paths).

Notice the `mvnw` , `mvnw.cmd` , and `.mvnw` entries, those are not part of the standard maven project. Those files are a wrapper files that delegate commands to the real maven executable and in cases where maven is missing, download and install it. We will revisit the use of these files in later sections.

The Project Object Model

Residing in the project root is the `pom.xml` file. This file describes the current project in XML, here's a sample `pom.xml` file:

```
<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.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging> <!-- we want a jar file as the executable -->

  <name>My Project</name> <!-- your project name here -->

  <dependencies>
    <dependencies>
    <!-- specify your dependencies here -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-suite</artifactId>
      <version>1.8.2</version>
      <scope>test</scope>
    </dependency>
    </dependencies>
  </dependencies>

</project>
```

The file is divided into several section:

Maven Coordinates

The maven coordinate looks like following:

```
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

It is used for others to find your project or for you to reference your own projects.

- `groupId` - a name to identify which group this package is from. By convention, it should be the root Java package name of the project.
- `artifactId` - the project name but without spaces or spacial characters.
- `version` - the version number of the project, can be any arbitrary string

As we'll be developing software for our own use, we can make something up here.

Dependencies

Maven projects can include other projects as a dependency. For example, in a later task we will be using a development tool call `JUnit`. In the following Maven fragment, we include `JUnit` as a dependency:

```
<dependencies>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

```
  <dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite</artifactId>
    <version>1.8.2</version>
    <scope>test</scope>
  </dependency>
  </dependencies>
```

All the required class/resources for JUnit will be downloaded and be ready to use in our project. Maven will also download additional dependencies that JUnit itself requires.

You don't need to worry about the operation of JUnit at this stage - we will introduce it properly later on in this unit (and probably in your other units as well !)

Available libraries

If you're interested in what sort of libraries is available for use, you can search for dependencies to use on sites such as these:

- mvnrepository
- The Central Repository

There are other root level tags that specify build plugins and reports which we won't go into for this unit, you can read more about them here.

As this isn't a course focused on DevOps or build systems, we will only use Maven as a tool to help everyone compile and build your Java project in a controlled and predictable way. Using Maven has the added advantage that TAs can better help you in case there are configuration problems.