Java Build Tools Exercises (1)

A Java Development Kit (JDK) contains the `javac` and `jar` tools as well as a JRE. This is what you need to develop in java.

`maven` is a Java package manager and build tool. It is not part of the Java distribution, so you will need to install it separately.

Installing on Debian

On Debian, install the `openjdk-17-jdk` and `maven` packages. This should set things up so you're ready to go but if you have *multiple versions* of Java installed you may need to set the `JAVA_HOME` and `PATH` variables to point to your install.

```
sudo apt install openjdk-17-jdk

sudo apt install maven
```

Running maven

Open a shell and type `mvn archetype:generate`. This lets you *generate an artifact from an archetype*, which is maven-speak for create a new folder with a maven file.

If you get a "not found" error, then most likely the maven `bin` folder is not on your path. If you're on a POSIX system and have used your package manager, this should be set up automatically, but if you've downloaded and unzipped maven then you have to `export PATH="$PATH:..."` where you replace the three dots with the path to the folder, and preferably put that line in your `~/.profile` too.

The first time you run it, maven will download a lot of libraries.

Maven will first show a list of all archetypes known to humankind (3046 at the time of counting) but you can just press ENTER to use the default, 2098

("quickstart"). Maven now asks you for the version to use, press ENTER again.

You now have to enter the triple of (groupId, artifactId, version) for your project - it doesn't really matter but I suggest the following:

```
groupId: org.example
artifactId: project
version: 0.1
```

If you're in a POSIX shell, then `find .` should show everything in the folder (in Windows, `start .` opens it in Explorer instead):

```
.
./src
./src/main
./src/main/java
./src/main/java/org
./src/main/java/org/example
./src/main/java/org/example/App.java
./src/test
./src/test/java
./src/test/java/org
./src/test/java/org/example
./src/test/java/org/example/AppTest.java
./pom.xml
```

his is the standard maven folder structure. Your java sources live under `src/main/java`, and the default package name is `org.example` or whatever you put as your groupId so the main file is currently `src/main/java/org/example/App.java`. Since it's common to develop Java from inside an IDE or an editor with "folding" for paths (such as VS code), this folder structure is not a problem, although it's a bit clunky on the terminal.

The POM file

Have a look at `pom.xml` in an editor. The important parts you need to know about are:

The artifact's identifier (group id, artifact id, version):

```
vagrant@debian12:~/groupId: org.example$ nano pom.xml

  GNU nano 7.2                              pom.xml

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.source
Encoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement><!-- lock down plugins versions to avoi
d using Maven defaul>
      <plugins>
        <!-- clean lifecycle, see https://maven.apache.org/re
f/current/maven-core>
        <plugin>
          <artifactId>maven-clean-plugin</artifactId>
          <version>3.1.0</version>
        </plugin>
        <!-- default lifecycle, jar packaging: see https://ma
```

```
ven.apache.org/ref/c>
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.22.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-install-plugin</artifactId>
          <version>2.5.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-deploy-plugin</artifactId>
          <version>2.8.2</version>
        </plugin>
        <!-- site lifecycle, see https://maven.apache.org/re
f/current/maven-core/>
        <plugin>
          <artifactId>maven-site-plugin</artifactId>
          <version>3.7.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-project-info-reports-plugin</arti
factId>
          <version>3.0.0</version>
        </plugin>
```

```
      </plugins>
    </pluginManagement>
  </build>
</project>
```

```
<groupId>org.example</groupId>
<artifactId>project</artifactId>
<version>0.1</version>
```

The build properties determine what version of Java to compile against (by passing a flag to the compiler). Unfortunately, the default maven template seems to go with version 7 (which for complicated reasons is called 1.7), but version 8 was released back in 2014 which is stable enough for us, so please change the 1.7 to 1.8 (there are some major changes from version 9 onwards, which I won't go into here):

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.source
Encoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

The dependencies section is where you add libraries you want to use. By default, your project uses `junit`, a unit testing framework - note that this is declared with `<scope>test</scope>` to say that it's only used for tests, not the project itself. You do not add this line when declaring your project's real dependencies.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
```

```
        <scope>test</scope>
    </dependency>
</dependencies>
```

The `<plugins>` section contains the plugins that maven uses to compile and build your project. This section isn't mandatory, but it's included to "lock" the plugins to a particular version so that if a new version of a plugin is released, that doesn't change how your build works.

The one thing you should add here is the `exec-maven-plugin` as follows, so that you can actually run your project:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
        <mainClass>org.example.App</mainClass>
    </configuration>
</plugin>
```

The important line is the `mainClass` which you set to the full name (with path components) of your class with the `main()` function.

Compile, run and develop

`mvn compile` compiles the project. The very first time you do this, it will download a lot of plugins, after that it will be pretty fast. Like `make`, it only compiles files that have changed since the last run, but if this ever gets out of sync (for example because you cancelled a compile halfway through) then `mvn clean` will remove all compiled files so the next compile will rebuild everything.

The `App.java` file contains a basic "Hello World!" program (have a look at this file). You can run the compiled project with `mvn exec:java` if you've set up the plugin as above. After you've run it the first time and it's downloaded all the files it needs,

lines coming from maven itself will start with `[INFO]` or `[ERROR]` or similar, so lines without any prefix like that are printed by your program itself. You should see the hello world message on your screen.

The development workflow is now as follows: you make your edits, then run `mvn compile test exec:java` to recompile, run your tests, then run the program. (Like `make`, you can put more than one target on a command, separated by spaces.)

`mvn test` runs the tests in `src/test/java`. There is an example test already created for you (have a look).

`mvn package` creates a jar file of your project in the `target/` folder.

I assume that you will be storing your Java projects in git repositories. In this case, you should create a file `.gitignore` in the same folder as the `pom.xml` and add the line `target/` to it, since you don't want the compiled classes and other temporary files and build reports in the repository. The `src/` folder, the `pom.xml` and the `.gitignore` file itself should all be checked in to the repository.

Exercise: make a change to the Java source code, then recompile and run with maven.