THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2212b - Winter 2014
Introduction to Software Engineering

Lab 2 - Build Automation with Maven

Part I
Introduction to Build Automation

Estimated Time: 10 minutes

1 Wrestling with the Classpath

Suppose we want to compile the Java classes `Class1.java` and `Class2.java`:

```
$ javac Class1.java Class2.java
```

"But wait," you say, "we can just use `*.java`!"

```
$ javac *.java
```

This works, but only if all of our classes exist in the same directory, and that's very rarely the case. If we need to compile `src/a/Class1.java` and `src/b/Class2.java`, for instance, then `*.java` won't work.

Now, suppose that `Class1.java` depends on a logging library `lib/log4j-1.2.17.jar`:

```
$ javac -cp lib/log4j-1.2.17.jar:. Class1.java
```

The `-cp` specifies the *classpath* – that is, the path in which `javac` should search for any classes for which it is looking. Above, we're telling it to look for classes both in the JAR file `log4j-1.2.17.jar`, and in the current directory (`.`). Suppose now that `Class2.java` depends on an HTTP client library `lib/httpclient-4.3.jar`:

```
$ javac -cp lib/log4j-1.2.17.jar:lib/httpclient-4.3.jar:. Class1.java \
          Class2.java
```

This is quickly becoming cumbersome. Now, multiply this simple example by *thousands* of source code files stored in *hundreds* of deeply nested subdirectories – a scenario not at all uncommon in a medium to large project – and one has quite the task on one's hands.

2 Dependency Hell

On top of that, consider that in any moderately-sized project, one is going to be using *dozens* of external libraries. We have to first download those libraries (i.e. the JAR files). We then have to worry about downloading our *transitive dependencies*: the libraries upon which our libraries depend. For instance, we might be using the Hibernate library. That only requires 1 JAR file to download, but it depends, in turn, on another 12 - 13 libraries, forcing us to download all of those as well.

We then have to ensure that we're downloading the correct versions of all of our libraries. Some libraries will have specific version requirements and we need to ensure that the versions of all libraries we've downloaded are both compatible with each other, and compatible with our own code.

Finally, as we saw above, we have to add each library to our classpath when compiling and also when running our programs.

Ideally, we'd also like some way to fix the version of any given library. Suppose Joe the Intern joins our team and is not very familiar with our code base. Joe sees that we're using version 1.1 of a library and notices that version 2.0 has been released. He downloads the library and adds it to the project. The problem? Our code might not *work* with the new version – going from version 1.1 to 2.0 is a *major version* number increment, which generally means *breaking changes* (i.e. it's not backwards compatible)[1]. If our code has been written to version 1.1 of the library's API, it probably won't work with the version 2.0 API. Additionally, one of our other dependencies might require version 1.1 of that library that Joe just upgraded. Now that dependency doesn't work either.

Thus, we need a way to ensure that the versions of our dependencies are managed and, ideally, we want something to automate and enforce this.

3 Other Motivations for Build Automation

We'll see later in this course that we often use *continuous integration servers* to automatically build and test our code for us, so we need a way to automate the building and testing process.

Typically, we also want to be able to automatically package our code. If we're developing an application or library, we might want a process that will automatically build our code, package it into a JAR file, and upload it to our web site where users can then download the latest version. If we're building a web site, we might want to be able to package our code as a WAR file and automatically deploy the latest version to our web server.

Finally, we may need to build packages for multiple platforms. Since you were in diapers, your well-meaning Computer Science professors have drilled it into your heads that Java is *write-once, run-everywhere* – that is, it is a multi-platform language. Contrary to what *they* told you, multi-platform issues do indeed arise in Java.

The Java Native Interface (JNI) allows C/C++ code to interoperate with Java code. We might have spent thousands of person-hours developing a library in C++. It will likely be far too expensive and time consuming to port all that code to Java, so we would ideally like to be able to make use of that C++ library.

Alternatively, we might need the performance of C/C++. If we're working heavily with graphics code in a game, for instance, then we're likely going to seek the performance of C.

As soon as we start talking about *native code* – in other words, binary code that runs outside of the Java virtual machine – we suddenly run into multiplatform issues, and we need separate builds for every platform we want to support (Windows, Linux, UNIX, Mac) and every architecture that we want to support (Intel 32-bit, Intel 64-bit, PowerPC, Itanium, SPARC, ...).

Hence, we would like a tool that can automate the builds of our code on all combinations of platforms and architectures that we are required to support.

Still think you can get away with `javac *.java`? *It ain't happenin'.*

[1] Read about Semantic Versioning at http://semver.org/spec/v2.0.0.html.

Part II
Maven

Estimated Time: 45 minutes

Maven is a build automation tool managed by the well-known Apache Software Foundation. It can be used with other languages, but is, in particular, highly adopted in the Java community. Incidentally, it's also a highly requested skill in the Java community, so it's a good thing to learn and put on your resume.

Maven handles the compilation of your code and the execution of your tests, and packages your code into an *artifact*, such as a JAR or WAR file. It also makes dealing with dependencies ridiculously easy. This is generally why most people start using Maven. We can tell Maven that we want to use version `x.y.z` of a library, and it will go download that library, along with all of its transitive dependencies. If you use Maven for no other reason, you'll want to use it for dependency management.

Maven also has a host of other features such as the ability to generate documentation, and the ability to generate a project web site containing information about our code. In this lab, however, we'll mostly focus on its dependency management capabilities, and see how we can use it to build and package our code. In the JUnit lab, we'll explore how to use it to test our code as well.

4 Installation

There are plenty of tutorials online detailing the installation of Maven, so we will assume you can do this on your own. See the installation instructions in *Maven: The Complete Reference* (http://books.sonatype.com/mvnref-book/reference/installation-sect-maven-install.html).

Note: if you are using Git Bash on Windows, you need to ensure that the directory that contains the Maven executable (`mvn.exe`) is added to your `PATH` environment variable **before** you open Git Bash. This is the same concept as was demonstrated with Java in the video posted for Lab 1.

5 Creating a Project

1. Change to your individual Git repository, create a `lab2` directory, and change to it:

```
$ cd ~/courses/cs2212/labs
$ mkdir lab2
$ cd lab2
```

2. Create a file `pom.xml` with the following contents:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>ca.uwo.csd.cs2212.USERNAME</groupId>
  <artifactId>USERNAME-lab2</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
</project>
```

   Of course, `USERNAME` should be replaced with your UWO username.

   `pom.xml` is a special file that indicates to Maven that the current directory is a Maven project. When we run the `mvn` command, Maven will search for `pom.xml`, and will exit with an error if it cannot find it.

   The `pom` in `pom.xml` stands for *Project Object Model*[2]. The file is an XML representation of our build configuration, and contains information about the project and configuration details used by Maven to build the

[2]Say that 10 times fast.

project. Put simply, it tells Maven what to do and what to generate.

We mentioned above that Maven generates an *artifact* such as a JAR or WAR file. There are four *coordinates* that uniquely identify an artifact:

- `groupId`: A string that uniquely identifies our project. For the `groupID`, we use the same conventions as Java package names, so if we own `eclipse.org` and we're working on the `jetty` project, then we'll use `org.eclipse.jetty` – the reverse domain name.
- `artifactId`: The name that Maven will use for our JAR file or whatever artifact type we're creating. Here, the convention is that we use `company_or_project_name-module_name`. So, if we're working on the `jetty` project, and within the `jetty` project, maybe we're developing an XML parsing module, then we would use `jetty-xml`. Similarly, if we worked for Dropbox and we were developing the Dropbox client that one installs on one's computer to sync files with the Dropbox service, we might use `dropbox-client`.
- `packaging`: The way we want to package our artifacts. The default is to package them as JAR files, but this can be changed easily to other formats we might need (WAR, EAR, RAR, etc.).
- `version`: The version of our application or library. The first number is the *major* version number, while the second is the *minor* version number. If a version has a third number, it is generally called the *patch* number, which is usually incremented when we release a bug fix. You can use anything you like for version numbers in Maven, but you should append the suffix `-SNAPSHOT` to the version number of any project that is currently in progress. The reason is beyond the scope of this lab, but the curious reader will find plenty of discussion of `-SNAPSHOT` online. For now, just know that if you're still working on the current version, then the version number should have `-SNAPSHOT` appended to it – this is important to Maven.

The `modelVersion` simply tells Maven the version of the XML schema we want it to use. For recent versions of Maven, this is `4.0.0`, so just put that in there and don't think about it again.

3. Create the directory structure to host our project:

```
$ mkdir -p src/{main,test}/{java,resources}
```

Maven adopts a *convention over configuration* approach: it is *opinionated* software. Rather than having you waste time coming up with conventions, it assumes you'll adopt its conventions. For instance, it assumes that:

- Source code will be located in `src/main/java`
- Tests will be located in `src/test/java`
- Resources (e.g. images) will be located in `src/main/resources`
- The project will generate a JAR file (by default)
- Compiled classes will be placed in `target/classes`
- The final artifact (JAR file) will be placed in `target`

By promoting this approach, Maven ensures consistency across all Maven projects. This makes it easier for us to join a project since we know exactly where everything is stored. Additionally, by following Maven's conventions, it will provide all sorts of functionality for us right out of the box: compiling, packaging, running tests, etc. We just put our code in the right directories, and Maven takes care of the rest!

4. Create the directory structure to house your Java package:

```
$ mkdir -p src/main/java/ca/uwo/csd/cs2212/USERNAME
```

Again, `USERNAME` should be your UWO username. You should get into the habit of using Java packages. Packages help us to avoid naming conflicts. Suppose we don't use packages and we create a class `DatabaseHelper` in the *default package*. We then import a poorly designed library that also does not use packages and also defines a class `DatabaseHelper` in the default package. The result? A compilation error. Hence, we organize our code into packages to eliminate the risk of naming conflicts.

Furthermore, packages are useful to bundle a set of related classes into a single, cohesive module. For instance, the Java API provides the `java.math` package, which contains classes for performing integer and decimal arithmetic, while the `java.net` package offers classes for implementing networked applications.

5. Create a class `App.java` in your package directory (i.e. in `src/main/java/ca/uwo/csd/cs2212/USERNAME`) with the following contents:

```
package ca.uwo.csd.cs2212.USERNAME;

public class App {

  public static void main(String[] args) {
    System.out.println("Hello Maven World");
    System.out.println("My username is USERNAME");
  }
}
```

6. Compile your code with Maven, and examine the contents of the `target` directory tree that was generated:

```
$ mvn compile
$ find target
```

   Observe that Maven has compiled our `App.java` class, and placed the resulting `class` file within the `target/classes` tree.

7. Package your code into a JAR file and list the contents of the `target` directory:

```
$ mvn package
$ ls -l target
```

   Notice that Maven has packaged our code into a JAR file. Pay attention to the name that Maven assigned to the JAR file: `artifactId-version.jar`. These are the values that we specified in our `pom.xml` file in Step 2.

8. Take a look at the contents of the JAR file:

```
$ jar tvf target/jshantz4-lab2-1.0-SNAPSHOT.jar
```

   Notice that `App.class` has been packaged into the JAR file, along with `pom.xml` and `MANIFEST.MF`, which provides some information to Java about the JAR file.

9. Change to the `target` directory, extract the contents of the JAR file, and view the contents of the manifest:

```
$ cd target
$ jar xvf jshantz4-lab2-1.0-SNAPSHOT.jar
$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: jeff
Build-Jdk: 1.6.0_65
```

   There's not much there right now, but we'll see how the manifest changes later as we add more to our `pom.xml` file. For now, we can see that Maven has added some details about what program was used to create the JAR file, the user that built the JAR file, and the JDK used to build it.

10. Run the `App` class from the JAR file:

```
$ java -cp jshantz4-lab2-1.0-SNAPSHOT.jar ca.uwo.csd.cs2212.jshantz4.App
Hello Maven World
My username is jshantz4
```

We can see that it runs, but it's cumbersome to have to type the full package name of the class we want to run.

11. Try running the JAR file with `java -jar`:

```
$ java -jar jshantz4-lab2-1.0-SNAPSHOT.jar
```

Notice that we receive the error `no main manifest attribute`. This is because the `MANIFEST.MF` in the JAR file does not contain an attribute that specifies the main class that is to be run when we execute the JAR file. We'll see how to fix this next.

6 Building an Executable JAR

Maven is a pluggable system, meaning that we can extend its functionality with plugins. When we add a plugin to the `pom.xml` file, Maven will automatically download that plugin from the Maven Central Repository (http://repo.maven.apache.org) and install it for us.

In this section, we'll see how to use the `maven-jar-plugin` to add a `Main-Class` attribute to the manifest of our JAR files so that they can be executed using `java -jar`.

1. Change back to the project root directory, edit `pom.xml`, and add the following:

```
<project>

  .
  .
  .
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <archive>
            <manifest>
              <mainClass>ca.uwo.csd.cs2212.USERNAME.App</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

2. Package the JAR file again:

```
$ mvn package
```

Observe in the Maven output that it downloads the `maven-jar-plugin` from the Maven Central Repository (http://repo.maven.apache.org).

3. Change to the `target` directory and extract the JAR file. Examine its manifest:

```
$ cd target
$ jar xvf jshantz4-lab2-1.0-SNAPSHOT.jar
$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Built-By: jeff
Build-Jdk: 1.6.0_65
Created-By: Apache Maven 3.0.5
Main-Class: ca.uwo.csd.cs2212.jshantz4.App
Archiver-Version: Plexus Archiver
```

Notice that the `Main-Class` attribute is now present in the manifest, meaning we should be able to run the JAR directly.

4. Run the JAR:

```
$ java -jar jshantz4-lab2-1.0-SNAPSHOT.jar
Hello Maven World
My username is jshantz4
```

It works!

7 Working with Dependencies

Suppose that we want to add logging to our program. A popular logging library for Java is the Apache log4j library. Let's integrate that into the program.

1. Change back to the project root directory
2. Edit `src/main/java/.../App.java` to look as follows:

```
package ca.uwo.csd.cs2212.USERNAME;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class App {

  static Logger logger = LogManager.getLogger(App.class.getName());

  public static void main(String[] args) {
    logger.trace("Entering main");
    logger.warn("Hello Maven / log4j World");
    logger.info("My username is USERNAME");
    logger.trace("Exiting main");
  }

}
```

3. Create a file `src/main/resources/log4j2.xml` with the contents below. You can copy/paste this from the following GitHub gist: https://gist.github.com/jsuwo/8706027#file-log4j2-xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
    <File name="File1" fileName="target/app.log" bufferedIO="false"></File>
  </Appenders>
  <Loggers>
    <Root level="trace" additivity="true">
      <AppenderRef ref="Console"/>
      <AppenderRef ref="File1" level="warn"/>
    </Root>
  </Loggers>
</Configuration>
```

Note: Characters often do not copy properly when copying from a PDF file. If you are going to copy/paste, please copy from the GitHub gist linked above.

This file is not related to Maven at all. Instead, it provides configuration details for the log4j library. Here, we're setting up two *appenders* for our logger. One will log all messages at the level `trace` or higher to the standard output. The other will log all messages at the level `warn` or higher to the file `target/app.log`.

Notice that we placed this file in `src/main/resources`. This directory is for various resource files that are used by our programs.

4. Add the dependencies to the `pom.xml`. For log4j, there are two dependencies we must add.

```
<project>

  .
  .
  .

  <dependencies>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>2.0-beta9</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
      <version>2.0-beta9</version>
    </dependency>
  </dependencies>

  <build>
  .
  .
  </build>

</project>
```

Observe that just as we specify a `groupId`, `artifactId`, and `version` for our project, we also must specify these for each of our dependencies, since these coordinates uniquely identify a Maven artifact.

If you are wondering how one would know what values to use here, there are several possibilities. In this case, this information was obtained from the log4j web site. Alternatively, we could have visited http://search.maven.org and searched for `log4j`. We would then have been presented with a list of artifacts matching that description.

5. Build and package the code again, and have a look at the contents of the JAR file:

```
$ mvn package
$ jar tvf target/jshantz4-lab2-1.0-SNAPSHOT.jar
```

Notice that the `log4j2.xml` file has been included in the JAR. This is another example of Maven's convention over configuration approach. We simply placed a file in the `src/main/resources` directory, and Maven knew to automatically bundle it in our JAR file.

6. Run the JAR:

```
$ java -jar target/jshantz4-lab2-1.0-SNAPSHOT.jar
```

We get a `NoClassDefFoundError` because Java cannot find the `LogManager` class. When we ran `mvn package`, Maven downloaded the dependencies we specified in `pom.xml` to our *local repository*, which is located on our computer in `~/.m2`. However, it does not automatically package dependencies with our JAR files. We'll see how to do that in the next section.

8 Building a Fat Jar

In the previous example, we could have successfully run the JAR if we had placed the log4j JARs on the classpath when running `java -jar`. However, if we have a lot of dependencies, this is tedious. Furthermore, since we're likely going to want to distribute our application, we want all of our dependencies bundled in a single JAR that a user can simply execute without having to worry about the classpath. In other words, we want a *fat jar*.

1. Add the following plugin to the `plugins` section in `pom.xml`. You can copy/paste this from the following GitHub gist: https://gist.github.com/jsuwo/8706027#file-pom-xml.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>ENTER CURRENT VERSION HERE</version>
  <configuration>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
```

```
    </descriptorRefs>
    <archive>
      <manifest>
        <mainClass>ca.uwo.csd.cs2212.USERNAME.App</mainClass>
      </manifest>
    </archive>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

2. Look up the current version of the `maven-assembly-plugin` on http://search.maven.org and replace the version placeholder in the XML with the appropriate version.

3. Change the `mainClass` element to reflect the name of your package.

4. Package your code again and list the `target` directory:

```
$ mvn package
$ ls -l target
```

   Observe that Maven has created two JAR files: `USERNAME-lab2-1.0-SNAPSHOT.jar` and `USERNAME-lab2-1.0-SNAPSHOT-jar-with-dependencies.jar`. You will also notice that the latter is much larger than the former, since it contains all of the dependencies of our program.

5. List the contents of the `target/...-jar-with-dependencies.jar` file:

```
$ jar tvf target/jshantz4-lab2-1.0-SNAPSHOT-jar-with-dependencies.jar
```

   Notice that all of the log4j classes have been bundled in the JAR.

6. Execute the JAR:

```
$ java -jar target/jshantz4-lab2-1.0-SNAPSHOT-jar-with-dependencies.jar
06:00:01.432 [main] TRACE ca.uwo.csd.cs2212.jshantz4.App - Entering main
06:00:01.433 [main] WARN  ca.uwo.csd.cs2212.jshantz4.App - Hello Maven / log4j World
06:00:01.434 [main] INFO  ca.uwo.csd.cs2212.jshantz4.App - My username is jshantz4
06:00:01.434 [main] TRACE ca.uwo.csd.cs2212.jshantz4.App - Exiting main
```

7. Print the contents of `target/app.log`:

```
$ cat target/app.log
Hello Maven / log4j World
```

   Everything is working as expected. Messages at level `trace` or higher are being logged to the standard output, while messages at level `warn` or higher are being logged to `target/app.log`. We successfully integrated log4j into our program and created an executable, fat JAR that contains all of our dependencies.

Part III
Conclusion

Estimated Time: 5 minutes

9 `.gitignore`

We don't want to check in the `target` directory to our source control. After all, we can easily rebuild this directory using `mvn package`.

1. Run `mvn clean` to remove the `target` directory:

   ```
   $ mvn clean
   $ ls -l
   ```

   You will notice that `target` no longer exists.

2. Rebuild the project using `mvn package`:

   ```
   $ mvn package
   $ ls -l
   ```

   Since it is this easy to regenerate the `target` directory (and thus the project artifacts), we should avoid checking this directory in to GitHub.

3. Change to the **main** repository directory, and edit the `.gitignore` file to tell Git to ignore the `target` directory:

   ```
   *.class
   target/
   ```

   Note that the first line should still be present from lab 1.

4. Type `git add .` and `git status` and verify that Git does not show the `target` directory as being untracked.

10 Submitting Your Lab

Commit your code and push to GitHub. To submit your lab, create the tag `lab2` and push it to GitHub. For a reminder on this process, see Lab 1.

11 But My IDE Builds My Code For Me!

We've barely scratched the surface of what Maven can do, but we'll see some more uses for Maven as the course progresses.

One question that may be lingering in your mind is "Why do I need Maven when my IDE already takes care of compiling my code for me?"

There are a few reasons for this. First, your IDE almost certainly does not download the dependencies of your project automatically, along with all transitive dependeicies. It may work for the relatively small Computer Science assignments you've completed to date, but managing transitive dependencies by hand in even a modestly sized project is just not feasible.

Also, relying solely on an IDE's build process is not a great idea. Not everyone uses the same IDE. Not everyone uses the same *version* of an IDE. Not everyone uses an IDE *at all*. Additionally, most projects are using continuous integration systems that automatically build your code and run your tests on a server each time you check in changes to your source control repository. These systems are headless – meaning that they don't use a GUI – so they need a way to build the project at the command line in an automated way.

Most IDEs offer support for working with Maven projects, so you can still use your IDE and have it use Maven to do the building, testing, and packaging that it would normally take care of itself. This is beyond the scope of this lab, but you are encouraged to explore Maven integration in your IDE of choice.