The Joy of Coding

> Apache Maven provides developers with a standard set of tools and conventions for building and deploying Java applications. Maven manages the integration of third-party libraries in a project and provides a plug-in architecture that allows new tools and reports to be easily integrated into a project's build system.

Building Java applications with Maven

- A history of building Java projects
- Maven: building, testing, and reporting

Why a build tool for Java?

As an application scales in size, its build process must be automated

The UNIX make utility has been around for years

- User specifies the dependencies among various application components in a "makefile"
- The `make` program analyzes those dependencies and rebuilds components as necessary
- Don't have to rebuild the entire product when you only change one source file

Make doesn't fit well with Java

- Dependencies among Java files are complex
- Make is rarely platform independent
- Syntax of makefiles is wiggy

Apache Ant

The Apache Foundation developed Ant in 2000 to build their Java projects

- Ant provides the building blocks for compiling, assembling, and testing a Java project
- Ant has tasks that invoke tools like `javac`, `javac`, and `javadoc`
- Tasks are grouped together into build targets that are configured in an XML file that specifies source file locations and class paths

Ant worked very well, but...

- Each project had its own build procedure with different directory layouts and naming conventions
  - Ramp-up time for new developers
- Dependencies on third-party libraries had to be managed by individual projects
  - Jar files were often checked into CVS along with source code
  - Often difficult to determine which version of the library was used

Enter Apache Maven

Developers noticed common patterns in the projects that they built

- Running unit tests as part of every build
- Generating reports and documentation about the project
- Versioning the software and deploying it to a central location so it can be shared

Apache Maven was developed to provide a standard build environment that could be used by many projects

- Each project produces some artifact (jar file, web application, etc.) from source code
- Unit tests are fully integrated into the project
- Third-party libraries are automatically downloaded and placed on the classpath
- Artifacts and supporting information (documentation and reports) can be published to a centralized repository for consumption by others

Getting started with Maven

Maven can be downloaded from

`http://maven.apache.org`

The `mvn` command line tool executes Maven

A new Java project can be created using a Maven "archetype"

```
$ mvn archetype:create \
  -DgroupId=edu.pdx.cs.joy.whitlock \
  -DartifactId=proj1
```

This generates a very simple Java project that can be built with Maven

```
$ find proj1 -type f
proj1/pom.xml
proj1/src/main/java/edu/pdx/cs/joy/whitlock/App.java
proj1/src/test/java/edu/pdx/cs/joy/whitlock/AppTest.j
```

`pom.xml` is the Maven "Project Object Model" configuration file that specifies how the project is built

Layout of a Maven project

Maven defines a standard layout the project's source files*.

```
+- proj1/
   +- pom.xml
   +- src/
      +- main/
         +- java/  (Base dir for source code)
            +- edu/pdx/cs/joy/whitlock/
         +- resources/ (Files on classpath)
            +- META-INF/MANIFEST.MF
      +- test/
         +- java/  (Base dir for test code)
            +- edu/pdx/cs/joy/whitlock/
         +- resources/ (Files on test classpath)
      +- site/     (Files for web site)
         +- site.xml (Describes site)
         +- apt/   (Supplementary site files)
         +- resources/
           +- css/
             +- site.css
           +- images/
   +- target/
      +- proj1-1.0.jar     (Project artifact)
      +- surefire-reports/ (Unit test output)
      +- site/ (Generated web site)
```

*The layout can be configured, if necessary.

What can you do with Maven?

Maven defines several build "phases" that perform common tasks

`mvn compile`	Compiles Java source code in `src/main/java`
`mvn test`	Recompiles necessary source code and runs unit tests
`mvn package`	After running unit tests, builds a jar file with the compiled source code
`mvn site`	Generates a web site for the project with information gleaned from `pom.xml`
`mvn clean`	Deletes all build artifacts
`mvn eclipse:eclipse`	Generates an Eclipse project based on the Maven project

Build artifacts are placed in the `target` directory

Looking at pom.xml

```
<project xmlns=''http://maven.apache.org/POM/4.0.0''
  xmlns:xsi=''http://www.w3.org/2001/XMLSchema-instan
  xsi:schemaLocation=''http://maven.apache.org/POM/4.
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.pdx.cs.joy.whitlock</groupId>
  <artifactId>proj1</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>proj1</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>
```

This POM configures version `1.0-SNAPSHOT` of `proj1` released by the `edu.pdx.cs.joy.whitlock` organization. The project builds a `jar` file and its tests depend on version `3.8.1` of `junit`.

Information about the project

The POM can contain several pieces of information about the project that Maven uses for various purposes*

`name`	The name of the project (often a code name such as “Guatemala”)
`description`	General description of project
`url`	URL where users can learn more about project
`inceptionYear`	When project was started

You can also include information about the project

- Organization that develops the project (`name`, `url`)
- Developers and contributes (`id`, `name`, `email`, `role`, a picture URL)
- License under which it is distributed
- Bug tracking system, Mailing lists
- Source code repository, Automated build server

*These are XML elements that are children of the top-level `project` element.

Managing dependencies

A project often requires third-party libraries to do its work. A POM specifies what libraries the project depends on and Maven downloads them as needed.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>
```

The `scope` element determine which build phase the dependency applies to

- `compile` (default): library and all of its dependencies are available to all build phases
- `runtime` not required for compilation; only needed to execute tests and run project
- `provided`: only needed at compile time; someone else provides it at runtime
- `test`: only needed to compile and run tests
- `system`: library is not available in repository
  - Location is provided by the `systemPath` POM element

Repositories

Maven specifies a layout for repositories of deployed artifacts

- The layout of the repository is based on the artifact’s group id, version, and id
- The JUnit 3.8.1 artifact is located at `junit/junit/3.8.1/junit-3.8.1.jar`
- The repository also contains a copy of the artifact’s POM so that artifacts that it depends on can be resolved

By default, Maven will search for an artifact in a public repository:

`http://repo1.maven.org/maven2/`

A POM can also specify repositories to search:

```
<repositories>
  <repository>
    <id>OurRepo</id>
    <name>My Company's Maven Repo</name>
    <url>https://internal.mycompany.com/repository/
  </repository>
</repositories>
```


Local Repository

Maven downloads project dependencies to a local repository that resides in your home directory:

`${user.home}/.m2/repository`

When resolving a project dependency, Maven will search the local repository before searching other repositories.

- Running `mvn` with `-o` or `--offline` will prevent it from accessing remote repositories

Build phases

A Maven build consists of a number of phases (each phase depends on the previous)

validate	Validates `pom.xml` and downloads necessary dependencies
compile	Compiles the source code
test	Tests the source code (can't require that code be packaged)
package	Builds the project artifact (jar file)
integration-test	Deploys the artifact (e.g. to a web server) for integration testing
verify	Makes sure that artifact meets quality criteria (coding standards, etc.)
install	Copies artifacts to the local Maven repository for use in other local projects
deploy	Publishes artifacts to remote Maven repository for use by others

Each build phase is dependent on the previous

- Running `mvn package` will invoke `validate`, `compile`, and `test`

Build Goals

Each build phase consists of goals that perform some build task like compiling source code, running tests, or building a jar file

Each phase has a default goal that depends on the projects `packaging` (`jar`, `war`, etc.)

compile	`compile:compile`
test-compile	`compiler:testCompile`
test	`surefire:test`
package	`jar:jar`

Maven *plugins* are artifacts that provide goals to a Maven build

- The plugin for compiling Java source has two goals: one for the `main` code (`compiler:compile`) and other for `test` code (`compiler:testCompile`)
- Plugins are configured in the POM and must be associated with a build phase

Information about the standard Maven plugins can be found at:

`http://maven.apache.org/plugins/index.html`

Compiling Source Code

Running `mvn compile` will compile the source code in `src/main/java` into `target/classes`

This will instruct the compiler to use Java 5 language features*:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId
      <version>2.4</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This compiler setting applies to any build phase (`main`, `test`, etc.) that compiles Java source.

Note that it is a best practice to specify the version your plugins. That way, all users of your build will get the same configurations.

*By default, Maven 2.0 uses `source 1.3`. Lame.

Running Tests

Running `mvn test` will

- Perform the `compile` phase
- Compile the source in `src/test/java` into `target/test-classes`
- Run the unit tests and place their output in `target/surefire-reports`

The `surefire` plugin runs unit tests

- `mvn surefire:test` runs the tests without recompiling
- Running `mvn test -DskipTests` will compile tests, but not run them

`surefire` supports both the JUnit and TestNG Java unit testing frameworks

Building a jar file

Running `mvn package` will

- Perform the `compile` and `test` phases
- Assemble `target/artifactId-version.jar` with the classes in `target/classes`

The `jar` plugin configures the jar's manifest

```
<build><plugins><plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.4</version>
  <configuration><archive>
    <manifest>
      <mainClass>edu.pdx.cs.joy.grader.Submit</mainCl
    </manifest>
  </archive></configuration>
</plugin></plugins></build>
```

Creates an "executable jar" whose execution starts with the `Submit` class

```
$ java -jar target/grader-1.0.jar
```


Running integration tests

The `integration-test` phase runs integration tests that test the project in the environment in which it is deployed

- Integration tests exercise "external" functionality like a command line tool, REST API, or functionality that requires software other than the project's artfiact
- The `pre-integration-test` and `post-integration-test` phases can be used to configure, start, and stop web containers, databases, etc.

The `failsafe` plugin run the integration tests

- By convention, the names of integration tests end with `IT` instead of `Test`
- If the integration tests fail, the `integration-test` phase does not fail.
  - This ensures that `post-integration-test` runs
- The `verify` phase checks to see if there were failures during `integration-test` and, if so, fails
- So, you need to run `verify` when running integration tests

Reports

There are Maven plugins that generate reports about your project

- `mvn javadoc:javadoc` generates API documentation
- `mvn jxr:jxr` generates a cross-reference of the project's source code
- `mvn surefire-report:report` runs the `test` phase a generates a HTML summary of test results
  - Will use the `jxr` report, if available, to link stack traces to source code
- `mvn pmd:pmd` will analyze your source code and look for smelly code
- `mvn findbugs:findbugs` will find common subtle mistakes in your program

Maven downloads and runs these reporting tools with virtually no work on your part!

Reports

Reporting plugins are configured in the `reporting` section of the POM

```
<reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jxr-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <linkJavadoc>true</linkJavadoc>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.8.1</version>
        <configuration>
          <links>
            <link>http://java.sun.com/javase/6/docs/a
          </links>
        </configuration>
      </plugin>
    </plugins>
  </reporting>
```

A Project's Web Site

Running `mvn site` will generate a web site for your project that includes

- Information about the project and its members (from POM)
- Project reports (API docs, test results, code analysis, etc.)
- Other pages that you specify
  - Content is authored in the APT ("Almost Plain Text") format using wiki-like syntax

Installing a Project

Running `mvn install` runs the `package` phase and copies the artifact into the local repository

- Now the project can be used with other projects on your machine

Similarly, you can install a third-party library that is not available from a public Maven repository with:

```
mvn install:install-file \
  -Dfile=/home/dave/watij.jar \
  -DgroupId=com.watij -DartifactId=watij \
  -Dversion=3.2.1 -Dpackaging=jar
```


Deploying a Project

Running `mvn deploy` builds the project and publishes artifacts to a remote Maven repository

- Files can be uploaded via FTP, SCP
- The `site` can also be uploaded to a public web site

The POM describes the repository to which the artifact is uploaded

```
<distributionManagement>
  <repository>
    <id>whitlock-PSU</id>
    <name>David Whitlock's PSU Repository</name>
    <url>scp://cs.pdx.edu/u/whitlock/public_html/re
  </repository>
  <site>
    <id>whitlock-PSU</id>
    <name>David Whitlock's PSU Web Site</name>
    <url>scp://cs.pdx.edu/u/whitlock/public_html/si
  </site>
</distributionManagement>
```

`${user.home}/.m2/settings.xml` contains username/password or private key information used when authenticating with the server

Deploying Third-Party Libraries

It is recommended that organizations have their own Maven repository

- A place to deploy their projects
- Developers can still build if no internet connection
- Can verify that artifacts come from a trusted source
- It's really easy to do (HTTP and SCP)

Running `deploy:deploy-file` will upload an artifact to a remote repository

```
$ mvn deploy:deploy-file \
  -Durl=scp://cs.pdx.edu/u/whitlock/public_html/repos
  -DrepositoryId=whitlock-PSU \
  -Dfile=/home/dave/watij.jar \
  -DgroupId=com.watij -DartifactId=watij \
  -Dversion=3.2.1 -Dpackaging=jar
```

Maven generates a POM file for the artifact on the server

- `-DpomFile` specifies a `pom.xml` for the third-party library that contains its dependencies

Sub-projects

Projects that have multiple components or artifacts (e.g. an API library and a web application) can be broken into multiple projects.

A parent Maven project (POM) provides configuration that is shared among the child projects (modules)

```
<project>
  <groupId>com.mycompany</groupId>
  <version>1.0</version>
  <artifactId>ecommerce</artifactId>
  <packaging>pom</packaging>
  <modules>
    <module>db</module>
    <module>business</module>
    <module>web</module>
  </modules>
</project>
```

Each module resides in a directory beneath its parent project

Some reports (`javadoc` and `jxr`) can be configured to roll-up info from all sub-projects into one report.

Sub-projects

Child projects refer to their parent and depend on their siblings' artifacts (from `business/pom.xml`):

```
<project>
  <parent>
    <artifactId>ecommerce</artifactId>
    <groupId>com.mycompany</groupId>
    <version>1.0</version>
  </parent>
  <groupId>com.mycompany</groupId>
  <version>1.0</version>
  <artifactId>business</artifactId>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>com.mycompany</groupId>
      <artifactId>db</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
</project>
```


Summary

Maven provides a standard framework for developing Java projects

- Manages (and downloads) dependencies
- Compiles code, run tests, builds project artifacts
- Runs reports, publishes artifacts and web site

You don't need to download and manage libraries and tools anymore!

As long as you play by Maven's rules, Maven will be very good to you.