

maven

By: Rajesh Kumar
rajesh@scmgalaxy.com

www.scmGalaxy.com

Topics

- **Installation**
- **Shock and Awe: Comparison with ant**
- **Project Object Model (POM)**
- **Inheritance and Modules**
- **Dependencies**
- **Build Configuration**
- **Whirlwind Tour of Plugins**
- **Lifecycles**
- **Build Profiles**
- **Sometimes Maven Lets You Down**

Installation

- **Requires JDK 1.4**
- **Download Maven from <http://maven.apache.org/>**
- **Unzip the archive**
- **Add the M2_HOME environment variable and point it to your installation.**
- **Make sure JAVA_HOME is set and java is in your PATH.**
- **Make sure \$M2_HOME/bin is in your PATH.**

Maven Terminology

With maven, you execute **goals** in **plugins** over the different **phases** of the **build lifecycle**, to generate **artifacts**. Examples of artifacts are jars, wars, and ears. These artifacts have an **artifactId**, a **groupId**, and a **version**. Together, these are called the artifact's "**coordinates**." The artifacts stored in **repositories**. Artifacts are *deployed* to remote repositories and *installed* into local repositories. A **POM** (Project Object Model) describes a project.

Create a Project Directory



Step

1

**Maven has a command
for starting a project:**

```
mvn archetype:create \  
-DgroupId=com.techmaine \  
-DartifactId=demo-mvn \  
-DpackageName=com.techmaine.mvndemo \  
-Dversion=1.0
```

Create a Project Directory



Step
1

Plugin Name

```
mvn archetype:create \  
-DgroupId=com.techmaine \  
-DartifactId=demo-mvn \  
-DpackageName=com.techmaine.mvndemo \  
-Dversion=1.0
```

Create a Project Directory



Step
1

Plugin Name
Goal

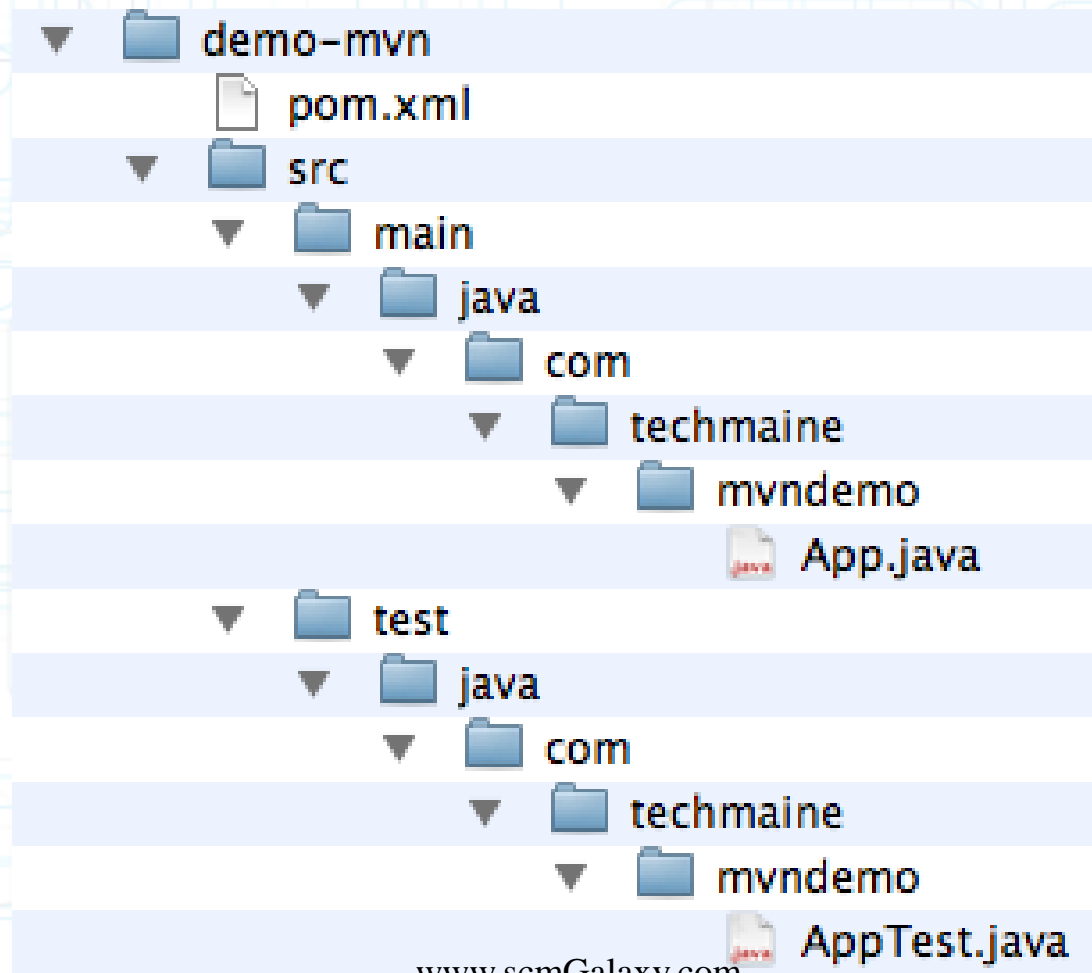
```
mvn archetype:create \  
-DgroupId=com.techmaine \  
-DartifactId=demo-mvn \  
-DpackageName=com.techmaine.mvndemo \  
-Dversion=1.0
```

Create a Project Directory



Step
1

Voila! Look what maven has done for you:



www.scmGalaxy.com

Set up the dependencies



Step
2

Open pom.xml. We need to tell maven that we have a dependency on Hibernate:

```
<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>com.techmaine</groupId>
  <artifactId>demo-mvn</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>demo-mvn</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>
```

We'll add the dependency here.

Set up the dependencies



Step

2

This is all we need to add:

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-annotations</artifactId>  
  <version>3.3.1.ga</version>  
</dependency>
```

We don't need to tell Maven about any of the jars on which Hibernate depends; Maven takes care of all of the transitive dependencies for us!

Set up the compiler



Step

3

STUPIDITY ALERT!

Maven assumes a default source version of 1.3. We need to tell it if we want 1.5. Here's a preview of plugin configuration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

www.scmGalaxy.com

Set up Hibernate Configuration



Step
4

Create a resources directory beneath the main (and, optionally, test) directory, and put the Hibernate configuration file there.

- Files in the resources directory get copied to the root of the classpath when packaging occurs as part of the resource:resource goal (more on that later)
- The resources directories are automatically created for many of the archetypes, but not for the quickstart archetype that we used.

Package



Step
5

Next, package everything up before we run it.

To do this, invoke maven thusly:

```
mvn package
```

This is an alternate way to invoke maven. Instead of specifying a plugin and goal, you specify a phase (in this case, package is the phase). A phase is a sequenced set of goals. The package phase compiles the java classes and copies the resources

Execute



Step

6

Next, use the exec plugin to run our application:

```
mvn exec:exec \  
-DmainClass=com.techmaine.mvndemo.App
```

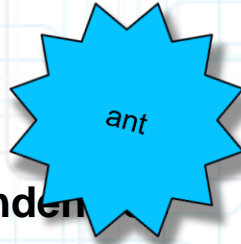
Recap



We told maven

- That we were making a “quickstart” project.
- That we depended on Hibernate Annotations.
- That we needed Java 1.5
- pom.xml was 35 lines (would have been 22 if maven defaulted to Java 1.5 instead of 1.3)

Recap – Why Maven is Cool



We downloaded jars and dependencies ourselves

We told ant

- The name of the jar file that we needed (Hibernate)
- All the dependent jar file names
- Where the jar files were located
- That it needed to compile java files
- Where the java files were located
- Where it should put the class files
- Where it should put the Hibernate configuration file
- How to run the application
- Where the jar and class files were located (again, this time for runtime)
- build.xml is 75 lines, but who's counting?



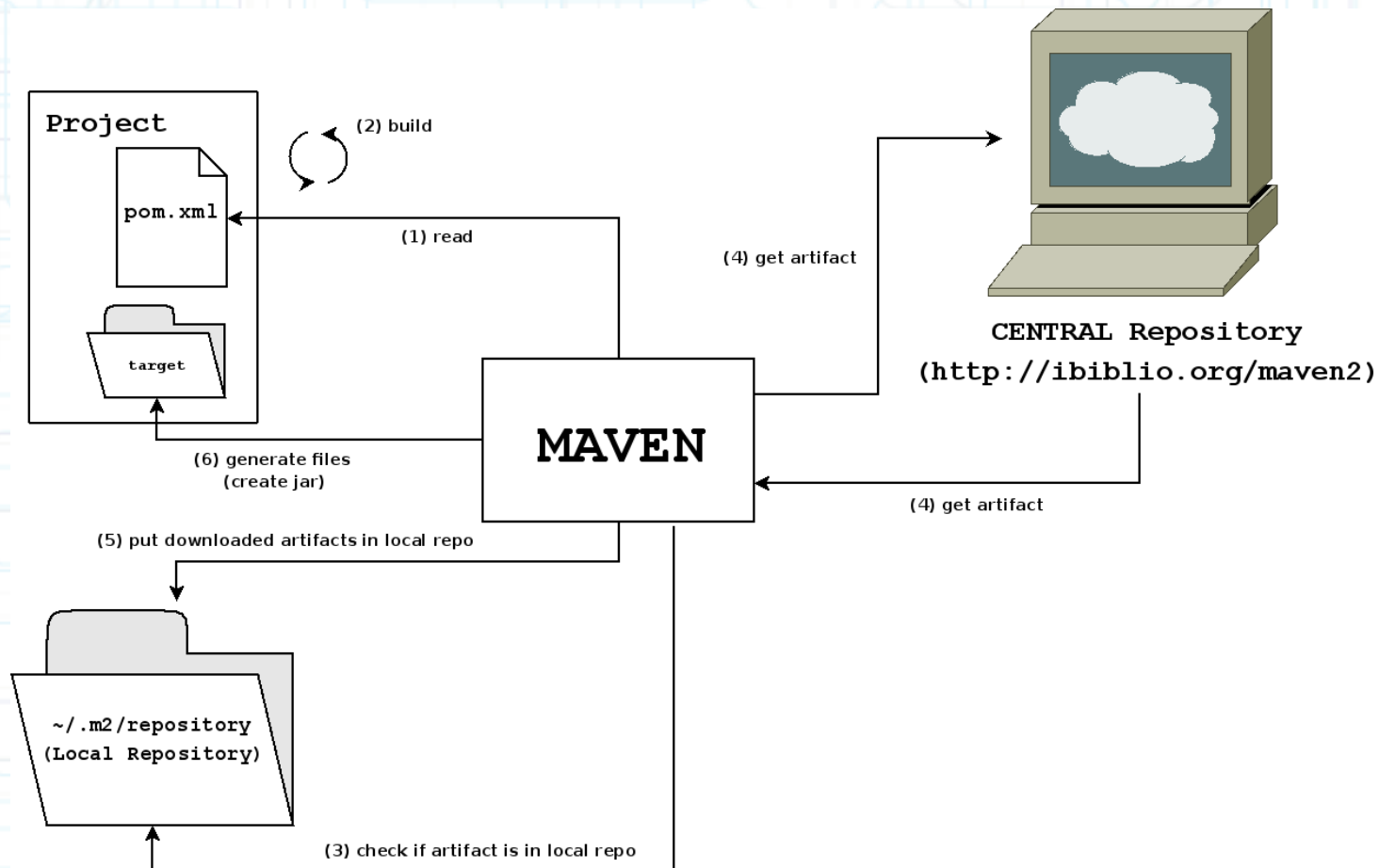
We told maven

- That we were making a “quickstart” project.
- That we depended on Hibernate Annotations.
- That we needed Java 1.5
- pom.xml was 35 lines (would have been 22 if maven defaulted to Java 1.5 instead of 1.3)

Maven Vs Ant

Build Configuration Aspects	Maven	Ant
Build File Name	pom.xml	build.xml
Project Name	Manual (line 4)	Manual (line 1)
Project Directory Structure	Automatic	Manual (lines 4-6)
Build Preparation	Automatic	Manual (init target)
Library Dependencies	Automatic	Manual (not in build ex)
Compile	Automatic	Manual (compile target)
Testing	Automatic	Manual (not in build ex)
Packaging	Automatic	Manual (dist task)
Versioning	Automatic	Manual (tstamp)
Deployment	Automatic	Manual (not in build ex)
Site Generation	Automatic	Manual (not in build ex)
Clean-up	Automatic	Manual (clean task)

Maven in a nutshell



What can Maven do?

When you first download it, almost nothing!

- Run goals
- Run phases (collections of goals)
- Download Dependencies*
- Download Plugins

* Actually, dependency downloads are done by a plugin, too.

But... from where?

Configuring Maven

- **Settings Files (settings.xml)**
 - In `~/.m2` (per-user settings) and in Maven's install directory, under `conf` (per-system settings)
 - Alternate location for repository
 - Proxy Configuration
 - Per-server authentication settings
 - Mirrors
 - Download policies, for plugins and repositories; snapshots and releases.

Configuring Maven

- **Project Object Model (pom.xml)**
 - **Inherited – individual projects inherit POM attributes from parent projects, and ultimately inherit from the “Super POM”**
 - **The Super POM is in Maven’s installation directory, embedded in the uber jar.**
 - **The Super POM defines, among lots of other things, the default locations for the plugin and jar repositories, which is <http://repo1.maven.org/maven2>**

Repositories

- **Local - in `~/.m2/repository`**
- **Remote - e.g., `http://repo1.maven.org/maven2` or another internal company repository (any directory reachable by sftp will do).**
- **Contains dependencies and plugins**
- **Can be managed by a “Repository Manager” like Nexus**

The POM

- **Describes the project, declaratively**
- **General Information - Project Coordinates (groupId, artifactId, Version)**
- **Build Settings – Configuration of the plugins**
- **Build Environment – We can configure different profiles that can be activated programmatically**
- **POM Relationships – Dependencies on other projects**

Anatomy of a POM File

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>Super Duper Amazing Deluxe Project</name>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```


General Information

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <version>1.0.0</version>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

} **Coordinates**

Project Inheritance

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <version>1.0.0</version>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Project Inheritance

What is inherited?

- **Identifiers (groupid, artifactId, one must be different)**
- **Dependencies**
- **Plugin, Report Lists**
- **Plugin Configurations**

Why Inherit?

- **Don't repeat yourself, e.g., several projects use the same version of log4j.**
- **Enforce plugin version across projects**

Multimodule Projects

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <version>1.0.0</version>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Multimodule Projects

- ***Not the same thing as POM inheritance!***
- **A multimodule project builds submodules, but rarely produces an artifact itself**
- **Directory structure mimics module layout (e.g., if B is a submodule of A, then B will be a subdirectory of A).**

Multimodule: Reactor

- **When Maven encounters a multimodule project, it pulls all of the POMs into the “Reactor”**
- **The Reactor analyzes module inter-dependencies to ensure proper ordering.**
- **If no changes need to be made, the modules are executed in the order they are declared.**
- **Maven then runs the goals on each module in the order requested.**

Maven Properties

Build properties

Defined in pom.xml `<properties>` element

Inherited from parent poms

System properties are copied over build properties

Usage

Plugin configuration

Dependency versions

Dependency management

Filtering

Supported in resources plugin, war plugin and some more.

Enabled by `<filtering>true</filtering>`

Standard resources in build definition

Plugin configuration for war plugin

Replaces `${xxx}` by corresponding build property.

During copy, not in place.

Build property can be overridden by system property.

At build time! Not changed later.

User-Defined Properties

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <version>1.0.0</version>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

www.scmGalaxy.com

User-Defined Properties

- **User-Defined properties are like ant properties:**

```
<properties>
  <hibernate.version>3.3.0.ga</hibernate.version>
</properties>
...
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>${hibernate.version}</version>
  </dependency>
</dependencies>
```

Example from *Maven: The Definitive Guide*, Sonatype, O'Reilly p.266

www.scmGalaxy.com

Other Properties

- Maven Properties, project.*
`${project.version}`
- Settings Properties, settings.*
`${settings.interactiveMode}`
- Environment Variables, env.*
`${env.JAVA_HOME}`
- Java System Properties
`${java.version}`, `${os.arch}`, `${user.dir}`

Dependencies

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <version>1.0.0</version>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Dependencies

Maven's *pièce de résistance*

```
<dependencies>  
  <dependency>  
    <groupId>com.techmaine</groupId>  
    <artifactId>awesome-lib</artifactId>  
    <version>1.3.5</version>  
    <scope>compile</scope>  
    <optional>>false</optional>  
  </dependency>  
</dependencies>
```

Dependencies

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>awesome-lib</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
    <optional>>false</optional>
  </dependency>
</dependencies>
```



groupId and artifactId: must be unique

Dependencies

```
<dependencies>  
  <dependency>  
    <groupId>com.techmaine</groupId>  
    <artifactId>awesome-lib</artifactId>  
    <version>1.3.5</version>  
    <scope>compile</scope>  
    <optional>>false</optional>  
  </dependency>  
</dependencies>
```

- **1.3.5** - prefer version 1.3.5, newer version is acceptable to resolve conflicts
- **(1.3.4,1.3.9)** - Any version between 1.3.2 and 1.3.9, exclusive
- **[1.3.4,1.3.9]** - Any version between 1.3.2 and 1.3.9, inclusive
- **[,1.3.9]** - Any version up to, and including, 1.3.9
- **[1.3.5]** - Only version 1.3.5, do not use a newer version.


Dependencies

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>awesome-lib</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
    <optional>>false</optional>
  </dependency>
</dependencies>
```

- **compile** - default, packaged. Available on compile-time and runtime CLASSPATH.
- **provided** - you expect the JVM or app container to provide the library. Available on compile-time CLASSPATH.
- **runtime** - needed to run, but not compilation (e.g., a JDBC driver)
- **test** - only needed during test execution (e.g., JUnit)

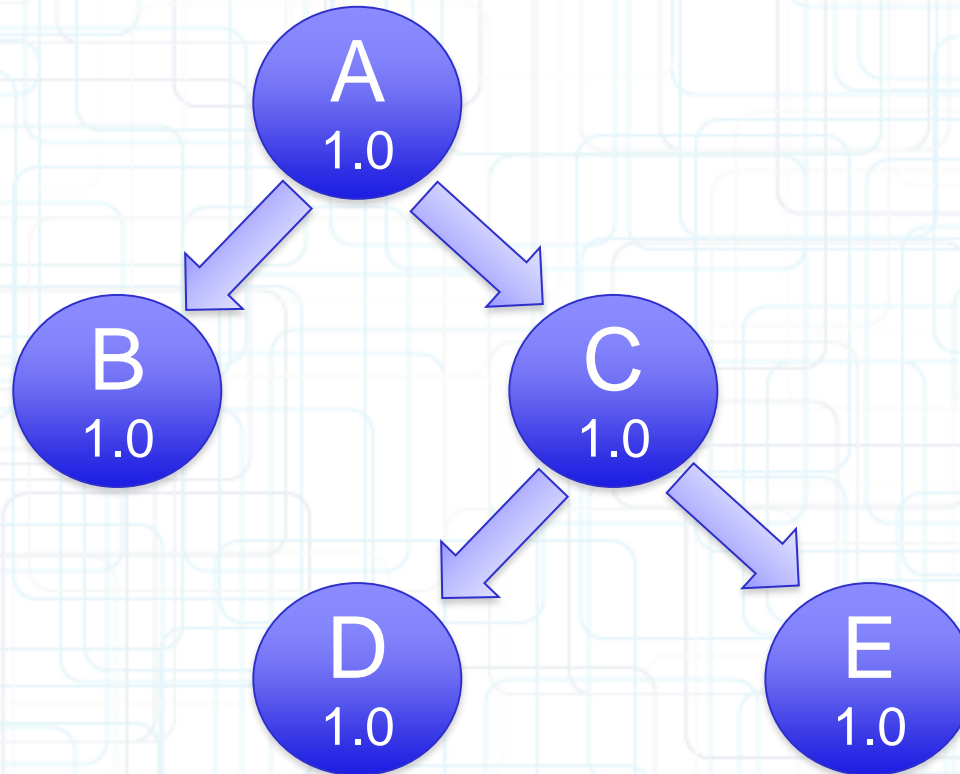
Dependencies

```
<dependencies>  
  <dependency>  
    <groupId>com.techmaine</groupId>  
    <artifactId>awesome-lib</artifactId>  
    <version>1.3.5</version>  
    <scope>compile</scope>  
    <optional>false</optional>  
  </dependency>  
</dependencies>
```



Prevents this dependency from being included as a transitive dependency if some other project depends on this project.

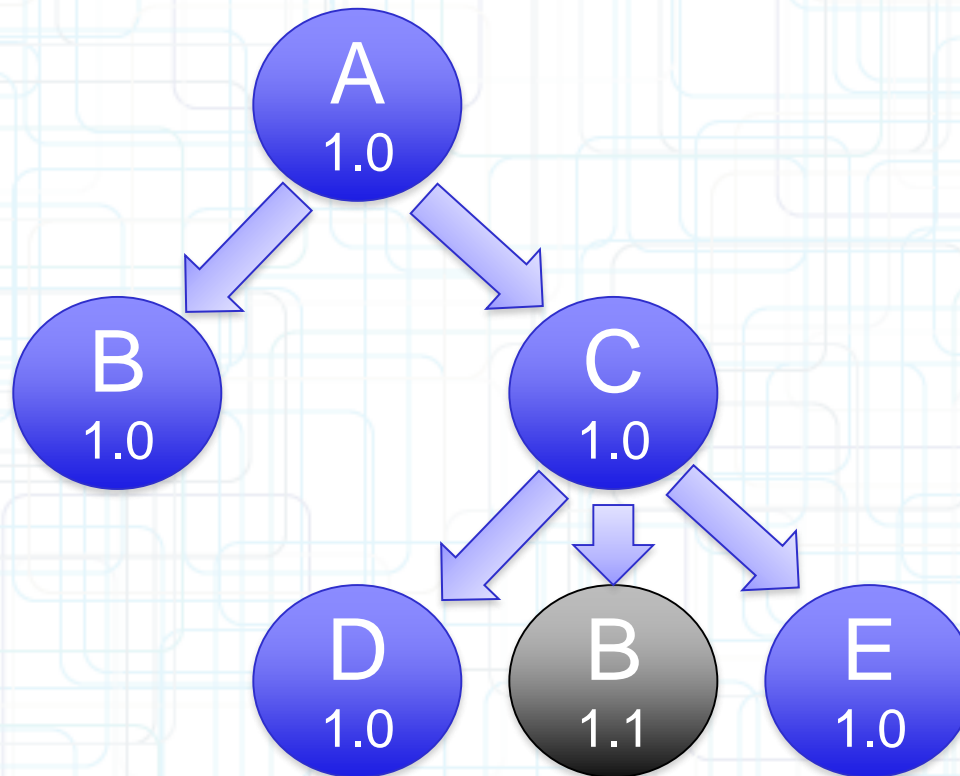
Transitive Dependencies



Our project (Project “A”) depends on B and C. Project C depends on projects D and E. Thus, our project depends on B, C, D, and E, and Maven will fetch and use these artifacts appropriately.

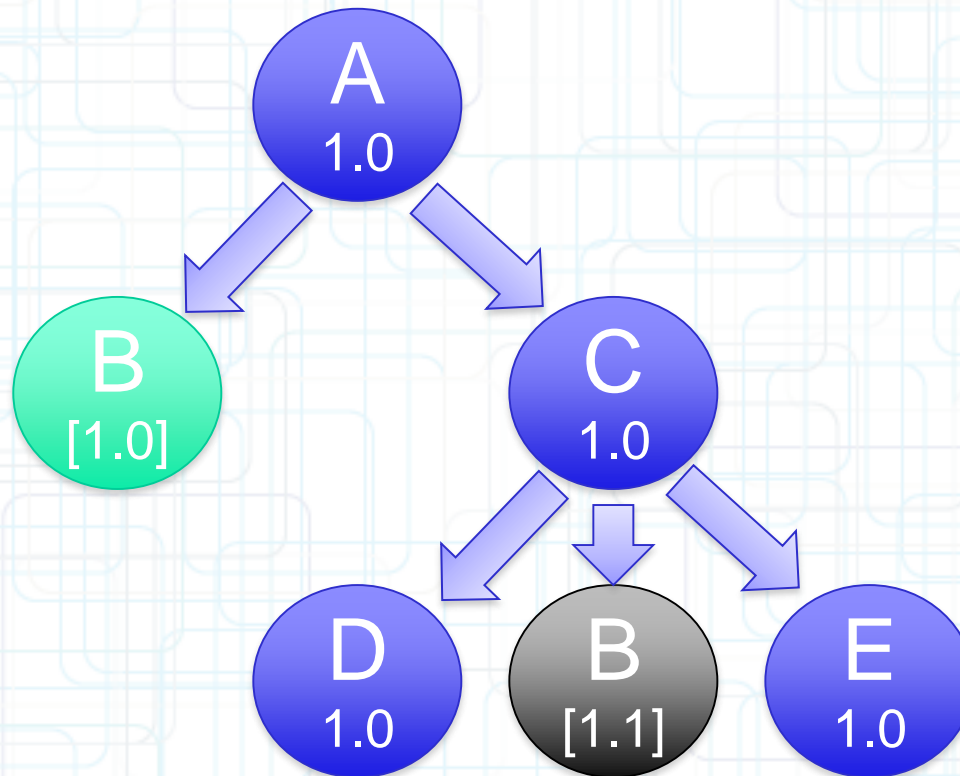
www.scmGalaxy.com

Transitive Dependencies



Now, let's say project C has a dependency on project B, but requires version 1.1. If project A's POM doesn't explicitly require version 1.0 or earlier, then Maven will choose version 1.1.

Transitive Dependencies



Uh oh. Now Project A is saying that it must use version 1.0 of B, and only version 1.0, and project C needs version 1.1 of project B.

Dependency Exclusions

One way to deal with conflicts is with exclusions

```
<dependencies>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>project-b</artifactId>
    <version>[1.0]</version>
  </dependency>
  <dependency>
    <groupId>com.techmaine</groupId>
    <artifactId>project-c</artifactId>
    <exclusions>
      <exclusion>
        <groupId>com.techmaine</groupId>
        <artifactId>project-b</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

Dependency Management

Parent POM

```
<dependencyManagement>  
  <dependencies>  
    <dependency>  
      <groupId>org.springframework</groupId>  
      <artifactId>spring</artifactId>  
      <version>2.5.5</version>  
    </dependency>  
  </dependencies>  
</dependencyManagement>
```

Child POM

```
<dependencies>  
  <dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring</artifactId>  
  </dependency>  
</dependencies>
```

The `dependencyManagement` element allows you to specify version numbers of dependencies in child POMs without making all children dependent on a particular library.

SNAPSHOT Versions

- **SNAPSHOT is a literal string appended to a version number, e.g., 1.2.3-SNAPSHOT**
- **Indicates that a version is “under development”**
- **Use if you need Maven to keep checking for the latest version**
- **Maven replaces SNAPSHOT with a UTC time stamp before putting it into the repository.**

Build Configuration

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <version>1.0.0</version>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Build Configuration

The build section of the POM, broken down further:

```
<project>
...
<build>
  <filters>
    <filter>filter/my.properties</filter>
  </filters>
  <resources>
    ...
  </resources>
  <plugins>
    ...
  </plugins>
</build>
...
</project>
```

Build Configuration Filters

```
<project>
...
<build>
  <filters>
    <filter>filter/my.properties</filter>
  </filters>
  <resources>
    ...
  </resources>
  <plugins>
    ...
  </plugins>
</build>
...
</project>
```

Path to a properties file (name=value).
When the resources are processed during packaging, maven will substitute any `{name}` strings with the corresponding value from the properties file.

Build Configuration Resources

- **The resources:resources goal copies files from the resources directory to the output directory**
- **Can process using filters**
- **Default location src/main/resources**
- **Can be further configured:**

```
<resources>
  <resource>
    <directory>src/main/scripts</directory>
    <filtering>>true</filtering>
    <targetPath>bin</targetPath>
    <includes>
      <include>run.bat</include>
      <include>run.sh</include>
    </includes>
  </resource>
</resources>
```

Build Configuration Plugins

- **All work in Maven is performed by plugins**
- **Like Dependencies, are Downloaded from a repository**
- **Because they are shared, you often benefit from the fact that someone else has already built a plugin for whatever function you may need**

Plugin Configuration

The plugin section of the POM has a configuration element where you can customize plugin behavior:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <!-- Configuration details go here -->
      </configuration>
    </plugin>
  </plugins>
</build>
```

Plugin Configuration: Example

Below, we have configured the clean plugin to delete files ending in .txt from the tmp directory

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <filesets>
          <fileset>
            <directory>tmp</directory>
            <includes>
              <include>**/*.txt</include>
            </includes>
            <followSymlinks>>false</followSymlinks>
          </fileset>
        </filesets>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Core Plugins



Maven
Plugin
Whirlwind
Tour

- **clean** - has only one goal, clean. Deletes the target directory, can be configured to delete other stuff
- **compiler** - compiles sources, uses javac compiler by default.
 - Has a compile and testCompile goal.
 - Can be configured to use any executable as the compiler
- **deploy** - uploads artifacts to a remote repository

Core Plugins, cont.



Maven
Plugin
Whirlwind
Tour

- **install** - installs the artifact into the local repository.
 - install goal, install this project's artifact
 - install-file goal, install a specific file into local repo (good for third-party stuff)
- **surefire** - runs all of the unit tests in the test source directory, and generates reports.
- **resources** - copies resources to be packaged

Packaging Plugins



Maven
Plugin
Whirlwind
Tour

- **ear, ejb, jar, war**
- **assembly** - builds a binary distribution including runtime dependencies
 - supports zip, tar.gz, tar.bz2, jar, dir, and war formats
 - uses “assembly descriptors” to configure (although several pre-fab ones are available)
 - one of the pre-fab descriptors builds executable jar files with all dependencies embedded

Utility Plugins



Maven
Plugin
Whirlwind
Tour

- **archetype** - builds skeleton of a working project for many different frameworks
 - Wicket, Tapestry 5, JSF, JPA, tons of others
- **help** - even the help is a plugin! Use the describe goal to learn what a plugin can do, e.g.,

```
mvn help:describe -Dplugin=compiler
```

- **scm** - source control stuff

Build Lifecycle

- **Usually, an artifact is built by executing a sequence of goals**
- **For example, to generate a WAR:**
 - **Clean the build area**
 - **Copy the resources**
 - **Compile the code**
 - **Copy the test resources**
 - **Compile the test code**
 - **Run the test**
 - **Package the result**

Maven's Lifecycles

Maven supports three standard lifecycles

- **clean** - as you might expect, starts us fresh
- **default** - the lifecycle that builds the code
- **site** - a lifecycle for building other related artifacts (e.g., reports and documentation)

Clean Lifecycle

The Clean Lifecycle has three phases:

- **pre-clean**
- **clean**
- **post-clean**

Only clean is “bound” by default, to the clean goal of the clean plugin. You can bind other tasks using executions.

Executions

Let's say you have a whizz-bang plugin named *mp3*, and it has a goal named *play* that lets you play an arbitrary audio clip, and you'd like to play a clip during pre-clean:

```
<plugin>
  <groupId>com.techmaine</groupId>
  <artifactId>mp3</artifactId>
  <version>1.0</version>
  <executions>
    <execution>
      <phase>pre-clean</phase>
      <goals>
        <goal>play</goal>
      </goals>
      <configuration>
        <audioClipFile>toilet-flush.mp3</audioClipFile>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Maven's Default Lifecycle

Maven models the software build process with the 21 step “default lifecycle”

validate	generate-test-sources	package
generate-sources	process-test-sources	pre-integration-test
process-sources	generate-test-resources	integration-test
generate-resources	process-test-resources	post-integration-test
process-resources	test-compile	verify
compile	test	install
process-classes	prepare-package	deploy

Package-Specific Lifecycles

Maven automatically binds goals to the phases on the previous slide based on the packaging type. E.g., for projects that package WARs:

Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

Profiles

Profiles allows alternative build executions within the same pom

Different <modules/>, <dependencies/>, etc.

Different plugin configuration, different resources

Inherited from parent

Activation

Explicit: \$ mvn -Pprofile1,profile2

Activated by properties

Activated by settings.xml

Could be used in combination with filtering

To have different artifacts for different usages

Use different classifier in resulting artifact

Should not be used to escalate from DEV->TEST->PROD

www.scmGalaxy.com

Build Profiles

```
<project xmlns=http://maven.apache.org/POM/4.0.0 >
  <modelVersion>4.0.0</modelVersion>
  <name>Super Duper Amazing Deluxe Project</name>
  <packaging>jar</packaging>
  <groupId>com.techmaine</groupId>
  <artifactId>superduper</artifactId>
  <version>1.0.0</version>
  <parent>
    <!-- Parent POM stuff if applicable -->
  </parent>
  <modules>
    <!-- Sub-modules of this project -->
  </modules>
  <properties>
    <!-- Ad-hoc properties used in the build -->
  </properties>
  <dependencies>
    <!-- Dependency Stuff -->
  </dependencies>
  <build>
    <!-- Build Configuration -->
    <plugins>
      <!-- plugin configuration -->
    </plugins>
  </build>
  <profiles>
    <!-- build profiles -->
  </profiles>
</project>
```

Profiles: Customized Builds

Sometimes our artifacts need to be tweaked for different “customers”

- **The Development version has different logging or database configuration than QA or Production**
- **There might be slight differences based on target OS or JDK version**

How to declare a profile

In the POM itself, in an external profiles.xml file, or even in settings.xml

```
<project>
  ...
  <profiles>
    <profile>
      <id>appserverConfig-dev</id>
      <properties>
        <appserver.home>/path/to/dev/appserver</appserver.home>
      </properties>
    </profile>

    <profile>
      <id>appserverConfig-dev-2</id>
      <properties>
        <appserver.home>/path/to/another/dev/appserver2</appserver.home>
      </properties>
    </profile>
  </profiles>
  ...
</project>
```

Build Configuration in a Profile

You can even configure plugins based on a profile:

```
<project>
  ...
  <profiles>
    <profile>
      <id>production</id>
      <build>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <debug>>false</debug>
              <optimize>>true</optimize>
            </configuration>
          </plugin>
        </plugins>
        <appserver.home>/path/to/dev/appserver</appserver.home>
      </build>
    </profile>
  ...
```

Activating a Profile

- **On the command-line:**

```
mvn package -Pmyprofile1,myprofile2
```

- **In your settings.xml file:**

```
<settings>
```

```
...
```

```
<activeProfiles>
```

```
<activeProfile>dev</activeProfile>
```

```
</activeProfiles>
```

```
...
```

```
</settings>
```

- **Activation elements**

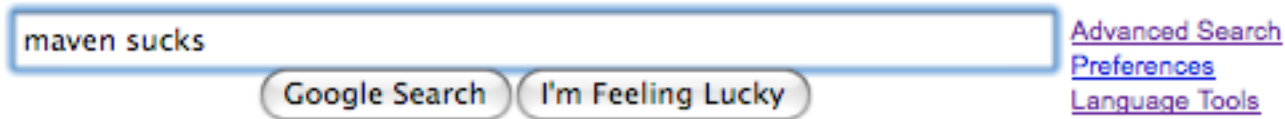
Activation Elements

```
<project>
...
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>2.0.9</value>
        </property>
        <file>
          <exists>file2.properties</exists>
          <missing>file1.properties</missing>
        </file>
      </activation>
    ...
  </profile>
</profiles>
</project>
```

www.scmGalaxy.com

Example from *Maven: The Definitive Guide*, Sonatype, O'Reilly p.204-205

Sometimes Maven Sucks



...returns “about 128,000” results

Common Criticisms

- **Poor Documentation - Lots of Maven's online documentation is automatically generated and is generally pretty horrible**
- **Simple things are sometimes counterintuitive with Maven - E.g., copying a file**
- **Maven adds to the number of places you need to look when something breaks - both your source repository, and the maven repository**
- **Everything breaks if someone changes an artifactId or groupId**
- **Doesn't work well if your network connectivity is unreliable or unavailable**
- **Gets tangled and confused if one of your transitive dependencies isn't available in a maven repository**

Maven reactor

Maven builds recursively, using nested <modules/>

Can be tweaked by maven parameters

\$ mvn -N -> non-recursive

\$ mvn -pl m1, m2,... -> project list

\$ mvn -pl :m1 -> by artifact id instead of directory name

\$ mvn -pl m1 -am -> also make requirements

\$ mvn -pl m1 -amd -> also make dependents

\$ mvn -rf m1 -> resume from

(Q)