# Introduction to Maven

Darryl J. Parks

Aug 12, 2010

# Agenda

- What is Maven?
- Benefits of Maven
- Comparison with Ant
- Default Directory structure
- POM, Super POM, Effective POM
- Lifecycle, Goals, Artifacts, Phases
- Plugins
- Properties
- Dependency
- Transitive Dependency
- Multi-module Projects
- Repository
- Profiles
- Snapshots
- Releases
- Lessons Learned

# What is Maven?

- Build management tool
- Dependency manager
- Software project management and comprehension tool
- Project organizer/simplifier
- Documentation manager
- Release/collaboration coordinator

# Why Maven?

- It is a way to manage and maintain projects consistently
- Sharing common code
  - Version isolation
  - JDK flexibility
- Makes the build process easy
- Provides a uniform build system
- Provides quality project information
- Provides Coherence, Reusability, Agility, Maintainability

# Key Features

- Convention over configuration
- Dependency Management
- Easy to write plug-ins in Java, ANT, or scripting languages
- Integrates with SCMs and manages the release of a project
- Large central repository of JARs and other dependencies

# Comparison To ANT

- Maven's compiler plugin is generally used without any configuration
- Ant's Javac task currently has 33 options, we typically use 6
- Ant has 9 example forms of usage, Maven has one pattern
- Even with macros, its use can only be standardized locally

# ANT Recap…

- Ant Pros
  - Will do what you ask it to do
  - Has a large set of “configurable” tasks that you can use
  - Extremely flexible, configurable, and powerful
- Ant Cons
  - You have to specify everything
  - No convention, only configuration (and lots of it!!)
  - Builds can vary across projects even for same types of artifacts
  - Build lifecycle is what you define and can vary between projects
  - Is XML really a great format for coding?

# Maven Comparison…

- Maven Pros
  - Defines a process for the build
  - Enforces best practices
  - Creates standardization and consistency in builds
  - Excellent dependency management
  - Eliminates redundancy (one particular version of a jar file on your machine - no matter how many projects use it! )
  - IDEs instantly aware of your application (No Configuration, IntelliJ Idea)
- Maven Cons
  - Requires an enterprise wide repository setup (Nexus) ??Really?? Repo1.maven.org
  - Initial learning curve, new vocabulary (phases, goals, artifacts)

# Why Not Just Use ANT?

- Ant provides no conventions
- Ant provides no models
- Ant provides no processes
- Ant provides no coherence
- Ant provides no native dependency management
- Maven goes to great lengths to make Ant use as simple as possible. We have the Ant Run plugin and Ant based Plugins.
- "Inner-Platform Effect" anti-pattern

# A quick comparison…

Another way to look at it…

**Ant**

(simple and unassuming)

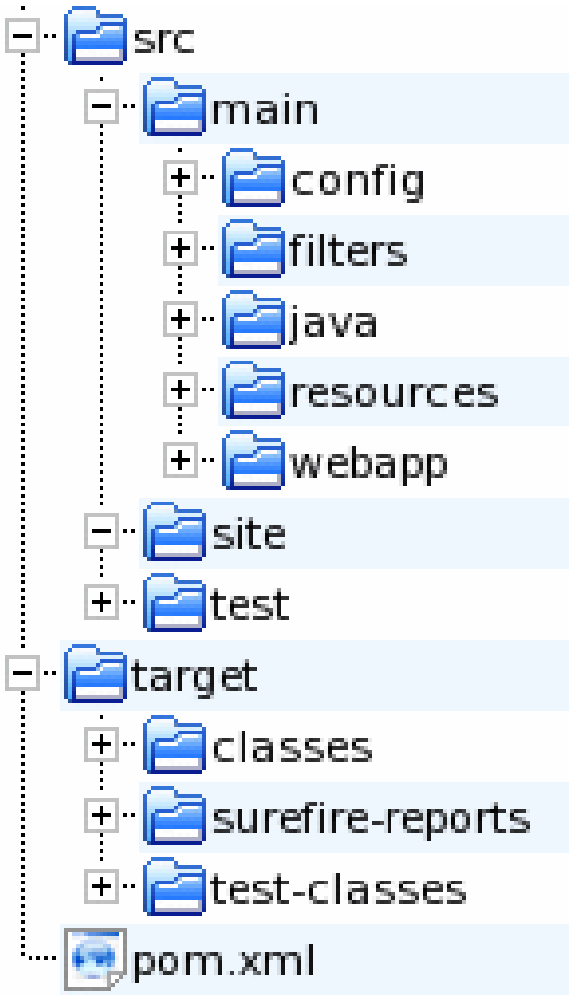
**Maven**

(opinionated and robust)

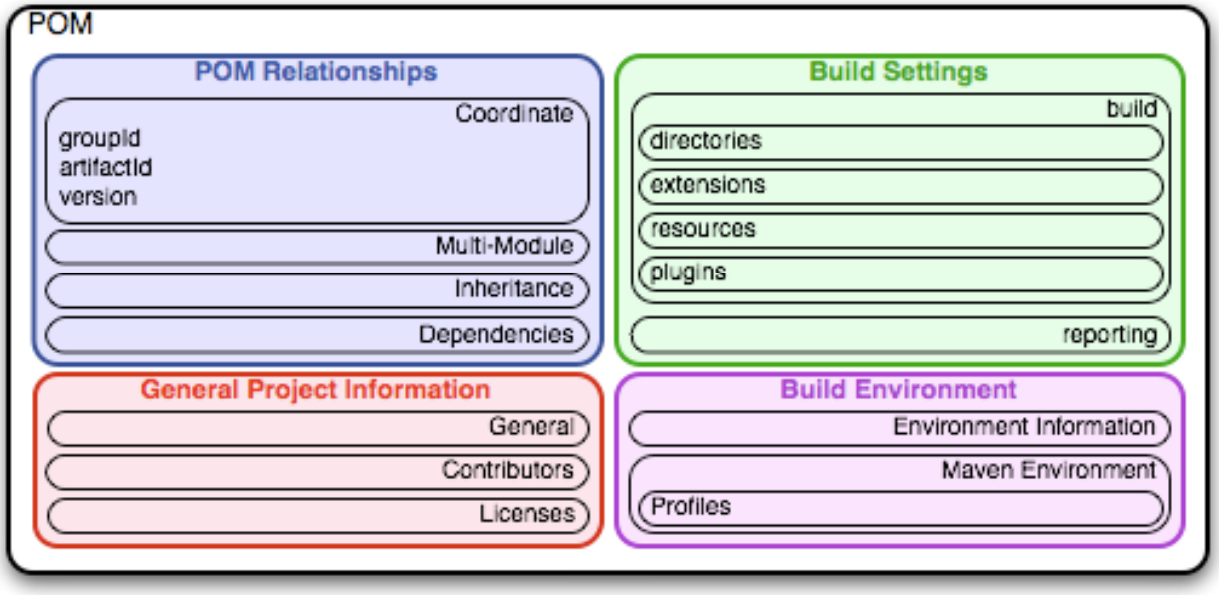
# Core concepts

- Project Directory Structure
- POM
  - Super POM
  - Corporate POM
  - Effective POM
- Artifact
- Lifecycle
- Phases
- Goals (MOJO)
- Plugins

# Project Directory Structure



# POM – Project Object Model

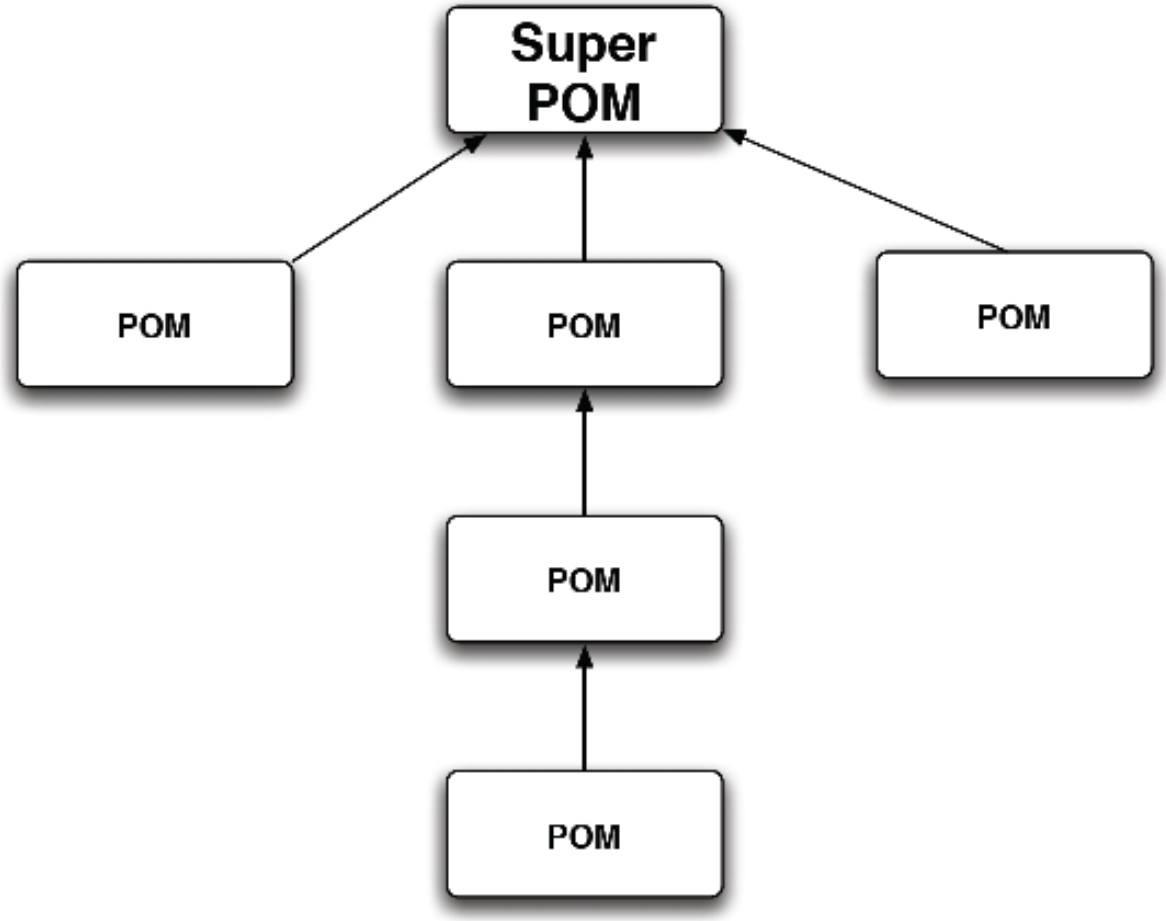



- XML file that represents the project conceptually as an object
- POM represents the project in a declarative manner
- One POM describes one artifact (Jar, WAR)

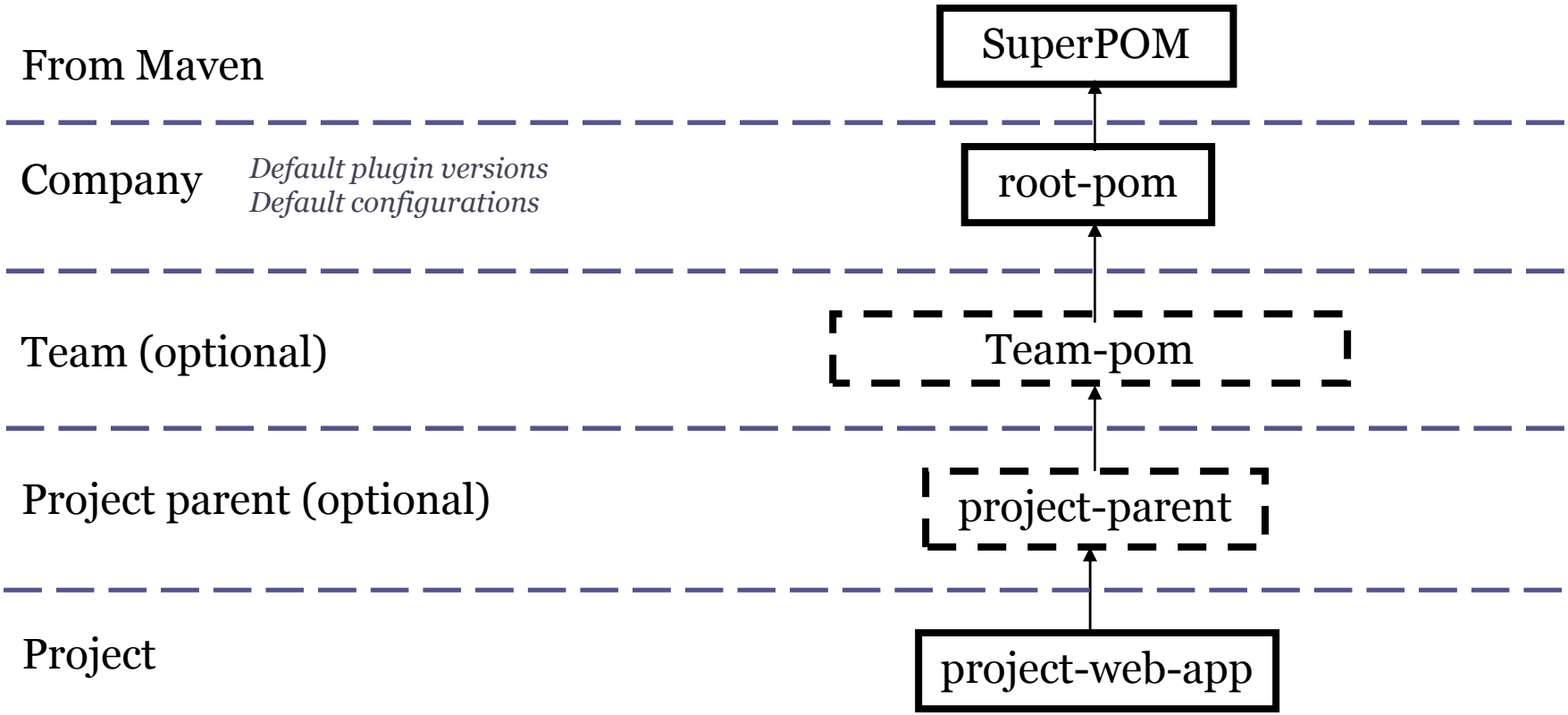
# Project Object Model (POM)

- Description of everything that relates to the software project
- Fundamental entity in Maven. Everything in Maven relates to POM in some way
- Follows an inheritance model that is similar to Java itself
  - Super POM which is similar to java.lang.Object
- Corresponds to one artifact (JAR, WAR, EAR) with a unique coordinate made up of the groupId, artifactId, & version

# Super POM – Mother of all POMs



# POM Hierarchy



# Example POM

```
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.djparks.killerapp</groupId>
   <artifactId>superapp</artifactId>
   <version>1.0-SNAPSHOT</version>
   <name>Maven Super Killer App</name>

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

# Effective POM

- Combination of Super POM, plus any parent POMs, and finally the current project's POM
- POMs down the hierarchy override the parent configuration
- Just like inheritance!
- **mvn help:effective-pom**

# Effective POM

```
<?xml version="1.0"?><project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com..killera
  <artifactId>superapp<
  <name>Maven Super Kil
  <version>1.0-SNAPSHOT
  <description>TPS pom<
  <url>http://w3.maven.
  <issueManagement>
   <system>TeamTrack</s
   <url>http://w3.teamt
  </issueManagement>
  <mailingLists>
   <mailingList>
    <name>General Java
    <subscribe>java-dev
    <unsubscribe>java-d
    <post>java-develope
    <archive>http://mhs
   </mailingList>
  </mailingLists>
  <developers>
   <developer>
    <id>edturc</id>
    <name>Eric Turcotte
    <email>eric.d.turco
    <organization>TPS:
    <roles>
      <role>Developer</
    </roles>
    <timezone>-6</timez
   </developer>
  </developers>
```

```
<scm>
    <connection>scm:
    Poms/tps-pom/sup
    <developerConnec
    re/djparkPoms/tp
</scm>
 <organization>
  <name>TPS</name>
</organization>
<build>
  <sourceDirectory>C
  <scriptSourceDirect
  <testSourceDirecto
  <outputDirectory>C
  <testOutputDirecto
 <resources>
  <resource>
    <directory>C:\cvs
  </resource>
 </resources>
 <testResources>
   <testResource>
    <directory>C:\cvs
   </testResource>
  </testResources>
  <directory>C:\cvs\
  <finalName>superap
  <pluginManagement>
   <plugins>
    <plugin>
      <artifactId>ma
      <version>2.2-a
```

```
<executions>
  <execution>
   <id>package-test
    <phase>package<
    <goals>
     <goal>test-jar
    </goals>
   </execution>
  </executions>
 </plugin>
 <plugin>
  <artifactId>maven
  <version>2.0.2</v
  <configuration>
   <source>1.5</sou
   <target>1.5</tar
  <meminitial>128</
 <maxmem>512</maxme
 <showDeprecation>f
 <executable>C:\jdk
 <compilerVersion>1
 <fork>true</fork>
 </configuration>
</plugin>
<plugin>
 <artifactId>maven-s
 <version>2.0-beta-5
</plugin>
<plugin>
 <artifactId>maven-s
 <version>2.3</versi
 <executions>
  <execution>
```

```
<executions>
  <execution>
  <id>package-test-jar</id>
  <phase>package</phase>
 <goals>
   <goal>test-jar</goal>
  </goals>
</execution>
  </executions>
 </plugin>
 <plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.0.2</version>
  <configuration>
<source>1.5</source>
<target>1.5</target>
<meminitial>128</meminitial>
<maxmem>512</maxmem>
<showDeprecation>false</showDeprecation>
<executable>C:\jdk1.5.0/bin/javac</executable>
<compilerVersion>1.5</compilerVersion>
<fork>true</fork>
  </configuration>
 </plugin>
 <plugin>
  <artifactId>maven-site-plugin</artifactId>
  <version>2.0-beta-5</version>
 </plugin>
 <plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.3</version>
  <executions>
   <execution>
```

Page 4 of 10

# The simplest POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.djparks</groupId>
  <artifactId>simplest-project</artifactId>
  <version>1</version>
</project>
```

- Running **mvn package** would compile all classes in **src/main/java**
- Produce jar in **target/simple-project-1.jar**

# Structure and Design

- Standardize Maven project structures
  - /pom.xml
  - /src/main/java
  - /src/main/resources
  - /src/test/java
  - /src/test/resources
- Standardize project POM's
  - Archetypes
  - Inheritance
  - Corporate POM

# POM Configuration (1 of 3)

```
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.djparks.killerapp</groupId>
   <artifactId>marker-picking</artifactId>
   <packaging>pom</packaging>
   <version>1.1.0-SNAPSHOT</version>
   <name>Marker picking</name>
   <description>Marker picking</description>
   <inceptionYear>2007</inceptionYear>

   <organization>
      <name>Parksy's Killer Applications</name>
   </organization>
```

# POM Configuration (2 of 3)

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

<scm>

  <connection>scm:cvs:pserver:cvs01:/rep:btmp</connection>

  <developerConnection>scm:cvs:pserver:cvs01:/rep:btmp</deve
  loperConnection>
</scm>
```

# POM Configuration (3 of 3)

```
<plugins>
   <!-- copy the WAR to the Tomcat webapps folder -->
   <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-antrun-plugin</artifactId>
       <executions>
         <execution>
           <id>copy-war-to-tomcat</id>
           <phase>install</phase>
           <goals>
             <goal>run</goal>
           </goals>
           <configuration>
             <tasks>
               <copy
 file="${project.build.directory}/${project.build.finalName}.war"
 tofile="${default.tomcat.home}/webapps/${project.build.finalName}.war"
 failonerror="false" />
```

# Dependencies

- Scopes
  - **compile** - this is the default scope, used if none is specified. Compile dependencies are available in all classpaths.
  - **provided** - this is much like compile, but indicates you expect the JDK or a container to provide it. It is only available on the compilation classpath, and is not transitive.
  - **runtime** - this scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
  - **test** - this scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.
  - **system** - this scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

# Dependencies (Exclusions)

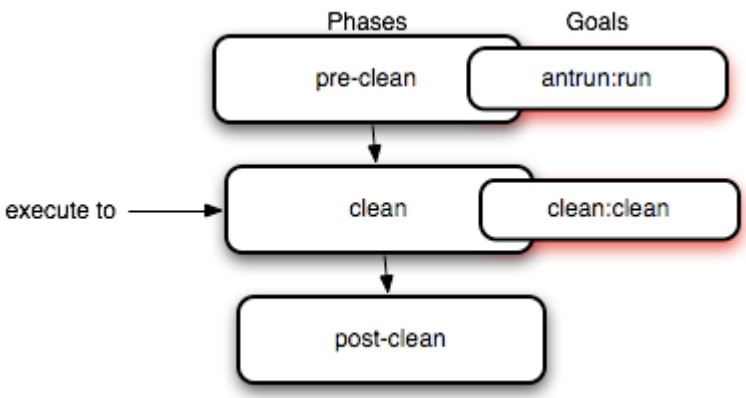
```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>group-a</groupId>
            <artifactId>artifact-a</artifactId>
            <version>1.0</version>
            <exclusions>
                <exclusion>
                    <groupId>group-c</groupId>
                    <artifactId>excluded-artifact</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>group-a</groupId>
            <artifactId>artifact-b</artifactId>
            <version>1.0</version>
            <type>bar</type>
            <scope>runtime</scope>
         </dependency>
     </dependencies>
</project>
```

# Build Lifecycle Explained

- Well-defined list of actions (phases)
- Improvement over ANT
  - Innumerable combinations of tasks
  - Target names change from project to project
- Organized sequence of phases
- Three lifecycles: clean, default, site

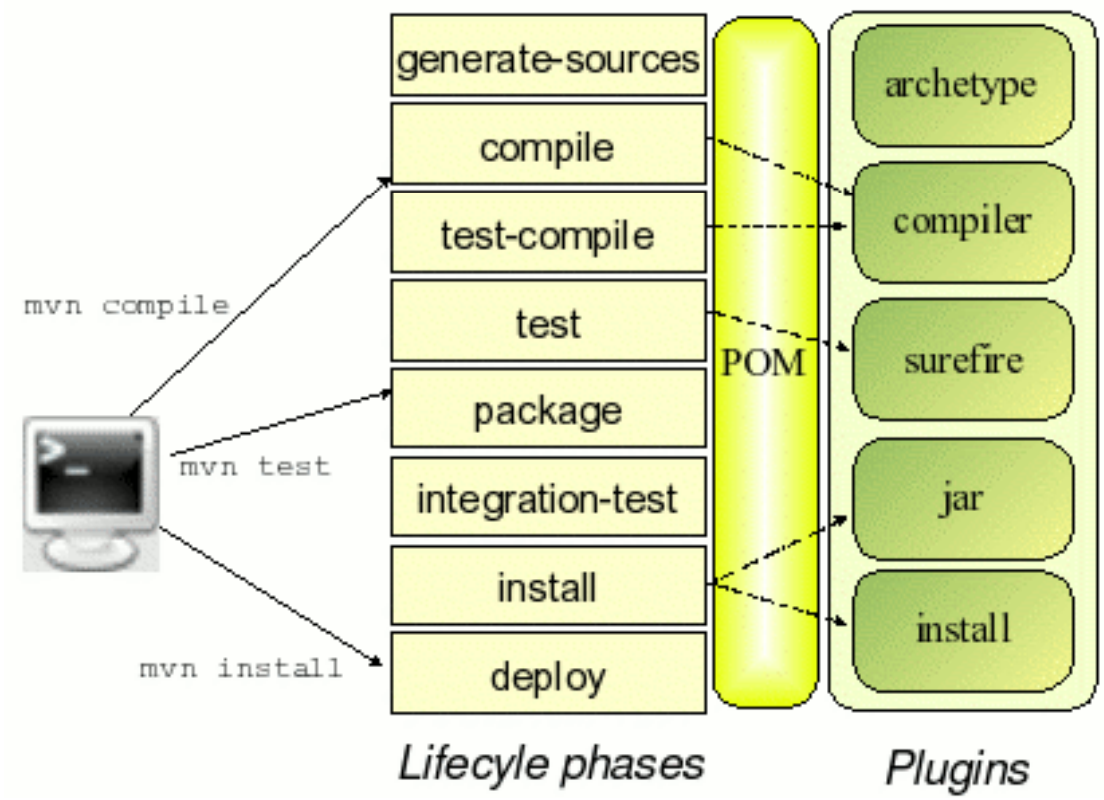
# Terminology

- Lifecycle – sequential list of phases
- Phase – lifecycle action that executes goals
  - compile, test-compile, test, package, install
- Goal – plugin task that is bound to phases
  - clean:clean, compiler:testCompile, compiler:compile, jar:jar, surefire:test
- Plugin – grouping of goals with similar functionality
  - maven-clean-plugin
  - maven-compiler-plugin
  - tomcat-maven-plugin



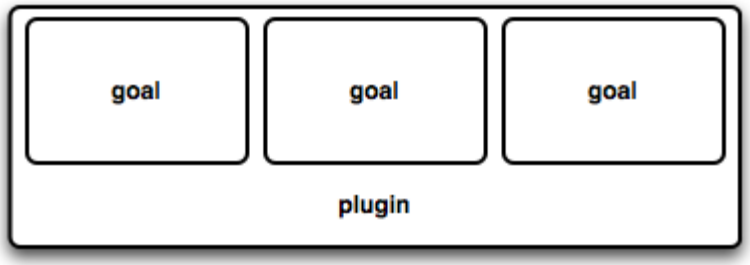
# Lifecyle, Phase, Goals

- A phase is a step in what Maven calls the "build lifecycle”
- compile, package, install are all phases



# Goal

- Something you want to achieve
- Analogous to “target” in Ant
  - Things you want to be done on a project
  - Executed stand-alone or along with other goals as part of a larger build
  - compile, deploy, release
- Arguments can be passed using –Dname=value pairs
- A goal is always a part of a plugin
- Each goal has acces



# The build lifecycle

- Each lifecycle consists of several phases
- Executing a phase will execute all phases from the first phase to that phase in order
- There are three built-in lifecycles:
  - default – project deployment
  - clean – project cleaning
  - site – creation of project's site documentation

# The clean lifecycle

- clean lifecycle uses **clean** plugin
- Has 3 phases
  - pre-clean
    - executes processes needed prior to the actual project cleaning
  - clean
    - remove all files generated by the previous build
  - post-clean
    - executes processes needed to finalize the project cleaning
- mvn clean
- mvn clean:clean

# The default lifecycle

**Default Lifecycle**

| | |
|---|---|
| validate | validate the project is correct and all necessary information is available. |
| generate-sources | generate any source code for inclusion in compilation. |
| process-sources | process the source code, for example to filter any values. |
| generate-resources | generate resources for inclusion in the package. |
| process-resources | copy and process the resources into the destination directory, ready for packaging. |
| compile | compile the source code of the project. |
| process-classes | post-process the generated files from compilation, for example to do bytecode enhancement on Java classes. |
| generate-test-sources | generate any test source code for inclusion in compilation. |
| process-test-sources | process the test source code, for example to filter any values. |
| generate-test-resources | create resources for testing. |
| process-test-resources | copy and process the resources into the test destination directory. |
| test-compile | compile the test source code into the test destination directory |
| test | run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed. |
| prepare-package | perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above) |
| package | take the compiled code and package it in its distributable format, such as a JAR. |
| pre-integration-test | perform actions required before integration tests are executed. This may involve things such as setting up the required environment. |
| integration-test | process and deploy the package if necessary into an environment where integration tests can be run. |
| post-integration-test | perform actions required after integration tests have been executed. This may including cleaning up the environment. |
| verify | run any checks to verify the package is valid and meets quality criteria. |
| install | install the package into the local repository, for use as a dependency in other projects locally. |
| deploy | done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects. |

# Default Lifecycle Phases

- **validate** - validate the project is correct and all necessary information is available
- **generate-sources** - generate any source code for inclusion in compilation
- **process-sources** - process the source code, for example to filter any values
- **generate-resources** - generate resources for inclusion in the package
- **process-resources** - copy and process the resources into the destination directory, ready for packaging
- **compile** - compile the source code of the project
- **process-classes** - post-process the generated files from compilation, for example to do bytecode enhancement on Java classes
- **generate-test-sources** - generate any test source code for inclusion in compilation
- **process-test-sources** - process the test source code, for example to filter any values
- **generate-test-resources** - create resources for testing
- **process-test-resources** - copy and process the resources into the test destination directory
- **test-compile** - compile the test source code into the test destination directory

# Default Lifecycle Phases (cont.)

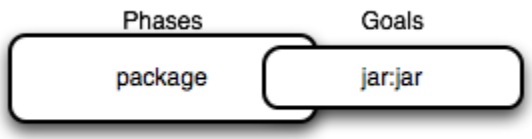
- **test** - run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **prepare-package** - perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package (coming in Maven 2.1+)
- **package** - take the compiled code and package it in its distributable format, such as a JAR
- **pre-integration-test** - perform actions required before integration tests are executed. This may involve things such as setting up the required environment
- **integration-test** - process and deploy the package if necessary into an environment where integration tests can be run
- **post-integration-test** - perform actions required after integration tests have been executed. This may including cleaning up the environment
- **verify** - run any checks to verify the package is valid and meets quality criteria
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects

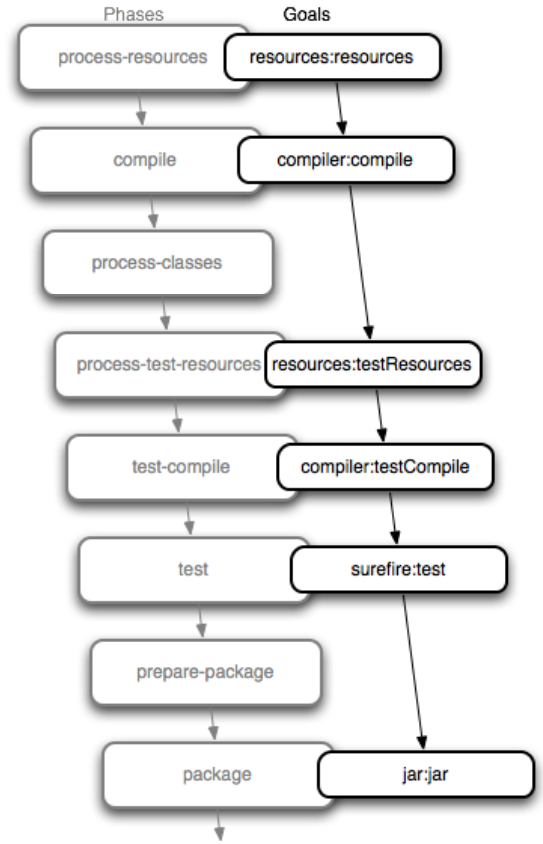
# The site lifecycle

- pre-site
  - executes processes needed prior to the actual project site generation
- site
  - generates the project's site documentation
- post-site
  - executes processes needed to finalize the site generation, and to prepare for site deployment
- site-deploy
  - deploys the generated site documentation to the specified web server

# Summary of the build lifecycle

- An ordered sequence of phases involved in building a pro
- clean, default, site
- **validate -> compile -> test -> package -> install ->**
- Some phases are named vaguely
  - Example: package may produce a jar or war
- Plug-in goals can be attached to a lifecycle phase




Note: There are more phases than shown above, this is a partial list al list

# Plugins

- Large number of plugins
- Maven figures out how to use it
- Will download the plugin if not there!
- Customize plugins that ship with Maven
- Add new plug-ins to your build
- Can bind a goal to a lifecycle phase

# Plugins…

## Examples

- **Jar** plugin
  - contains goals for creating JAR files
- **Compiler** plugin
  - contains goals for compiling source code and unit tests
- **Surefire** plugin
  - contains goals for executing unit tests and generating reports
- Specialized plugins – **Hibernate3**
  - Integration with Hibernate

# Plugins…

- Specialized plugins – **JRuby**
  - execute Ruby as part of a Maven build
  - write Maven plugins in Ruby
- Specialized plugins – **IDEA**
  - Will generate Intellij files (ipr, iml, iws)
  - Need pom.xml file first
- Custom Plugins
  - Written in Java, Ant, Ruby, Groovy, and beanshell

# Plugins – configure existing

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

# Fun with plugins

- archetype
  - **mvn archetype:create -DgroupId=org.djpark -DartifactId=lunchlearn -DpackageName=com.djpark.lunchlearn**
- idea
  - **mvn idea:idea**
- eclipse
  - **mvn eclipse:eclipse**
- help
  - **mvn help:effective-pom**
  - **mvn help:describe -Dplugin=archetype –Dfull**
  - **mvn help:describe -Dplugin=help**

# Site Generation

- Generate project status reports and other code-related content
- Maintain consistent website navigational elements, including menu items that unfold
- Render consistent website content from a mixture of source-document formats
- Publish content in multiple formats, including XHTML and PDF
- Create portable documentation bundles, for embedding in the project's binary release archive
- Consistent, meaningful, current, easy documentation

# Report Configuration

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-changes-plugin</artifactId>
        <configuration>
          <xmlPath>src/site/changes.xml</xmlPath>
        </configuration>
...
```

# Plugins resources

- **Mojo** project at CodeHaus
  - http://mojo.codehaus.org
  - A mojo is a **M** aven plain **O** ld **J** ava **O** bject
  - Website has collection of plugins for Maven 2

- Plugin Developers Center
  - http://maven.apache.org/plugin-developers/index.html

Note: Developing custom plugins is out of scope in this presentation.

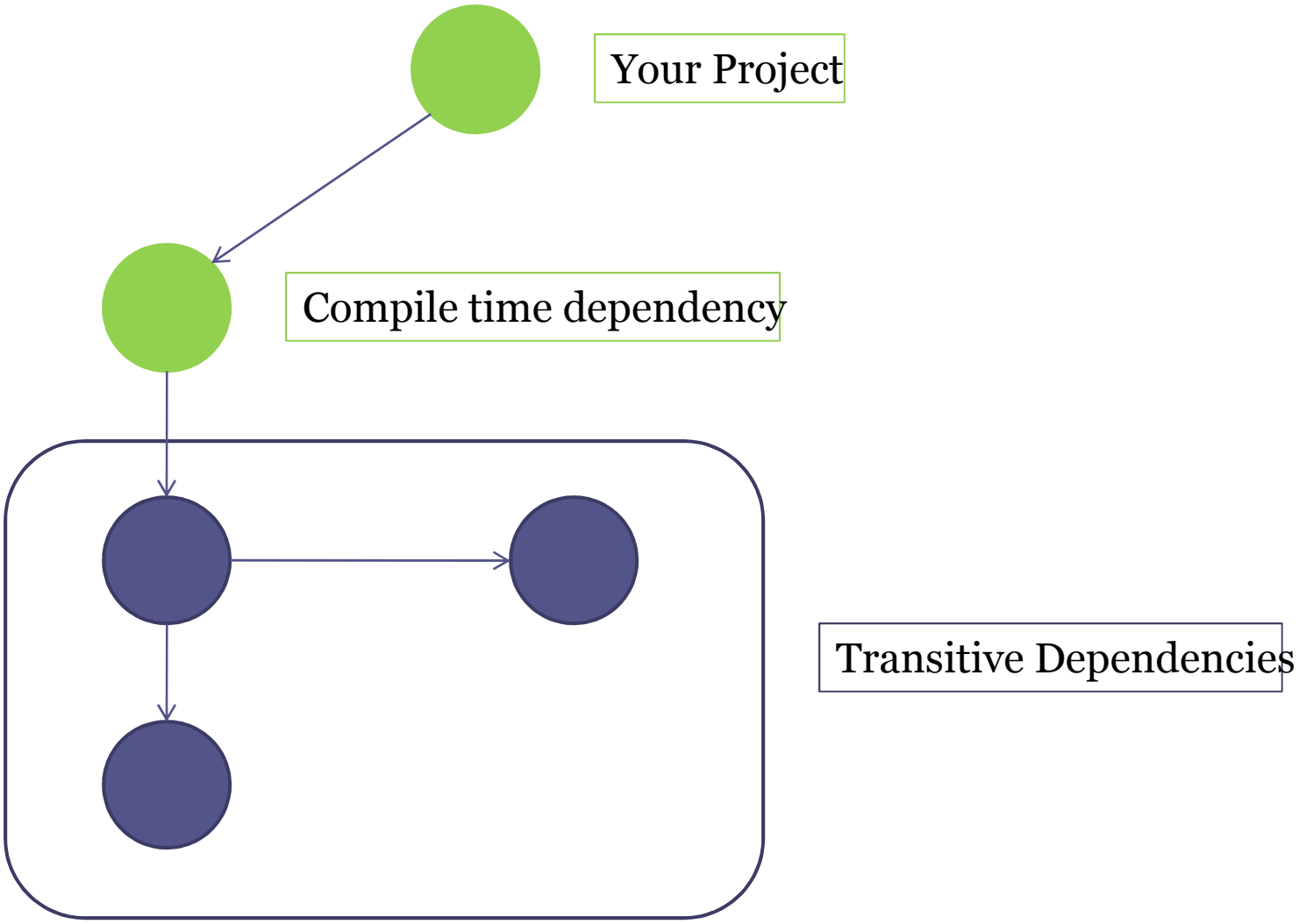
# Plugin concluding remarks

- Maven is highly customizable through plugins
- Maven build lifecycle is actually plugin goals executing in order
- New plugin goals can be bound to a phase
- Plugins can be configured through params

# Transitive Dependency

- Avoids the onus of finding all the dependencies that all the project dependencies might have and including them in the project i.e. dependencies of dependencies.
  - No limit to the number of levels that dependencies can be gathered from
- Critical feature in Maven
- New feature in 2.0

# Transitive Dependency (cont'd)



# Transitive Dependency Features

- Provides flexibility to the user through following features
  - Dependency Mediation
    - Uses “nearest definition” technique
  - Dependency Management
  - Dependency Scope
  - Excluded Dependencies
  - Optional Dependencies
    - Excluded by default

# Multi-Module Projects

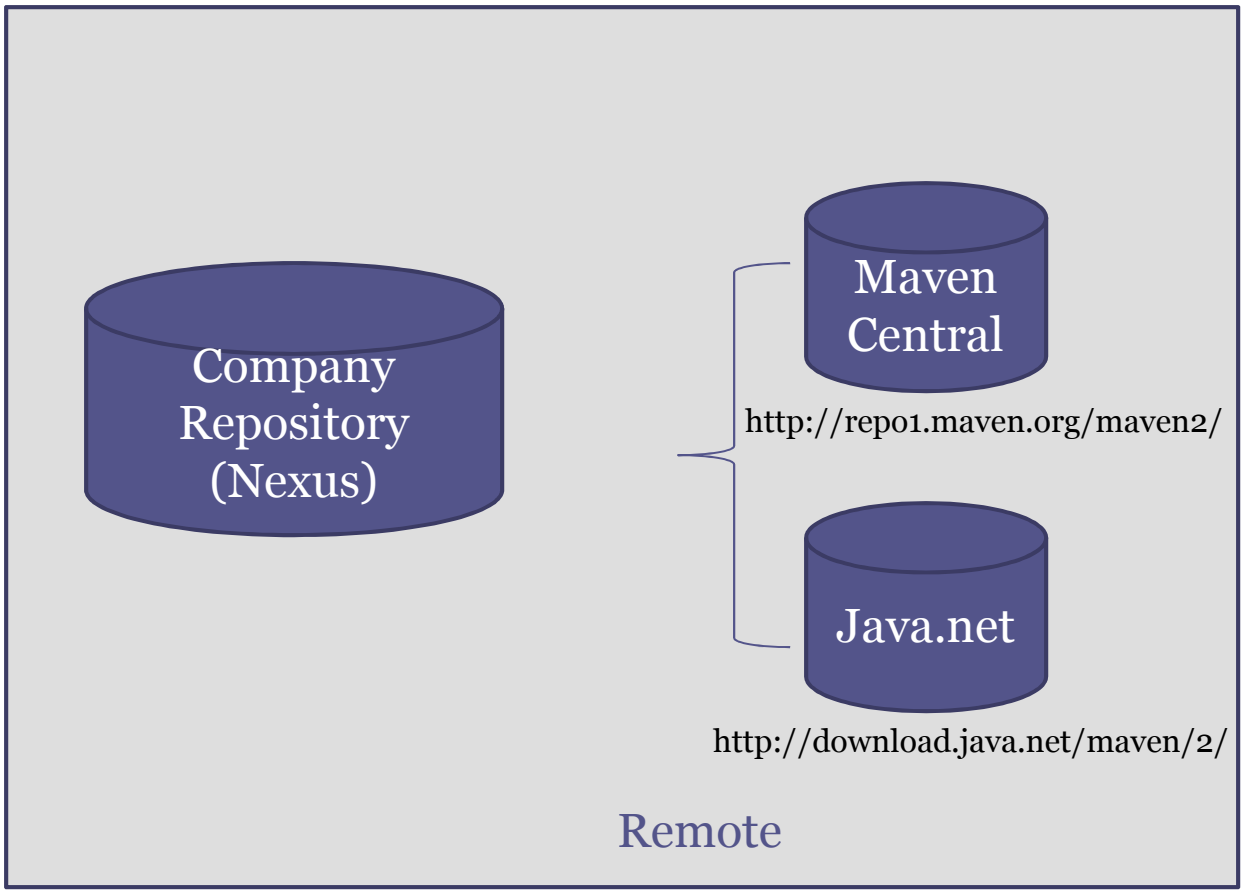
- Maven is most often used to build applications which consist of many parts.
  - In Maven a build that is made up of many components is called a multi-module project. New feature in 2.0
- No different from a single module project except multiple artifacts are being built in the case of multi-modules

# Multi-Module Projects (cont'd)

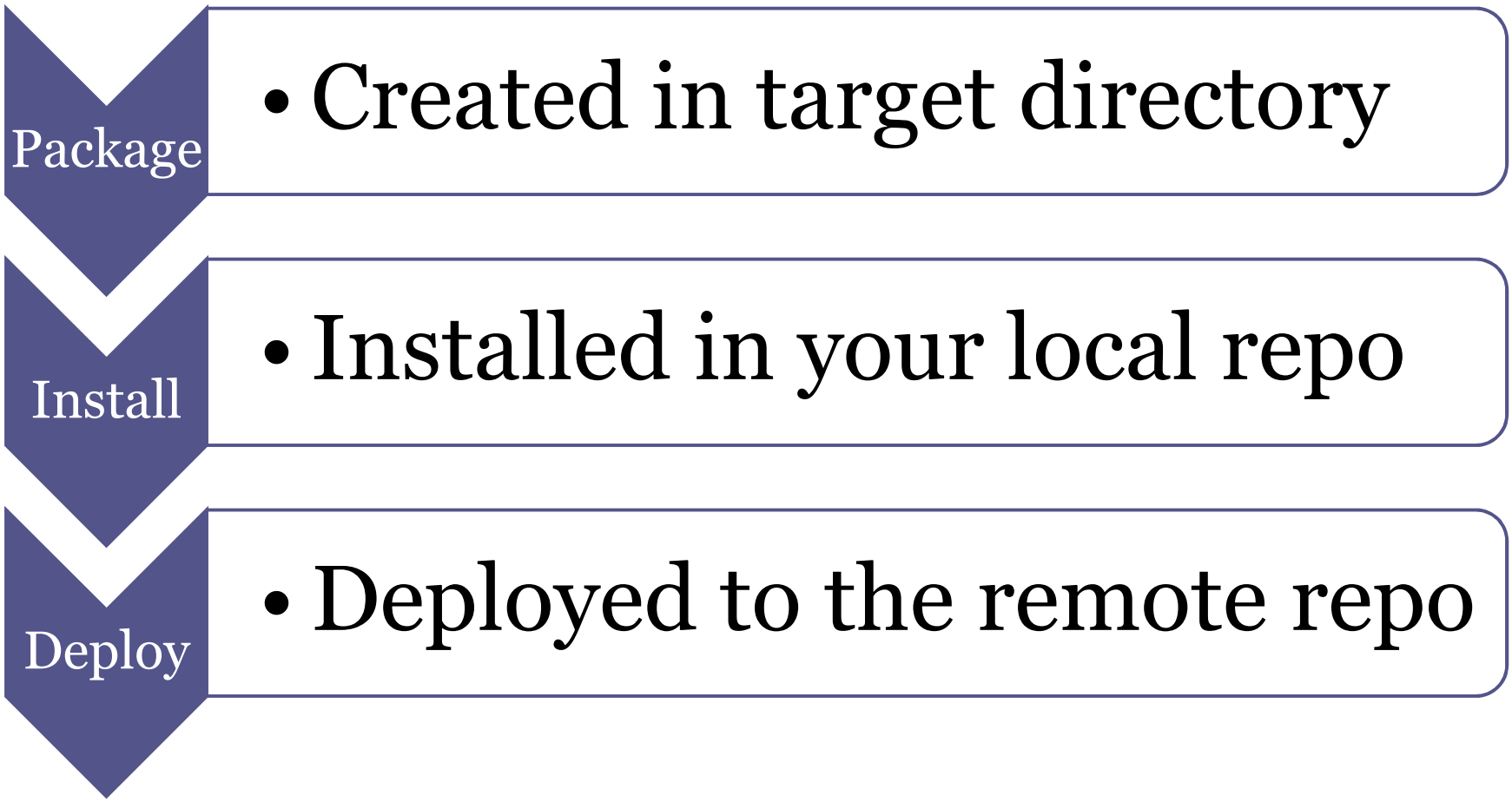
- Promotes separation of concerns
  - The ability to identify, encapsulate, and operate on the pieces of software that are relevant to a particular concept, goal, or purpose
  - This makes sure code doesn't get coupled inadvertently and inherently allows for better architecture.
  - This also makes sure you isolate your tests appropriately.
  - An example would be business logic code, web client and services client

# Maven Repository Basics

C:\Documents and Settings\<userid>\.m2



# Repository Basics for Upload



# Repository Basics for Download

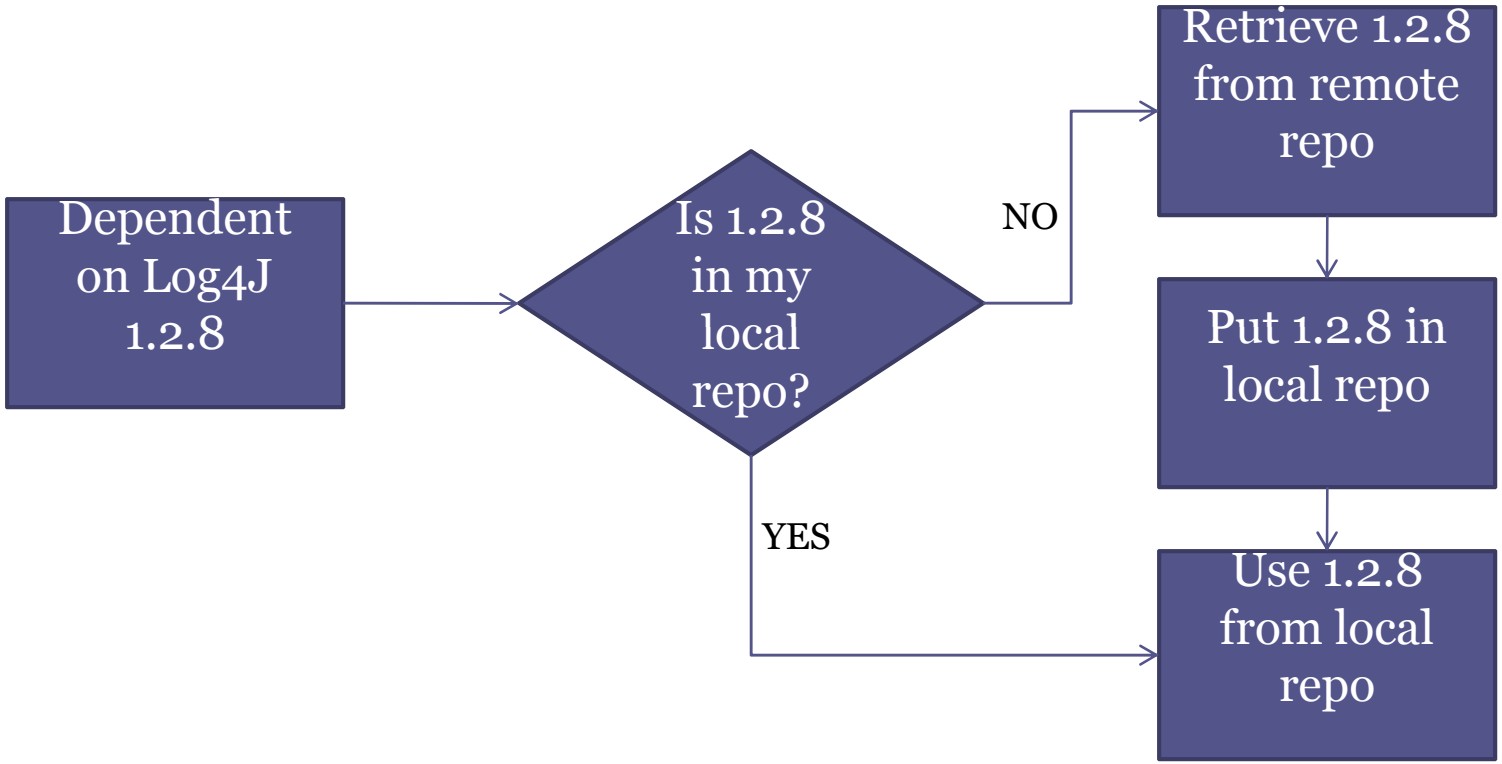
Local repository = Cache

- It is needed for efficiency/performance

How does Maven know when your cache is stale?

*VERSIONING*

# Basic Strategy – Using versions




**A version never changes.**
**Once a version, always a version.**

# Profiles

- Builds should be portable…but sometimes this isn't entirely possible
- Profiles allow you to create environmental variations in the build lifecycle
- Different build configurations for different target environments (dev, test, prod)
- Designed to be used in complementary sets to give **equivalent-but-different** parameters for a set of target environments

# Profiles in external files

- Pro: Can apply to multiple projects
- Con: Makes a project less portable
- Ability to add environmental variation to:
  - Repositories
  - Plug-in Repositories
  - Properties
- Specified in one of 3 places:
  - Per User - %USER_HOME%/.m2/settings.xml
  - Global - %M2_HOME%/conf/settings.xml
  - Profile Descriptor – profiles.xml in project base dir

# Profiles in pom.xml

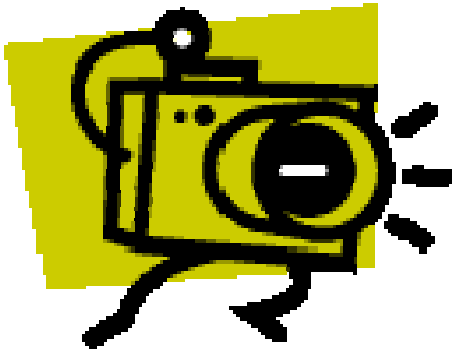
- Pro: Better chance of project portability
- Con: Cannot be shared across projects
- Ability to add environmental variation to just about anything in the POM:
  - Repositories
  - Plug-in Repositories
  - Dependencies
  - Plug-ins
  - Properties
  - Modules
  - Reporting
  - Dependency Management
  - Distribution Management
  - Build (default goal, resources, test resources, final name)

# Profile FAQs

- How do you activate a profile?
  - Command-line: mvn -PmyProfile deploy
  - Using the **activeProfiles** element in your **settings.xml** file
  - Can be triggered automatically based on JDK, OS, or existence of certain properties
- How can I tell which profiles are in effect during a build?
  - mvn help:active-profiles

# Introduction to the Release Plug-in

- Snapshots
- Building a Release



# Sophisticated Strategy – Using Snapshots

- SNAPSHOT is a designator to Maven that the artifact is in active development mode.
- A single SNAPSHOT version of an artifact may be deployed to the Maven repository ***multiple*** times.
- Maven will attempt to download the latest snapshot from a repository when you run a build

# What does this mean to you?

- The pom.xml in your local sandbox should ALWAYS be in SNAPSHOT mode.
- Your project dependencies will typically be non-SNAPSHOT versions.
- You will depend on a SNAPSHOT version when you're changing and testing integration between two projects.
- At the time of releasing your project, you should have NO SNAPSHOT dependencies

# Manual release process

1. Get the latest source code from Subversion to a clean directory
2. Ensure it builds cleanly and all tests pass
3. Ensure there are no SNAPSHOT dependencies
4. Update version in POM – remove SNAPSHOT (e.g. 1.0.0)
5. Run “mvn deploy”
6. Update version in POM – increment to next version and add SNAPSHOT (e.g. 1.0.1-SNAPSHOT)

# Maven release-plugin

- Maven's attempt to automate the manual release steps.
- Consists primarily of two phases:
  1. Prepare: Marks the point-in-time of the release
  2. Perform: Builds the actual release artifact
- Handles updating and checking in POM changes, tagging source repository, and building and deploying the release artifacts.
- Includes some enforcement (no snapshot dependencies, all tests pass, no locally edited files)

# mvn release:prepare

1. Check that there are no uncommitted changes in the sources
2. Check that there are no SNAPSHOT dependencies
3. Change the version in the POMs from x-SNAPSHOT to the release version
4. Update the SCM section of the POM.xml with the release tag
5. Run the project tests against the modified POMs to confirm everything is in working order
6. Commit the modified POMs
7. Tag the code in the SCM with the release tag
8. Bump the version in the POMs to the next development SNAPSHOT version and remove the release tag from the SCM section.
9. Commit the modified POMs

# mvn release:perform

1. Extract file revisions versioned under the new tag name.
2. Execute the maven build lifecycle on the extracted instance of the project.
3. Deploy the versioned artifacts to appropriate local and remote repositories.

# mvn release:branch

1. Creates the branch in subversion
2. Edits POM.xml on the trunk

- Useful if you have a multi-module project.

**mvn release:branch**
**–DbranchName=<branchTagName>**

# Good points

- Takes care of POM file edits…especially with multi-module projects
- Adds enforcements to ensure tests pass, files are not locally edited, etc.
- Implements a good versioning system which can help with project communication, issue-tracking, auditing, etc.

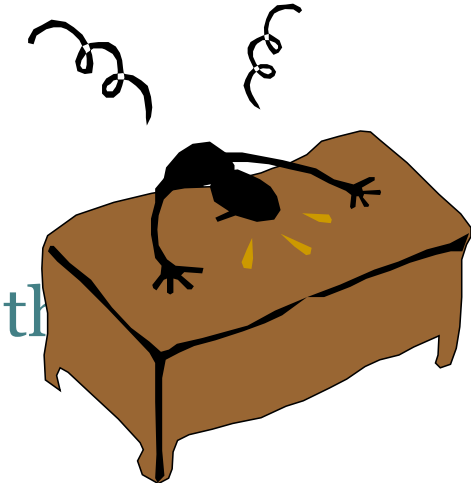
# Pain points

1. Insufficient documentation
2. Separate step to ensure you have the latest code from Subversion.
3. If interrupted mid-stream it is difficult to clean up and start over
4. Intermittent subversion error
5. Need for “mvn install” before starting the release
6. Triple the build time
7. Multi-step process

Apache Website http://maven.apache.org/

# Other Lessons Learned

- settings.xml
  - %M2_HOME%/conf/settings.xml
  - %USER_HOME%/.m2/settings.xml
  - Beware of spaces (Use Progra~1 rather th
    Program Files)
- WAR overlays
  - Share non-Java web resources (css, js, jsp, etc)



# Wrap up/Summary of Maven benefits

- Standardized project layout and project structure generator.
- Standardized dependency-management mechanism.
- Multiple project support.
- Instant downloads of new plugins and features as the developer needs them.
- Website generation for up-to-date project information.
- Integration with source control: CVS and Subversion.

# Reference

- Apache Website http://maven.apache.org/
  - http://maven.apache.org/guides/index.html
- Free books on the web
  - *Better Builds with Maven*
    - http://www.maestrodev.com/better-build-maven
  - *Maven: The Definitive Guide* **and** *Maven by Example*
    - http://www.sonatype.com/documentation/books

# Questions