

MAVEN

Pre-reqs and required softwares

- Knowledge of Java programming language
- Knowledge of Java web application
- Basic knowledge of build tool such as Ant will be an advantage
- Basic XML
- Required softwares:
 - JDK 5+, Apache Maven 2+
 - Eclipse
 - Web server like Jboss to run web apps
 - Internet connectivity

Session Contents

- An Introduction to Maven
 - Key Features of Maven
 - Alternatives to Maven
 - Comparing Maven and Ant
- Installing Maven
 - Maven Directory
 - Maven Settings (Local and Global)
 - Local Maven Repository
 - Running Maven
 - Maven Options
- Core Concepts
 - Maven Directory Structure
 - Standard Maven Lifecycle
 - Key Lifecycle Phases
 - Customizing the Lifecycle
 - Lifecycle Overview
 - Default Lifecycle
 - Site Lifecycle
 - Package-specific Lifecycles

Session Contents

- Maven Plugins and Goals
 - Maven Coordinates
 - Dependency Management
 - Maven Repositories
 - Maven Archetypes
- Project and Dependencies
 - The Project Object Model (POM)
 - The Simplest POM
 - POM Components
 - Project Coordinates
 - GAV (Group, Artifact, Version)
 - Project Versions
 - Snapshot Versions
- Dependencies
 - Dependency Scope
 - Optional Dependencies
 - Version Ranges
 - Transitive Dependencies
 - Visualizing Dependencies
 - Dependency conflicts

Session Contents

- POM Inheritance and Aggregation
 - POM Inheritance
 - The Super POM
 - Inherited Behavior
 - The Effective POM
 - Multi- Projects
 - Directory Structure
 - Container Projects
 - POM Aggregation

INTRODUCTION

Common problems and activities

- Multiple jars
- Dependencies and versions
- Project structure
- Build, publish deploy

"The build" can contain many things

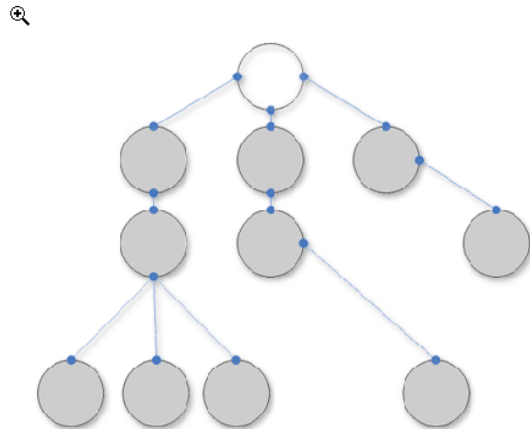
- Compilation of source files (for languages/environments that support a separate/explicit compilation step)
- Linking of object code (for languages/environments that support a separate/explicit linking step)
- Production of distribution packages, also called "installers"
- Generation of documentation that is embedded within the source code files, e.g. Doxygen, Javadoc
- Execution of automated tests like unit tests, static analysis tests, and performance tests
- Generation of reports that tell the development team how many warnings and errors occurred during the build
- Deployment of distribution packages. Eg, the build could automatically deploy/publish a new version of a web application (assuming that the build is successful).

What is Maven?

A build tool

```
C:\WINDOWS\system32\cmd.exe
Downloading: http://repo1.maven.org/maven2/org/apache/maven/wagon/wagon/1.0-alpha-4/wagon-1.0-alpha-4.pom
3K downloaded
Downloading: http://repo1.maven.org/maven2/org/apache/maven/wagon/wagon-provider-api/1.0-alpha-4/wagon-provider-api-1.0-alpha-4.jar
45K downloaded
Downloading: http://repo1.maven.org/maven2/org/apache/maven/maven-artifact-manager/2.0-alpha-3/maven-artifact-manager-2.0-alpha-3.jar
32K downloaded
[INFO] Installing: install
[INFO] Installing C:\my-app\target\my-app-1.0-SNAPSHOT.jar to C:\Documents and Settings\Administrator\TOSHIBA\.m2\repository\com\mycompany\app\my-app\1.0-SNAPSHOT\my-app-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 47 seconds
[INFO] Finished at: Fri Jun 24 16:24:10 PDT 2005
[INFO] Final Memory: 2M/5M
[INFO] -----
C:\my-app>
```

A dependency management tool
(automatic downloads)



A documentation tool



What is Maven?

- Maven is a :
 - project management framework
 - build tool
 - a scripting framework
- Maven encompasses a set of build standards, an artifact repository model, and a software engine that manages and describes projects.
 - It defines a standard life cycle for building, testing, and deploying project artifacts.
 - It provides a framework that enables easy reuse of common build logic for all projects following Maven's standards.
 - Simplifies builds, documentation, distribution, and the deployment process

What Does Maven Provide?

- Maven provides you with:
 - A comprehensive model for software projects
 - Tools that interact with this declarative model
- Maven provides a comprehensive model that can be applied to all software projects. The model uses a common project “language”.
- Maven is the s/w tool & is just a supporting element within this model.
 - Projects and systems that use Maven's standard, declarative build approach tend to be more transparent, more reusable, more maintainable, and easier to comprehend.

Maven's Objectives

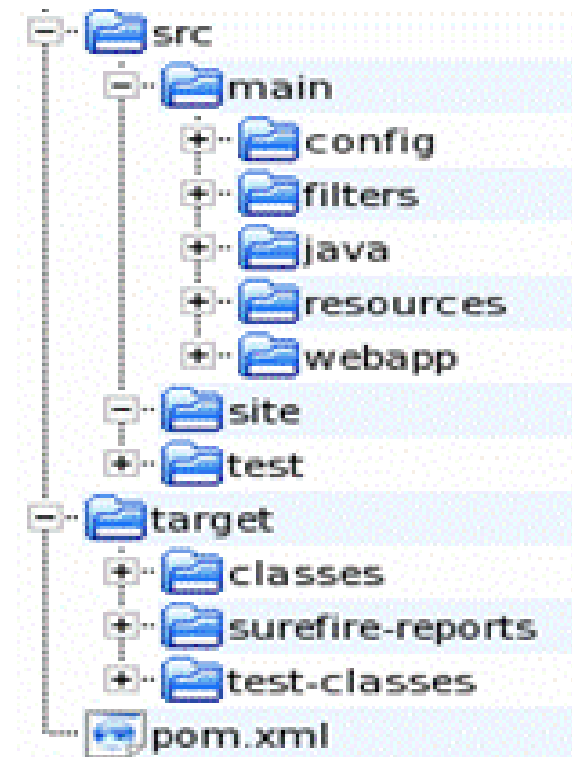
- Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time.
- In order to attain this goal there are several areas of concern that Maven attempts to deal with:
 - Making the build process easy
 - Providing a uniform build system
 - Providing quality project information
 - Providing guidelines for best practices development
 - Allowing transparent migration to new features

Maven's Principles

- Maven principles for creating a shared language:
 - Convention over configuration
 - Declarative execution
 - Reuse of build logic
 - Coherent organization of dependencies
- These principles enable developers to describe their projects at a higher level of abstraction.
 - Thus allowing more effective communication and freeing team members to get on with the important work of creating value at the application level.

Convention over configuration

- Maven uses *Convention over Configuration* which means developers are not required to create build process themselves.
 - Maven provides sensible default behavior for projects.
 - When a Maven project is created, Maven creates default project structure.
 - Developer is only required to place files accordingly and he/she need not to define any configuration in pom.xml.
- Maven employs 3 primary conventions to promote a standardized development environment:
 - Standard directory layout for projects
 - The concept of a single Maven project producing a single output
 - Standard naming conventions



Maven Conventions

- Maven is opinionated about project structure
- target: Default work directory
- src: All project source files go in this directory
- src/main: All sources that go into primary artifact
- src/test: All sources contributing to testing project
- src/main/java: All java source files
- src/main/webapp: All web source files
- src/main/resources: All non compiled source files
- src/test/java: All java test source files
- src/test/resources: All non compiled test source files

Reuse of Build Logic

- Maven promotes reuse by encouraging a separation of concerns.
 - It does this by encapsulating build logic into coherent modules called *plugins*
- Plugins are central feature in Maven that allow for reuse of common build logic across multiple projects.
 - Project build can be customized by using existing or custom plugins
 - In Maven there is a plugin for compiling source code, a plugin for running tests, a plugin for creating JARs, a plugin for creating Javadocs, and many other functions
 - The execution of Maven plugins is coordinated by Maven's build lifecycle with instructions from POM

Maven can be thought of as a framework that coordinates the execution of plugins in a well defined way.

Declarative Execution

- Everything in Maven is driven in a declarative fashion using *Maven's Project Object Model* (POM) and specifically, the plugin configurations contained in the POM.
- **POM (Project Object Model)?**
 - POM consists of entire project information in a xml file (pom.xml)
 - POM plays a vital role that drives Maven execution as a model-driven execution
 - POM file can be inherited to reuse the project resources in another project
- The execution of Maven's plugins is coordinated by Maven's build life cycle in a declarative fashion with instructions from Maven's POM.
- **Buildlife cycle?**
 - In Maven buildlife cycle consists of series of phases where each phase can perform one or more actions
 - Facilitates automatic build process

POM

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

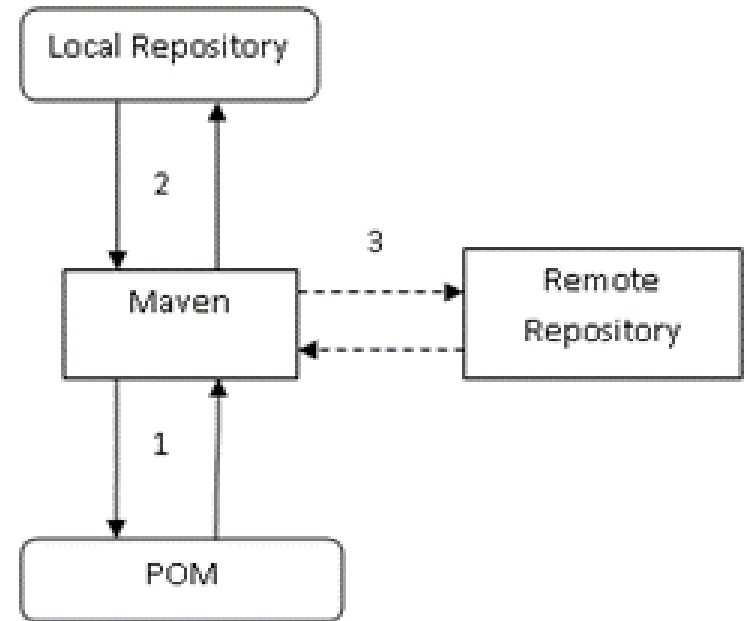
groups a set of
related artifacts

Project's main
identifier; artifactIds
are unique within a
particular groupId.

Signifies that a
project is currently
under active
development.

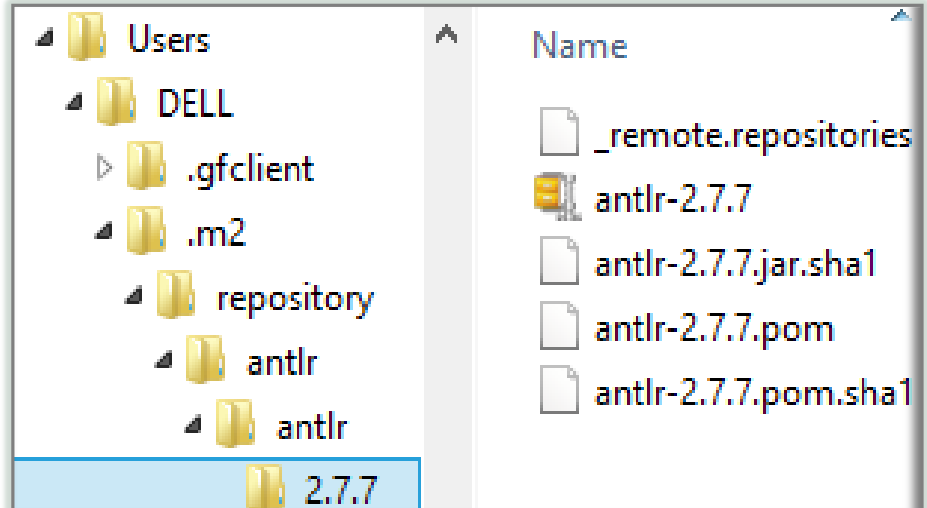
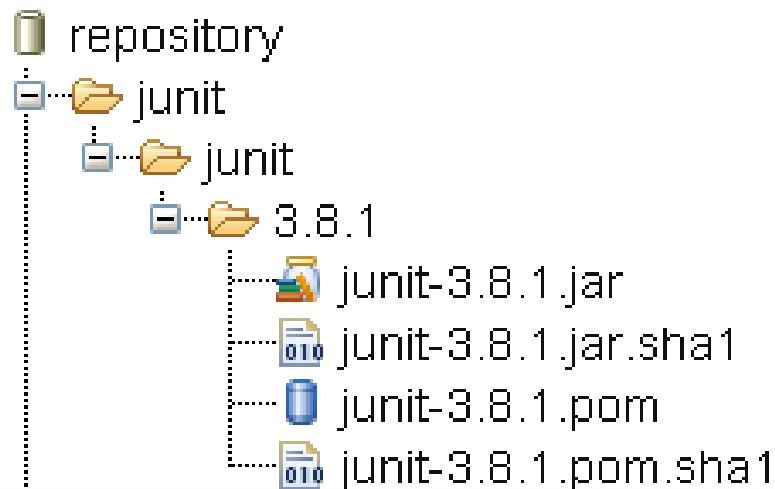
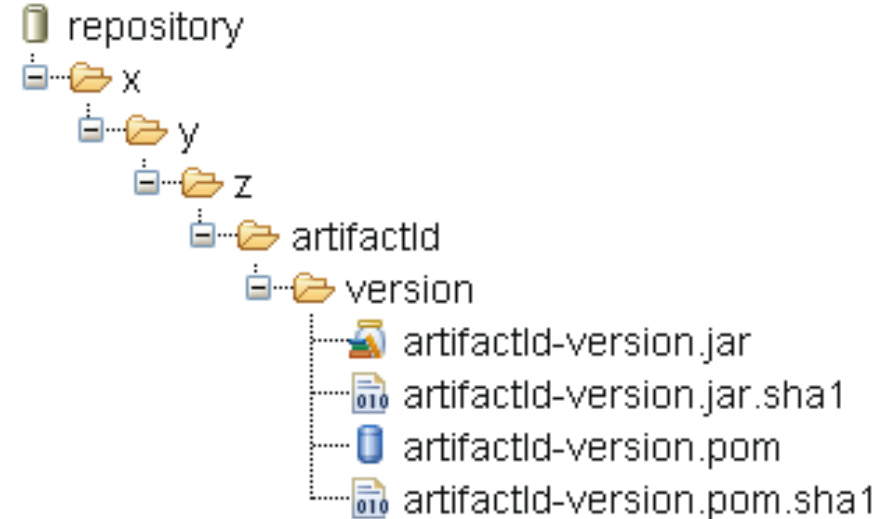
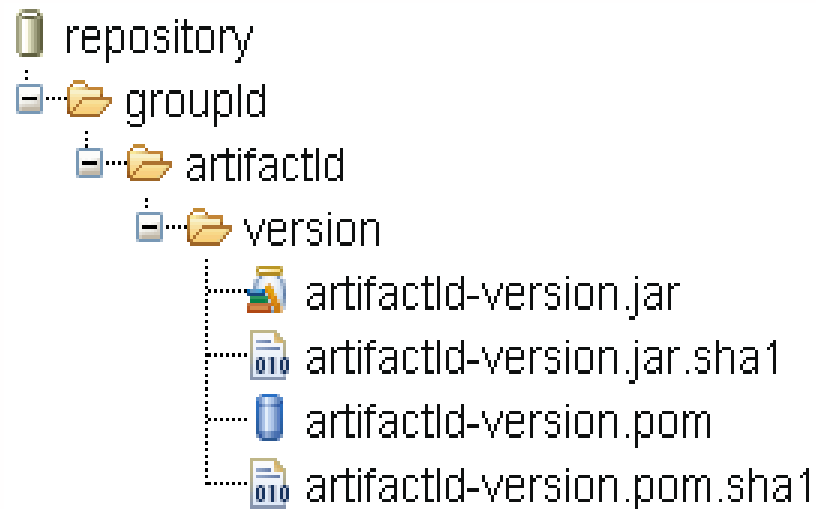
Coherent Organization of Dependencies

- A dependency is a reference to a specific artifact that resides in a repository
 - Dependencies are requested in a declarative fashion with dependency's coordinates by looking up in repositories
- There are 2 repositories available:
 - **Local repository** : By default Maven creates your local repository in `~/.m2/repository`
 - **Remote repository** : Default central Maven repository
<http://repo1.maven.org/maven2>



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

Local Maven repository



Other Java Build Tools

- Ant (2000)
 - Granddaddy of Java Build Tools
 - Scripting in XML
 - Very flexible
- Ant+Ivy (2004)
 - Ant but with Dependency Management
- Gradle (2008)
 - Attempt to combine Maven structure with Groovy Scripting
 - Easily extensible
 - Immature
- Buildr

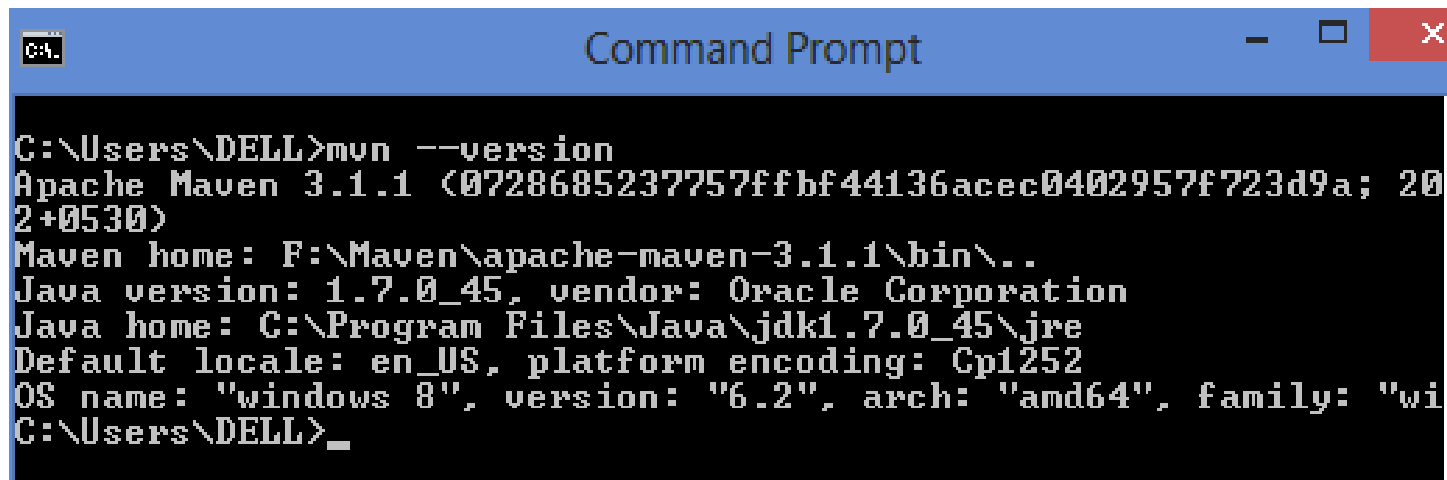
Maven vs ANT

Maven	Ant
Standard Naming Conventions	No Formal Conventions
Declarative	Procedural
Build Lifecycle	Own Lifecycle
Reusable plugins, repositories	Scripts are not reusable
Standard Directory Layout	No Standard Layouts followed
Reuse of Project Object Model file is easy	Reuse of build file is hard
Generates a website of project	Not possible to generate website
Dependencies downloaded automatically	Manually goal dependencies need to be set

GETTING STARTED WITH MAVEN

Installing Maven

- Download & install latest version from maven.apache.org/download.html
 - Unzip Maven distribution to a desired location like C:\Program Files. The distribution stores everything under a subdirectory, eg, apache-maven-3.1
 - We'll refer to this location as M2_HOME, so for example this would C:\Program Files/apache-maven-3.1.1.
 - Add the bin subdirectory of M2_HOME to your PATH variable.
- Verify the installation by running the following command:
 - `mvn --version`



```
Command Prompt

C:\Users\DELL>mvn --version
Apache Maven 3.1.1 (0728685237757ffbf44136acec0402957f723d9a; 2012-05-30)
Maven home: F:\Maven\apache-maven-3.1.1\bin\..
Java version: 1.7.0_45, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.7.0_45\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 8", version: "6.2", arch: "amd64", family: "wi
C:\Users\DELL>
```


Installing Maven

- If you are behind a firewall, then you will have to set up Maven to understand that.
- Edit settings.xml file with the following content:

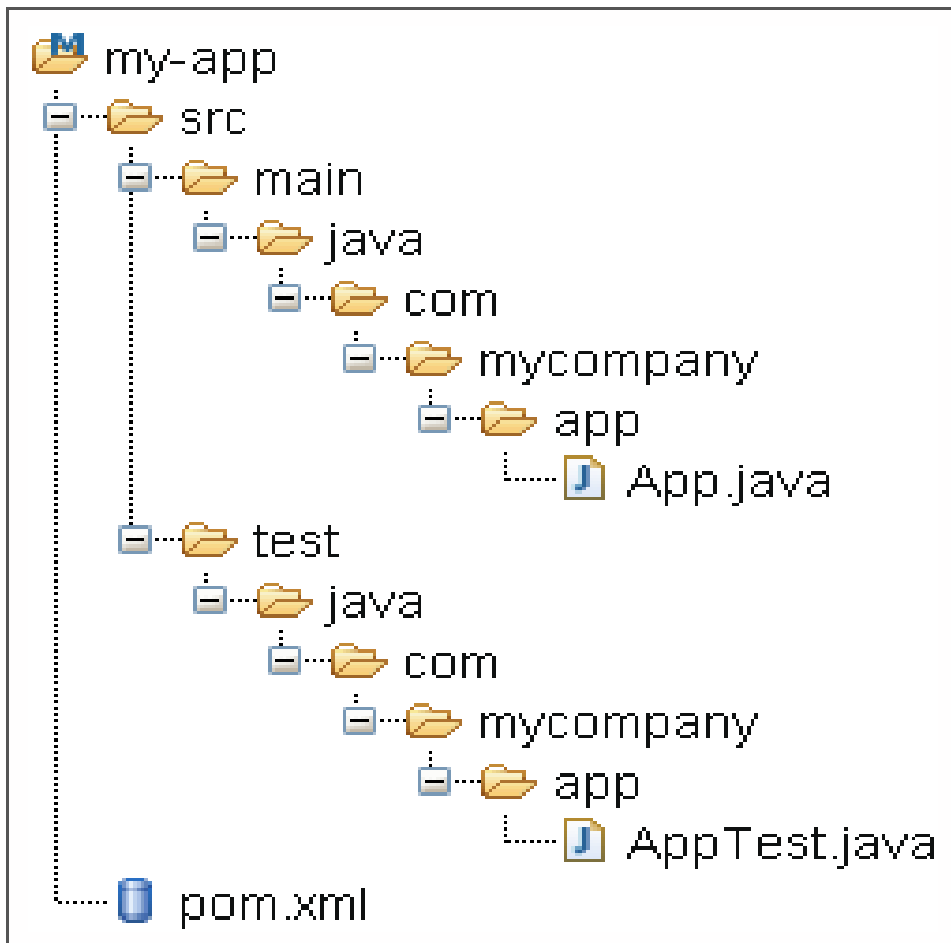
```
<settings>
<proxies>
<proxy>
<active>true</active>
<protocol>http</protocol>
<host>proxy.mycompany.com</host>
<port>8080</port>
<username>your-username</username>
<password>your-password</password>
</proxy>
</proxies>
</settings>
```

Creating Your First Maven Project

- Using Maven's archetype mechanism - an archetype is a template of a project which is combined with some user input to produce a working Maven project that has been tailored to the user's requirements.
- Creating a project from an archetype involves three steps:
 - the selection of the archetype,
 - the configuration of that archetype,
 - the effective creation of the project from the collected information.
- `mvn archetype:generate` : the plugin will first ask to choose the archetype from the internal catalog. Enter the number of the archetype.
 - Then enter values for `groupId`, `artifactId` and version of the project to create and the base package for the sources.

Demo

- Run archetype:generate to generate a quick Java app



Check with:
F:\Maven>tree my-app

Packaging

- Build type identified using the “packaging” element
- Tells Maven how to build the project
- Example packaging types:
 - pom, jar, war, ear, custom
 - Default is jar

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>maven-training</artifactId>
  <groupId>org.lds.training</groupId>
  <version>1.0</version>
  <packaging>jar</packaging>
</project>
```

Demo



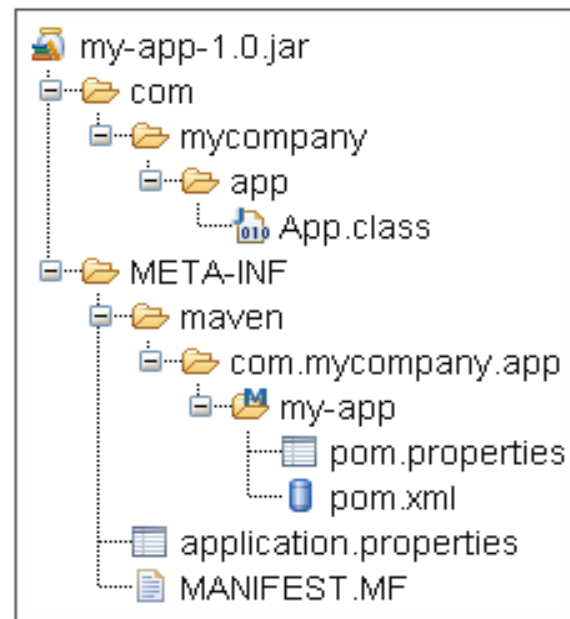
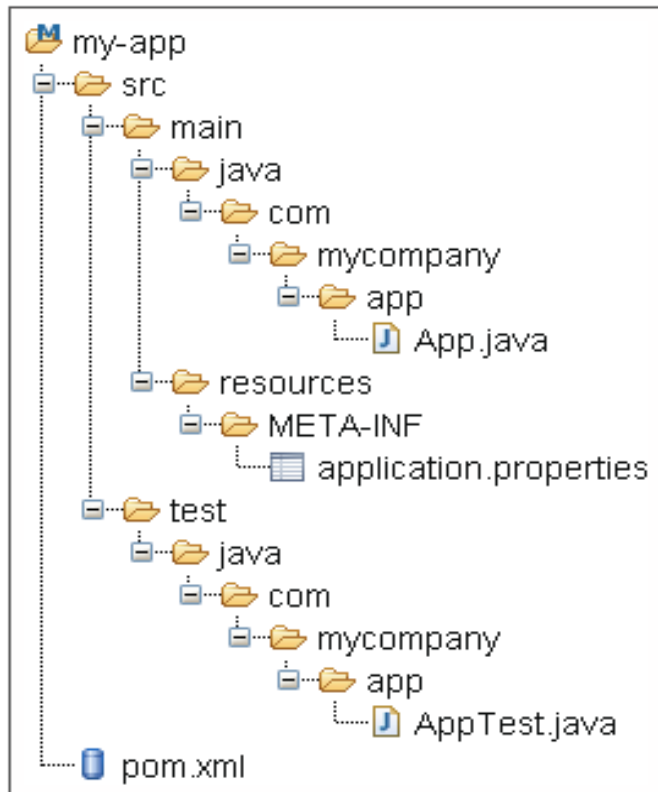
- Compiling Application Sources:
 - mvn compile : compiles entire application.
 - Downloads any plugins required and creates a folder called target
 - mvn package : packages application into a jar file <artifactId>-1.0.jar
 - Also runs all the test cases
 - java -cp target/<artifactId>-1.0.jar com.training.maven.App
 - Displays "Hello World"

The Maven Exec Plugin also allows you to execute Java classes and other scripts. It is not a core Maven plugin, but it is available from the Mojo project hosted by Codehaus. To use:

mvn exec:java -Dexec.mainClass=com.training.MyClass

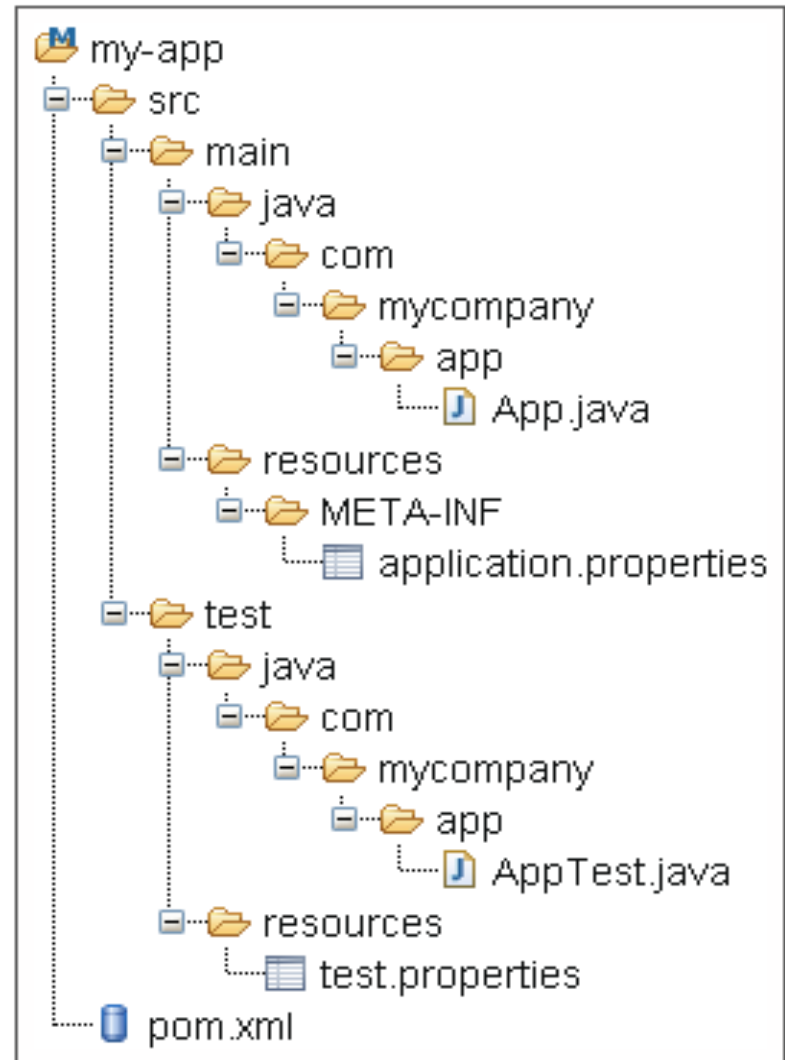
Handling Classpath Resources

- Any resources that needs to be packaged into JAR must go in `src/main/resources`.
 - See how META-INF directory contains `application.properties` file. If you unpacked the JAR that Maven created you would see the following:



Handling Test Classpath Resources

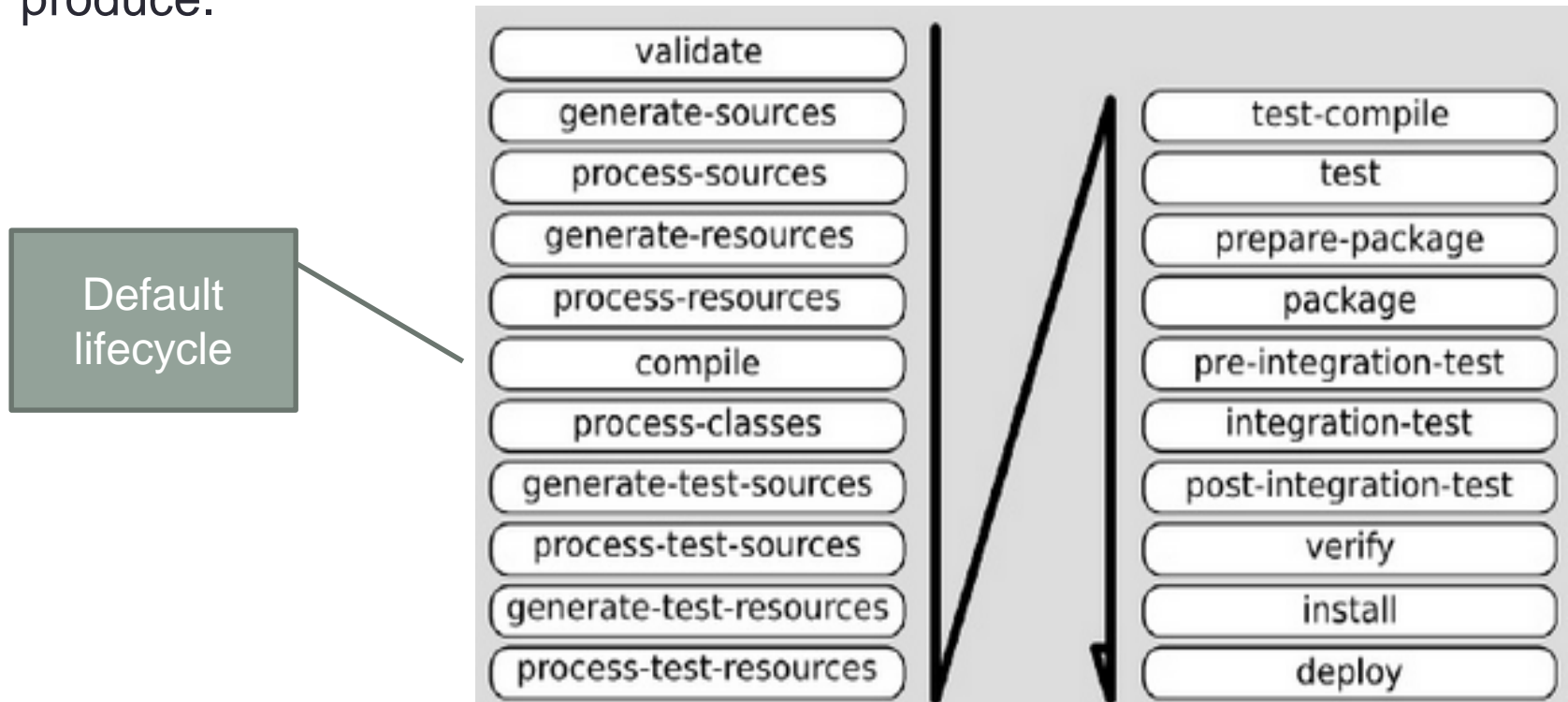
- Place resources in the src/test/resources directory



BUILDING THE PROJECT

Build lifecycle

- In Maven, the build is run using a predefined, ordered set of steps called the build lifecycle.
- The individual steps are called phases, and the same phases are run for every Maven build using the default lifecycle, no matter what it will produce.



Build lifecycle

- When a phase is given, Maven will execute every phase in the sequence up to and including the one defined. Eg, if we execute the compile phase, the phases that actually get executed are:
 - validate
 - generate-sources
 - process-sources
 - generate-resources
 - process-resources
 - Compile
- There are two other Maven lifecycles beyond the default list:
 - clean: cleans up artifacts created by prior builds
 - site: generates site documentation for this project

*For a complete description of all phases refer to
<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>*

Build Goals

- When you tell Maven to build a project, you are telling Maven to step through a defined sequence of phases and execute any goals which may have been registered with each phase.
 - The organized sequence of phases in a build lifecycle gives order to a set of goals. Those goals are chosen and bound by the packaging type of the project being acted upon
- Build goals are the finest steps in the Maven build process.
- A goal can be bound to one or more build phases, or to none at all.
 - If a goal is not bound to any build phase, you can only execute it by passing the goals name to the mvn command.
 - Eg: mvn clean dependency:copy-dependencies package
 - This command will clean the project, copy dependencies, and package the project (executing all phases up to package, of course).

Demo



- **Run:**

1. mvn install : see prj installed in local repository
2. mvn site : generate documentation site for your project information
3. mvn eclipse:eclipse : to convert project into an eclipse project
4. mvn clean : to delete target folder
5. mvn archetype:generate -DgroupId=com.companyname -DartifactId=consumerBanking -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

Demo



- Adding a dependency
 - Edit App.java and bring logging functionality in

The screenshot shows the Sonatype Nexus Professional Edition web interface. The browser address bar displays <https://repository.jboss.org/nexus/index.h>. The page header includes the JBoss Community logo and the text "Sonatype Nexus™ Professional Edition, Version: 2.6.4".

On the left sidebar, under "Sonatype™ Servers", the "nexus" server is selected. Below it, the "Artifact Search" section is active, showing a search bar and a link to "Advanced Search". The "Views/Repositories" section shows "Repositories" selected.

The main content area is titled "Welcome" and "Search". A "Keyword Search" box contains the text "slf4j". Below the search bar, a tree view shows the repository structure: "Maven Central" > "org" > "slf4j13" > "slf4j-log4j13" > "1.0-beta9" > "slf4j-log4j13-".

On the right, the "Maven" tab is selected, displaying the following details:

Group:	org.slf4j13
Artifact:	slf4j-log4j13
Version:	1.0-beta9
Extension:	pom
XML:	<pre><dependency> <groupId>org.slf4j13</groupId> <artifactId>slf4j-log4j13</artifactId> <version>1.0-beta9</version> <type>pom</type> </dependency></pre>

Demo

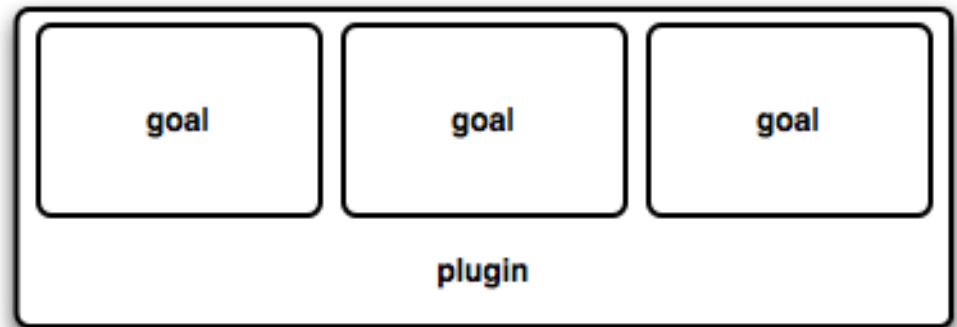


- Creating **a web app** in interactive mode.
 - Package into WAR & Deploy in server
- mvn archetype:generate -DgroupId={project-packaging} -DartifactId={project-name} -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=**false** : : to generate a web application
- mvn eclipse:eclipse -Dwtpversion=2.0 : To convert the Maven web project to support Eclipse IDE
 - *You must add the -Dwtpversion=2.0 argument to make it as a Eclipse web project. Imports it into Eclipse IDE, with a globe icon on top of project*

PLUGINS AND GOALS

Plugins

- Maven delegates most responsibility to a set of Maven Plugins which can affect the Maven Lifecycle & offer access to goals.
- A Maven Plugin is a collection of one or more goals.
 - Almost any action that you can think of performing on a project is implemented as a Maven plugin.
 - Examples of Maven plugins can be simple core plugins like the Jar plugin which contains goals for creating JAR files, Compiler plugin which contains goals for compiling source code and unit tests, or the Surefire plugin which contains goals for executing unit tests and generating reports.
 - Other, more specialized Maven plugins include plugins like the Hibernate3 plugin for integration with the popular persistence library Hibernate



Plugins

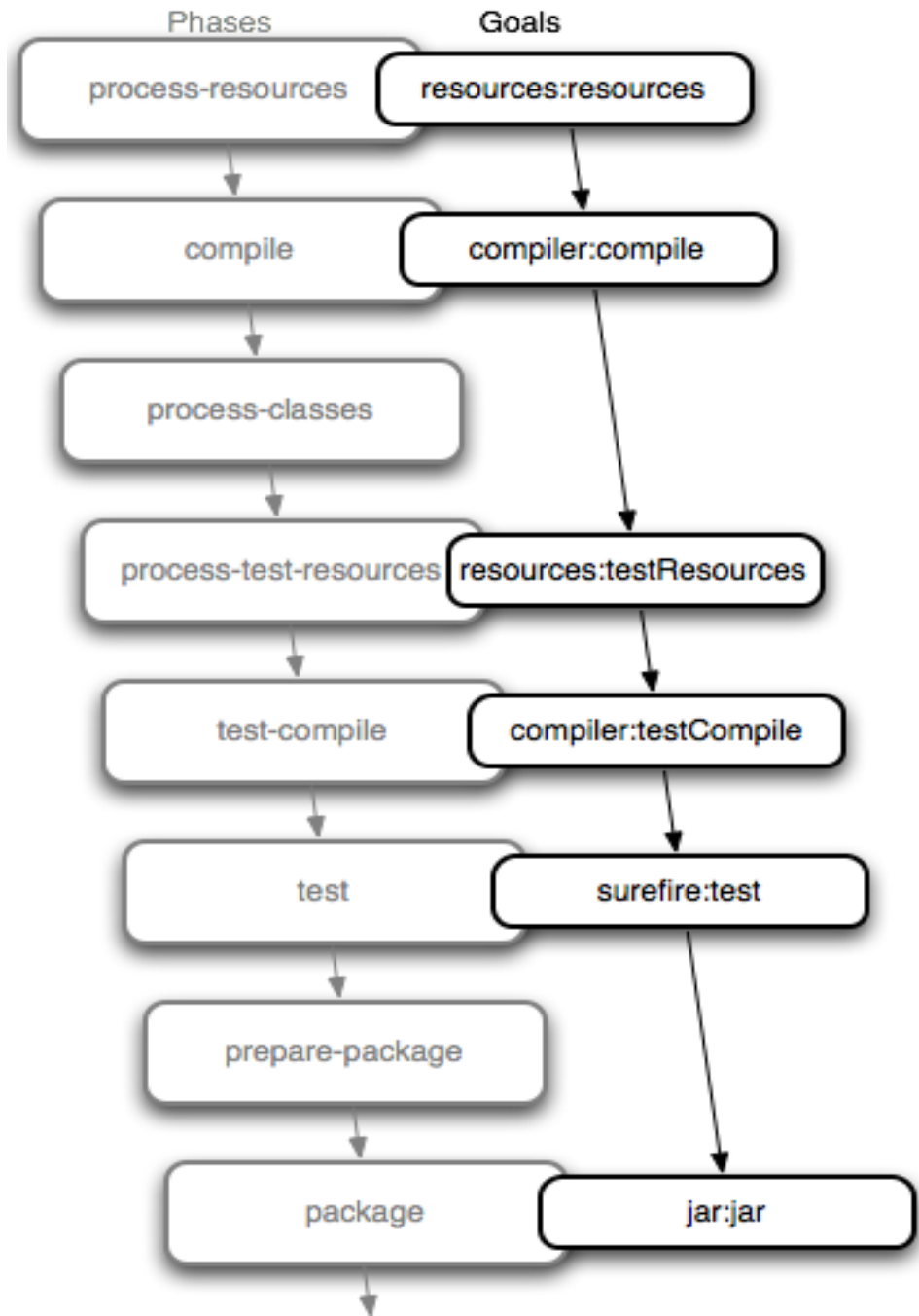
- Plugins are the central feature of Maven that allow for the reuse of common build logic across multiple projects.
- They do this by executing an "action" (i.e. creating a WAR file or compiling unit tests) in the context of a project's description - the POM.
 - For eg, The Maven Clean plugin (maven-clean-plugin) is responsible for removing the target directory of a Maven project. When you run "mvn clean", Maven executes the "clean" goal as defined in the Clean plug-in, and the target directory is removed.
- Plugin behavior can be customized through a set of unique parameters

Goals

- A goal is a specific task that may be executed as a standalone goal or along with other goals as part of a larger build.
- A goal is a “unit of work” in Maven.
 - Examples of goals include the **compile** goal in the Compiler plugin, which compiles all of the source code for a project, or the **test** goal of the Surefire plugin, which can execute unit tests.
- Goals are configured via configuration properties that can be used to customize behavior
 - Eg: `mvn archetype:generate -DgroupId=com.demo -DartifactId=HelloWorldPrj -DarchetypeArtifactId=maven-archetype-quickstart`
 - *Notice how we passed in configuration parameters groupId and artifactId to the generate goal of the Archetype plugin via the command-line parameters*
- Goals define parameters that can define sensible default values
 - Eg, in above eg, we had omitted the packageName parameter, so the package name would default to com.demo

A Goal Binds to a Phase

- Goals are executed as Maven steps through the phases preceding package in the Maven lifecycle
- Each phase corresponds to zero or more goals
- When Maven executes a goal, each goal has access to the information defined in a project's POM.
- *Goals are packaged in Maven plugins which are tied to a phases in a build lifecycle.*



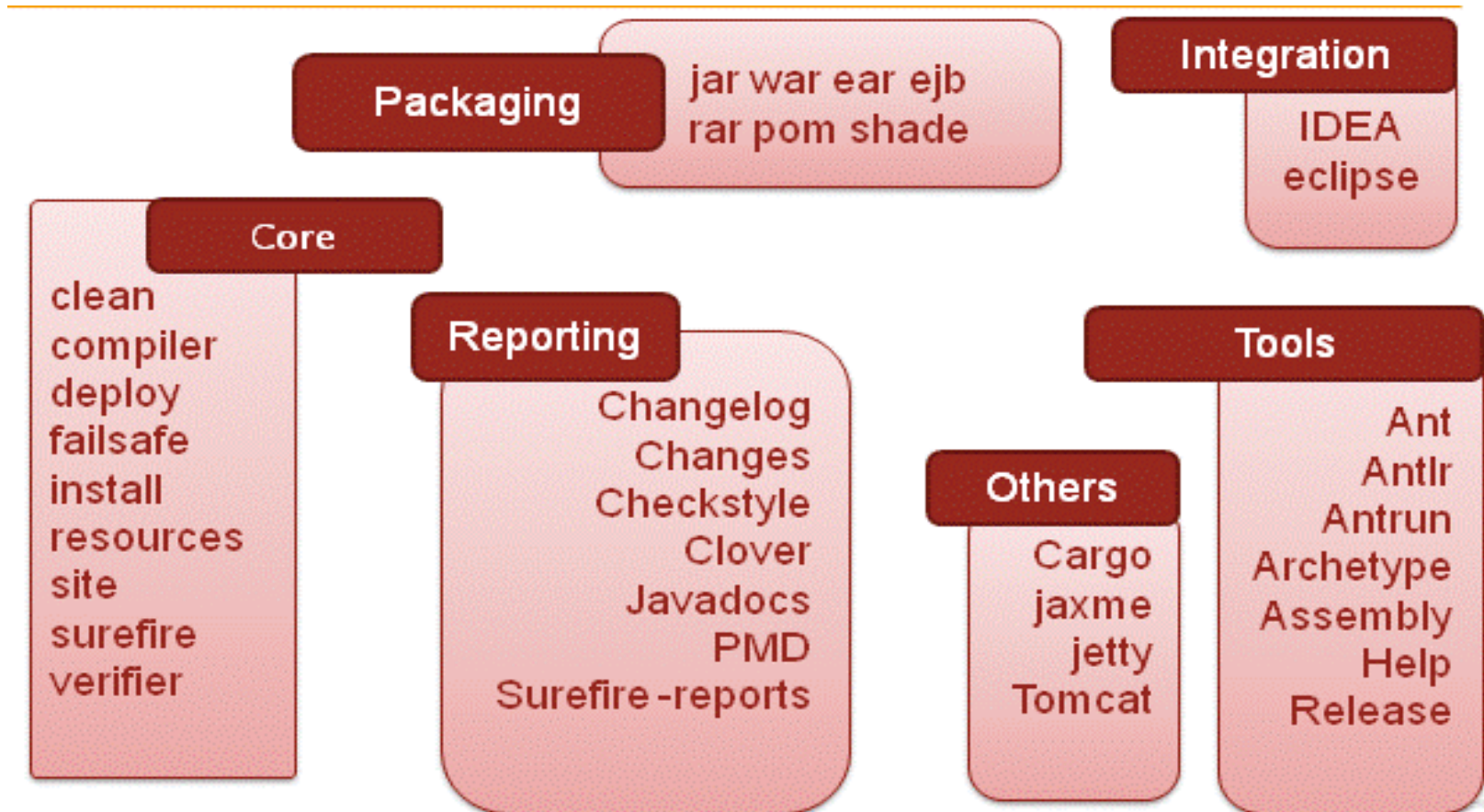
Plugins

- A plugin generally provides a set of goals and which can be executed using following syntax:
 - `mvn [plugin-name]:[goal-name]`
- Eg, a Java project can be compiled with the maven-compiler-plugin's compile-goal by running following command
 - `mvn compiler:compile`
- Eg of using plugins →

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Standard maven plugins

- Refer to <http://maven.apache.org/plugins/> for a list of available plugins



MAVEN'S DEPENDENCY MANAGEMENT

Dependencies in Pom.xml

```
...  
<dependencies>  
  <dependency>  
    <groupId>org.codehaus.xfire</groupId>  
    <artifactId>xfire-java5</artifactId>  
    <version>1.2.5</version>  
  </dependency>  
  <dependency>  
    <groupId>junit</groupId>  
    <artifactId>junit</artifactId>  
    <version>3.8.1</version>  
    <scope>test</scope>  
  </dependency>  
  <dependency>  
    <groupId>javax.servlet</groupId>  
    <artifactId>servlet-api</artifactId>  
    <version>2.4</version>  
    <scope>provided</scope>  
  </dependency>  
</dependencies>  
...  
</project>
```

Dependency scope

- Scope controls which dependencies are available in which classpath, and which dependencies are included with an application. There are five scopes available:
 - **compile** - default scope. Compile dependencies are available in all classpaths, and they are packaged.
 - **runtime** - runtime dependencies are required to execute and test the system, but they are not required for compilation. For example, you may need a JDBC API JAR at compile time and the JDBC driver implementation only at runtime.
 - **provided** – provided dependencies are used when you expect the JDK or a container to provide them. For example, if you were developing a web application, you would need the Servlet API available on the compile classpath to compile a servlet, but you wouldn't want to include the Servlet API in the packaged WAR; the Servlet API JAR is supplied by your application server or servlet container. provided dependencies are available on the compilation classpath (not runtime). They are not transitive, nor are they packaged.

Dependency scope

- **test** - test-scoped dependencies are not required during the normal operation of an application, and they are available only during test compilation and execution phases.
- **system** - The system scope is similar to provided except that you have to provide an explicit path to the JAR on the local file system.
 - This is intended to allow compilation against native objects that may be part of the system libraries.
 - The artifact is assumed to always be available and is not looked up in a repository. If you declare the scope to be system, you must also provide the `systemPath` element.

```
<dependency>
  <groupId>com.example</groupId>
  <artifactId>MySpecialLib</artifactId>
  <version>1.2</version>
  <scope>system</scope>
  <systemPath>${basedir}/src/main/webapp/WEB-INF/lib/MySpecialLib-1.2.jar</systemPath>
</dependency>
```

Dependency Version Ranges

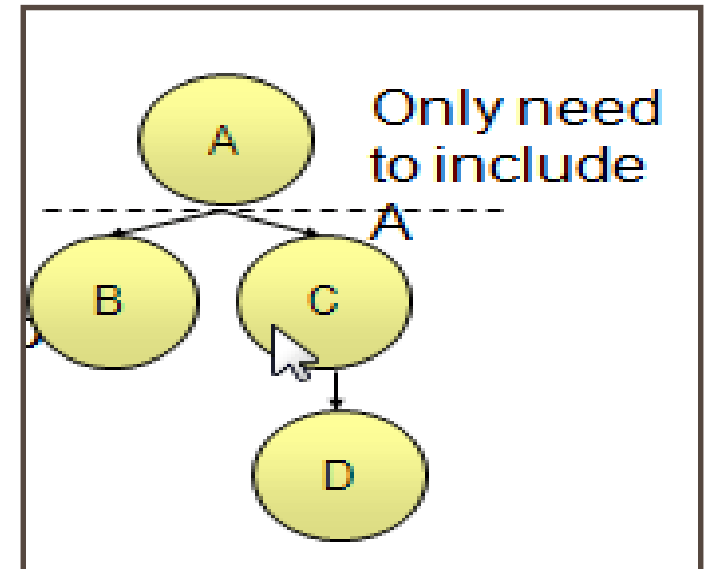
- You don't just have to depend on a specific version of a dependency; you can specify a range of versions that would satisfy a given dependency.
 - For example, you can specify that your project depends on version 3.8 or greater of JUnit, or anything between versions 1.2.10 and 1.2.14 of JUnit.
 - Do this by surrounding one or more version numbers with the following :
 - (,) : Exclusive quantifiers
 - [,] : Inclusive quantifiers

When declaring a "normal" version such as 3.8.2 for Junit, internally this is represented as "allow anything, but prefer 3.8.2"

Versions Range	Meaning
[1.2.12]	A specific version
(,1.0]	Less than or equal to 1.0
[1.2,1.3]	Between 1.2 and 1.3 (inclusive)
[1.0,2.0)	Greater than or equal to 1.0, but less than 2.0
[1.5,)	Greater than or equal to 1.5
(,1.1),(1.1,)	Any version, except 1.1

Transitive Dependencies

- Support for transitive dependencies is one of Maven's most powerful features.
 - Resolving dependencies of dependencies are called transitive dependencies, and they are made possible by the fact that the Maven repository stores more than just bytecode; it stores metadata about artifacts.
- Allows you to avoid needing to discover and specify the libraries that your own dependencies require, and including them automatically.
- More formally spoken, transitivity means that if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$



Excluding Dependency using exclusion element

- There will be times when you need to exclude a transitive dependency, such as when you are depending on a project that depends on another project, but you would like to either exclude the dependency altogether.

```
<dependency>
  <groupId>sample.ProjectA</groupId>
  <artifactId>Project-A</artifactId>
  <version>1.0</version>
  <scope>compile</scope>
  <exclusions>
    <exclusion> <!-- declare the exclusion here -->
      <groupId>sample.ProjectB</groupId>
      <artifactId>Project-B</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

dependency element adds a dependency on project-a, but excludes the **transitive** dependency project-b.

Dependency Management

- Maven provides a way for you to consolidate dependency version numbers in the dependencyManagement element.
 - The dependencyManagement element is normally in a top-level parent POM for an organization or project.
 - Using this element allows to reference a dependency in a child project without having to explicitly list the version.
 - Maven will walk up the parent-child hierarchy until it finds a project with a dependencyManagement element, it will then use the version specified in this dependencyManagement element.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>test</groupId>
      <artifactId>d</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Dependency Management : an example

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
</project>
```

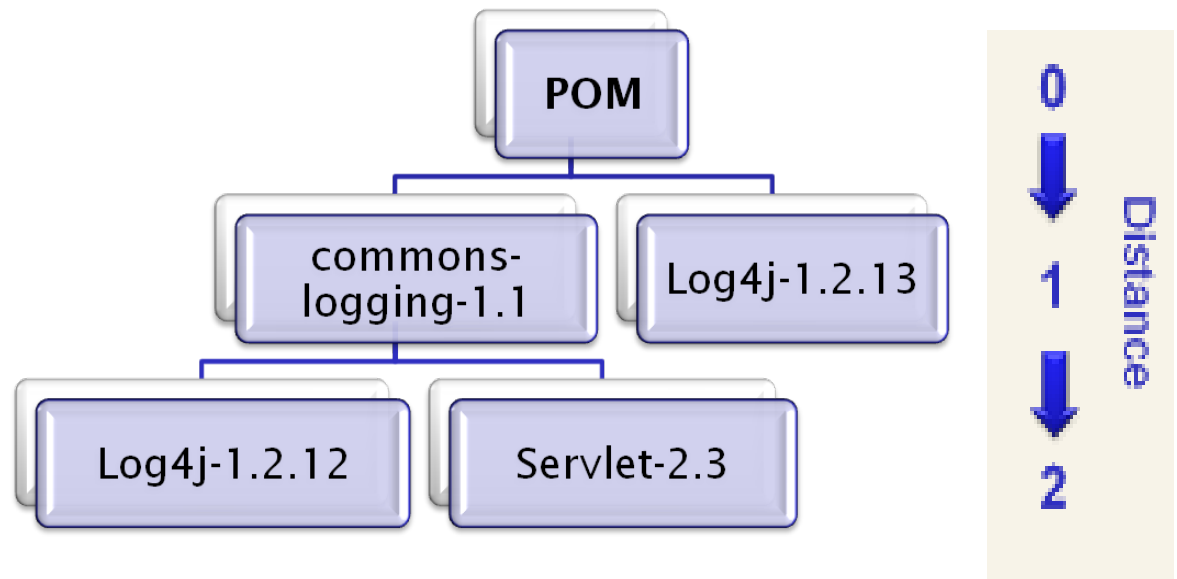
Defining Dependency Versions in a Top-level POM

Adding a dependency to the MySQL Java Connector in a child project

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
    </dependency>
  </dependencies>
</project>
```

Resolving Dependency conflicts

- Conflicts arise when same dependency (eg Log4j) of different version is identified in dependency graph
- While resolving such conflicts, Maven traverses the dependency in a top down manner and selects the version “nearest” to the top of the tree.



Demo

- Demo ProjectA which depends on ProjectB
- See pom.xml
- To know about the entire dependency your project, you can run
- `mvn dependency:tree`

