Advanced Maven Techniques

Anders Hammar

CTO, Devoteam Quaint

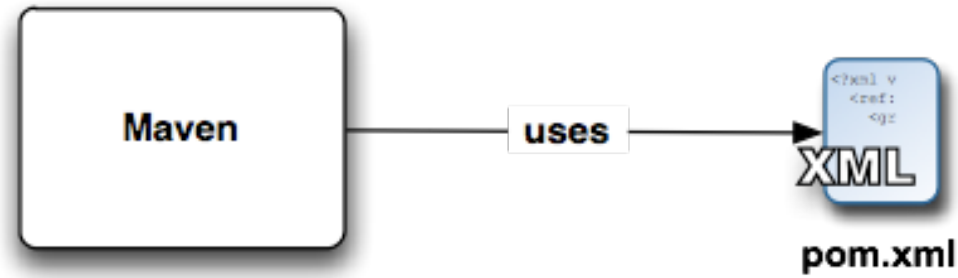
Jfokus 2010

Who am I?

- Consultant at Devoteam Quaint
- 10 years of Java
- 4 years of Maven
- Maven trainer
- Active within the Maven community
- Nexus OSS contributor

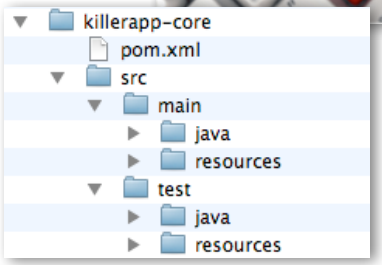
Welcome to Maven

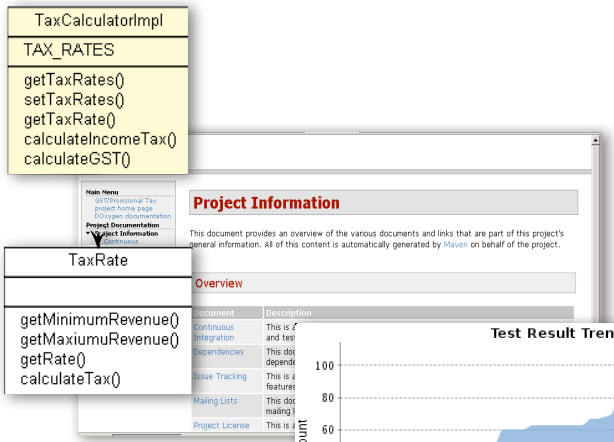
- So what is Maven about, anyway?
  - Maven manages the build process
    - Reuse standard build logic (compile, package,...)
    - Applies it's logic to a project, guided by project description (or "metadata")
  - Maven uses a declarative approach
    - Describe your project, not just the steps required to build it
    - Your project description (or "object model") goes in a `pom.xml` file

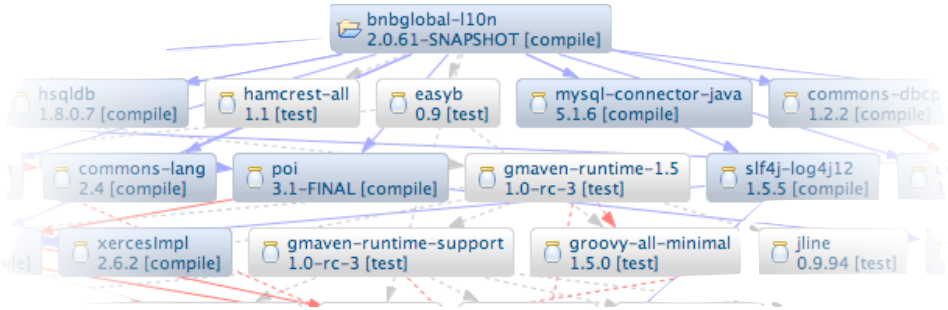


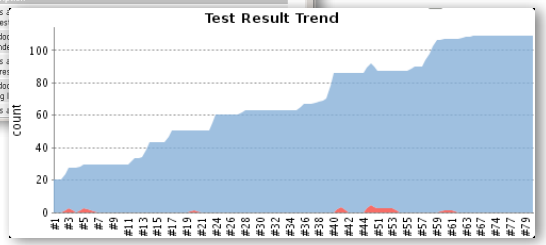
Key Features of Maven

- So how can Maven help me and my team?
  - A standardized build and deployment process
  - A standardized project directory structure
  - Improved dependency management
  - Easy to generate reasonable technical reports









Benefits of Maven

- Build standardization - "A Common Interface"
  - All basic functionality is provided no matter what Maven project you use
- Dependency management
  - No more manual management of dependencies and guessing versions
- Lifecycle management
  - Provides a build life cycle instead of making completely you build your own
- Project management best practices
  - Consistent directory structure provides easy understanding of artifacts

Maven Golden Rule

- A Maven project creates one artifact
  - Secondary artifacts might exist (sources JAR, Javadoc JAR, etc.)
- Want more than one artifact?
  - Create several projects!

Maven Versions

- 2.0
  - latest: 2.0.10
  - 2.0.11 - end-of-life?
- 2.1
  - latest: 2.1.0
  - Do not use - has issues!
- 2.2
  - latest: 2.2.1
- 3.0
  - latest: 3.0-alpha-6

Supported Languages

- Maven requires Java to execute...
- ...but supports many programming languages
  - Java, Flex, .Net, C++, ...

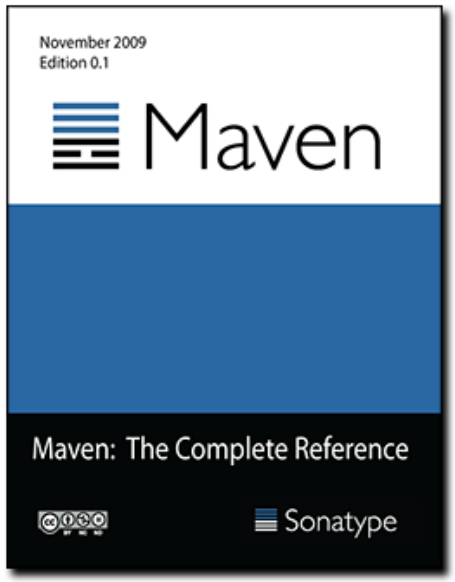
Maven Resources

- Apache Maven Project
  - Website: http://maven.apache.org
  - Mailing Lists: http://maven.apache.org/mail-lists.html
    - Maven Users Mailing List
    - Maven Developers Mailing List
- Sonatype: http://www.sonatype.com
  - Sonatype Blogs: http://blogs.sonatype.com
  - Maven support available from Sonatype

Maven Books

- Maven: By Example
- Maven: The Complete Reference

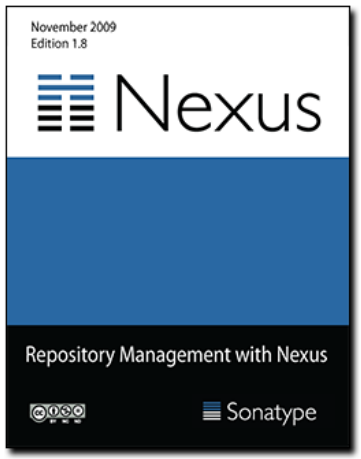
- http://books.sonatype.com

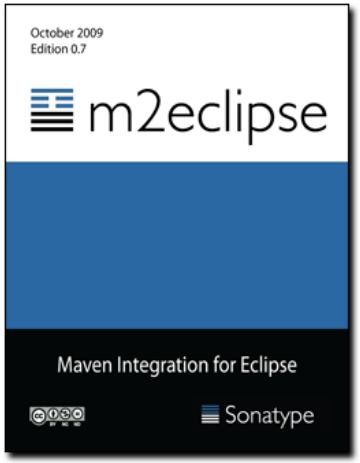


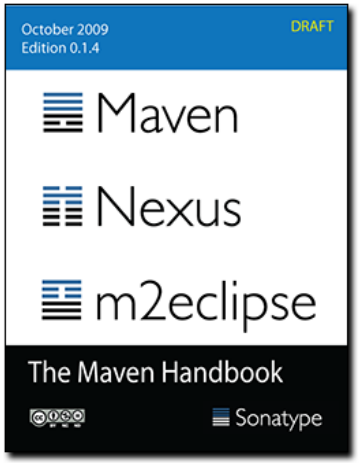
More Maven Books

- Repository Management with Nexus
- Developing with Eclipse and Maven
- The Maven Handbook

- http://books.sonatype.com







Today's Topics

- Advanced Dependency Management
- Lifecycle Customization
- Plugin Management

Credit

- Tutorial based on Maven training material
- Courtesy by Sonatype

Maven @ Jfokus 2010

- *Next Generation Development Infrastructure: Maven, M2Eclipse, Nexus & Hudson* by Jason van Zyl
  - 14.15-15.00, Jan 27
- Also come visit Sonatype's booth!

Advanced Maven Techniques

Maven in your IDE

Part 0 - M2Eclipse

Maven in Eclipse

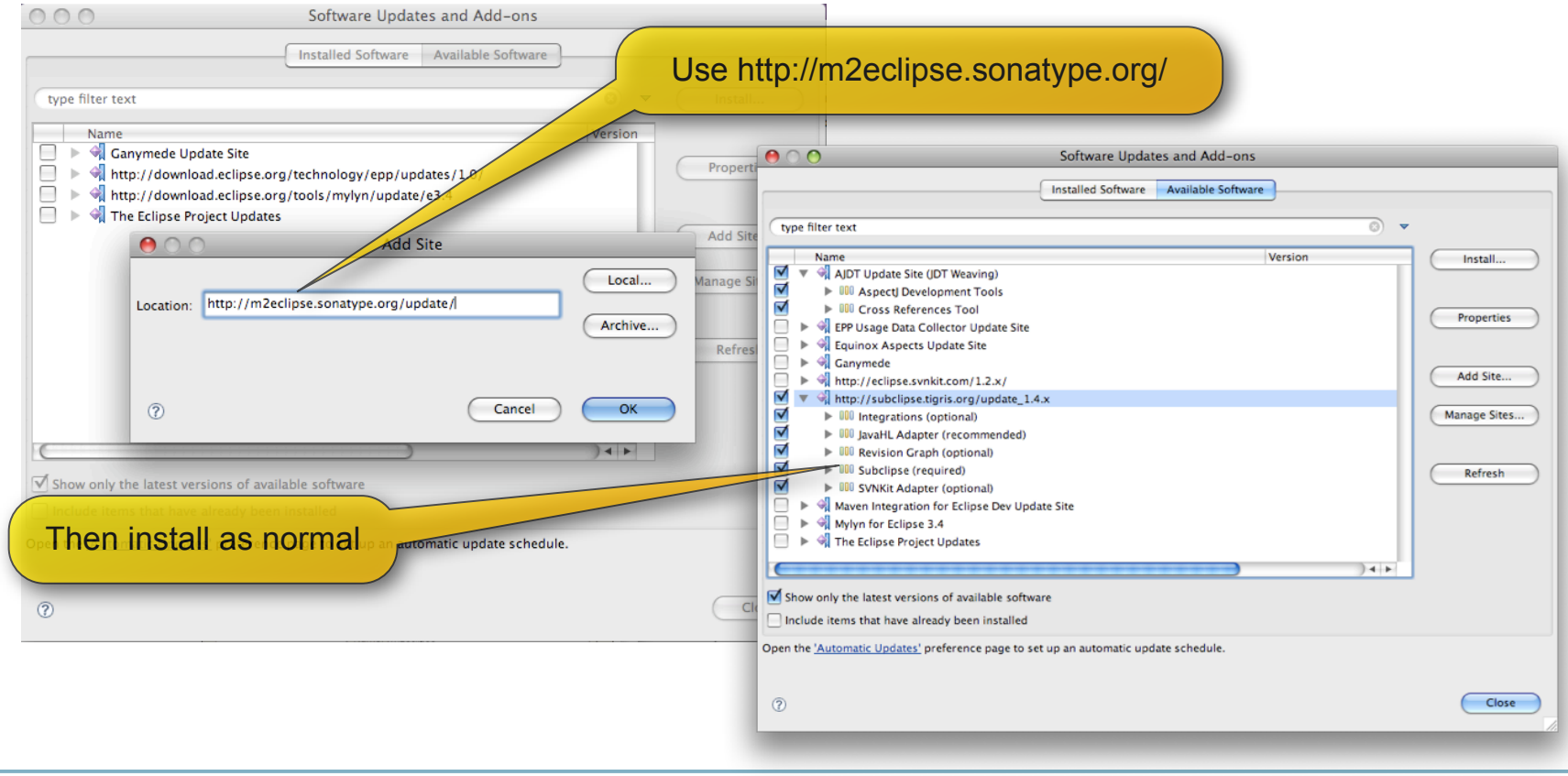
- Maven Eclipse Plugin (maven-eclipse-plugin)
  - mvn eclipse:eclipse
- M2Eclipse
  - Eclipse plugin
- static vs. dynamic

Maven in Eclipse

- Using Maven in Eclipse
  - The *M2Eclipse* plugin currently provides the best IDE support for Maven
    - Eclipse build path based on the POM
    - Launch Maven from within Eclipse
    - Graphical pom editor
    - Dependency graphs
    - Simplified dependency management
    - Quick search for dependencies
    - Automatic download of sources and javadoc
    - Materialize a project from POM

Maven in Eclipse

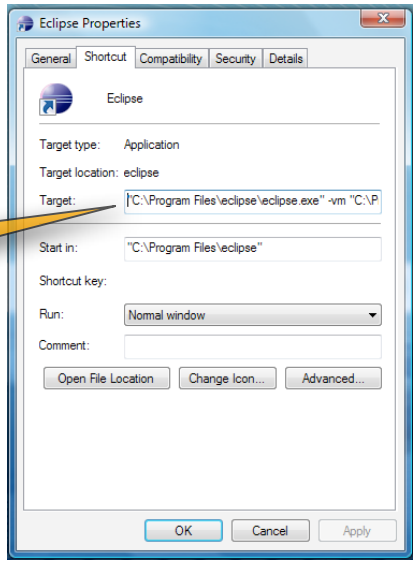
- Installing the *m2eclipse* plugin:
  - Install the m2eclipse plugin from the Sonatype update site



Maven in Eclipse

- Always run Eclipse using a JDK
  - The m2eclipse plugin expects a JDK.
  - Eclipse doesn't always run with a JDK by default
  - Use the **-vm** option to point to your JDK in Windows

```
"C:\Program Files\eclipse\eclipse.exe"
-vm "C:\Program Files\Java\jdk1.6.0_05\bin"
```



Demo

- Let's have a look...

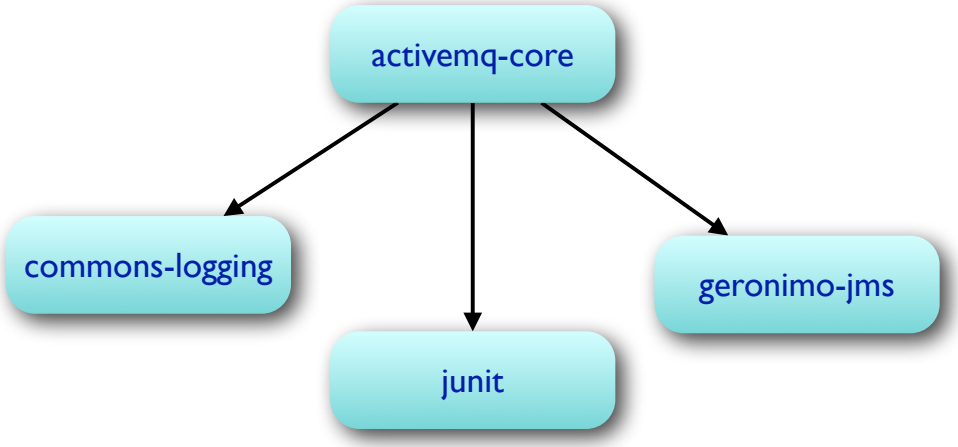
Advanced Maven Techniques

Managing your dependencies

Part 1 - Advanced Dependency Management

Dependencies

- What are dependencies?
  - Artifacts on which a project relies to compile, test, run, etc.
  - Example:
    - activemq-core dependencies
      - commons-logging
      - junit
      - geronimo-jms



Dependencies

- Project dependencies are defined in the POM:
  - Defined using the Maven artifact co-ordinates
  - Defined in the <dependencies> section

```
<dependencies>
  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jms_1.1_spec</artifactId>
    <version>1.1.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.4</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Dependency on commons-logging 1.1

geronimo-jms will be provided by the application server

JUnit 4.4 is only required to compile and execute the tests

Dependency Scope

- Different dependencies have different uses:
  - The commons-logging dependency is a **compile** dependency:
    - The project depends on this artifact for compilation, testing and runtime
  - The geronimo-jms_1.1_spec dependency is a **provided** dependency:
    - The project needs this artifact for compilation and testing; at deployment runtime the container will supply it
  - The junit dependency is a test dependency is a **test** dependency:
    - This projects needs this artifact for testCompile and test phases
- Why scoping?
  - Scoping helps to define the various classpaths for different phases
  - Scoping also affects the packaging phase
    - Whether a dependency is included in the artifact package

Dependency Scope

- Dependency scopes
  - Dependencies can have different scopes:
    - compile
    - provided
    - test
    - runtime
    - system
    - (imported)

Dependency Scope

- Compile scope
  - The default scope
  - Available in all classpaths
  - Bundled with the packaged application
  - Examples: Hibernate, Spring, ...

```
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
        <version>2.5.3</version>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- Provided scope
  - Supplied by the JDK or a container at runtime
  - Include in the compile and test classpaths
  - Don't include it in the final package
  - Examples: Servlet API,...

```
<dependencies>
    ...
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.4</version>
        <scope>provided</scope>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- Test scope
  - Not needed for normal use of the application
  - Included in the test compilation and execution classpaths
  - Not bundled with the packaged application
  - Examples: JUnit, TestNG, ...

```
<dependencies>
    ...
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.4</version>
        <scope>test</scope>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- Runtime scope
  - Required to test and execute the application
  - Not required for compilation
  - Bundled with the packaged application
  - Examples: Oracle JDBC

```
<dependencies>
    ...
    <dependency>
        <groupId>oracle</groupId>
        <artifactId>ojdbc14</artifactId>
        <version>10.2.0.2.0</version>
        <scope>runtime</scope>
    </dependency>
    ...
</dependencies>
```

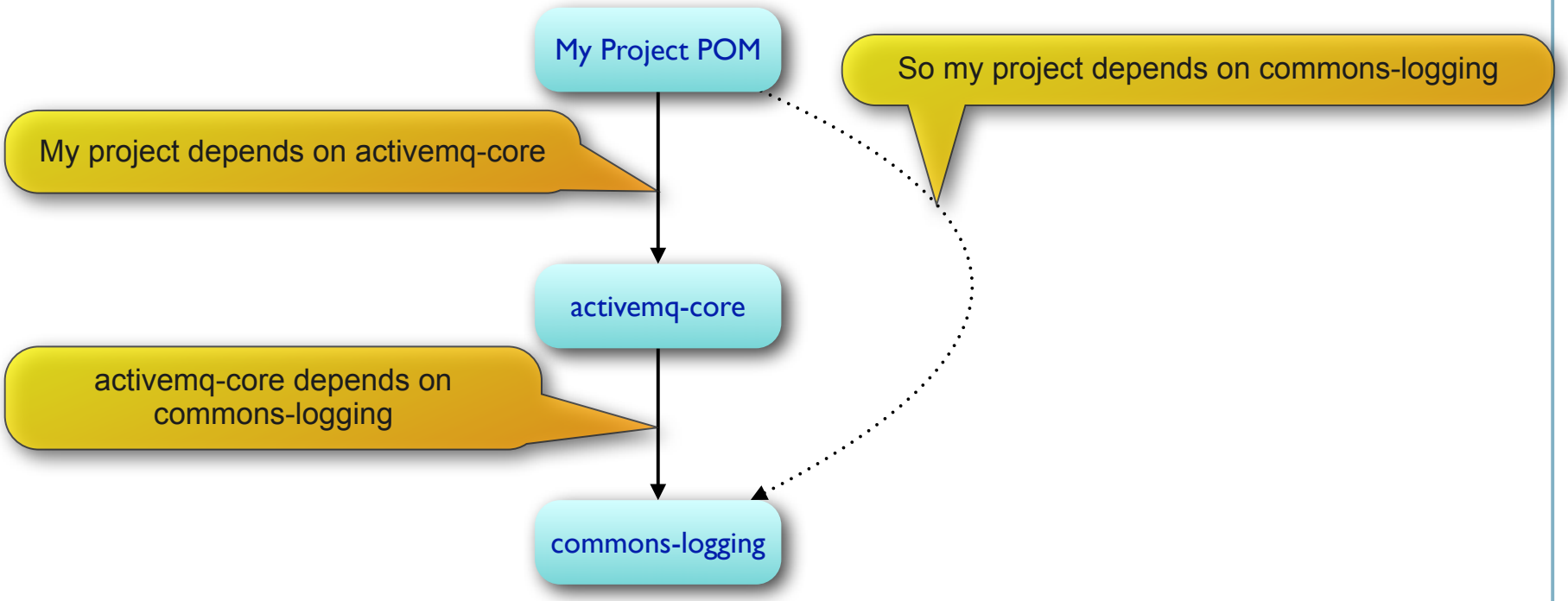
Dependency Scope

- System scope
  - Similar to the provided scope
  - Provide the artifact explicitly as a file path
  - Rarely used (better to use the repositories)

```
<dependencies>
    ...
   <dependency>
      <groupId>CommonsLoging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0</version>
      <scope>system</scope>
      <systemPath>${basedir}/lib/commons-logging-1.0.jar</systemPath>
    </dependency>
    ...
</dependencies>
```

Transitive Dependencies

- Dependencies of a dependency
  - Golden Rule: *"The dependency of my dependency is my dependency. (Mostly)"*



Transitive Dependencies

- Transitive Dependencies
  - POMs declare dependencies on other artifacts
  - Using Maven coordinates, Maven recursively adds the dependencies to the current project
- Maven builds graphs of dependencies and handles any conflicts that may occur
  - Always favors a more recent version of any artifact when selecting from a range

Demo

- Visualizing Dependencies in Eclipse
  - Dependency Hierarchy
  - Dependency Graph

Visualizing Dependencies

- From the command line
  - List your dependencies

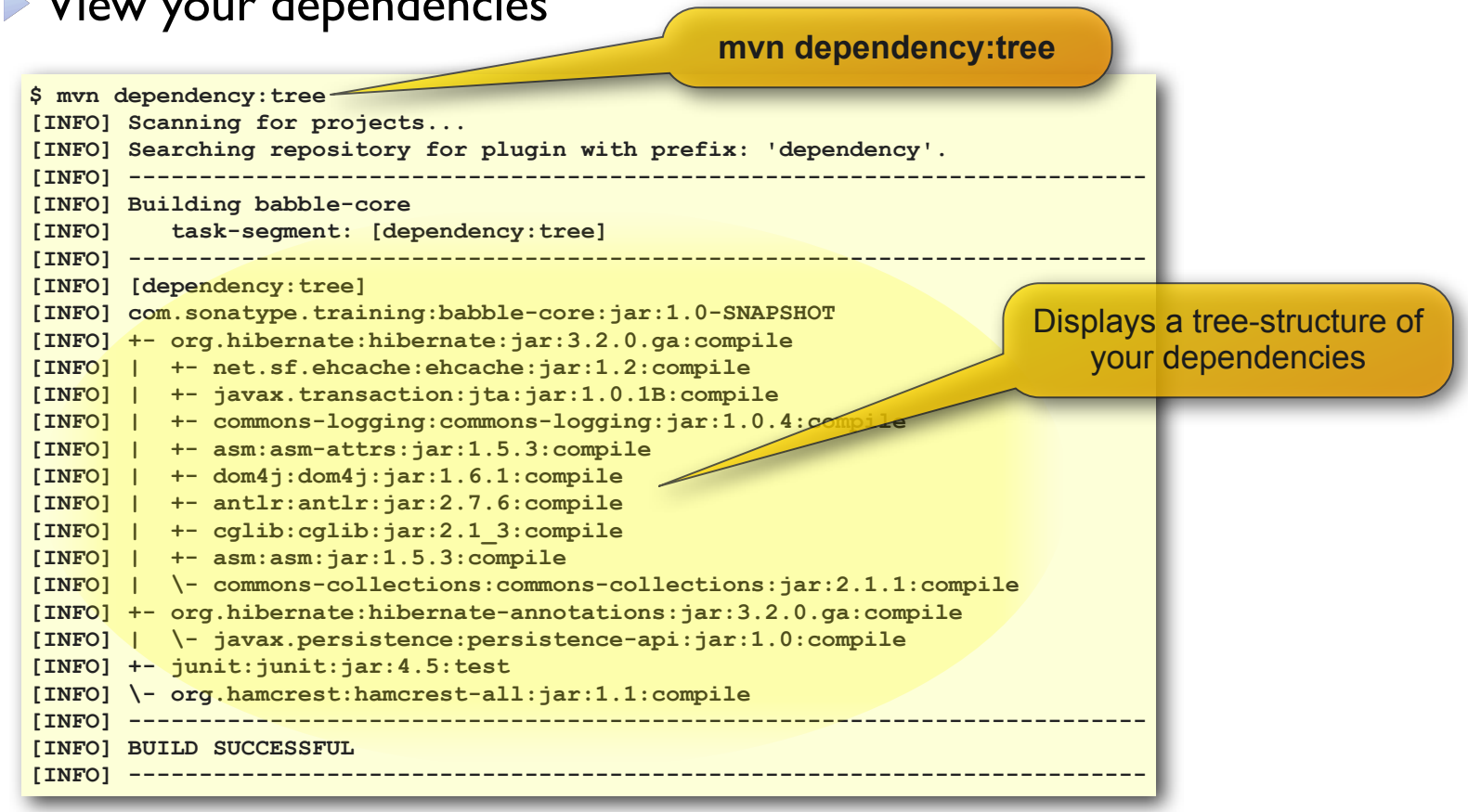
```
$ mvn dependency:list
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building babble-core
[INFO]    task-segment: [dependency:list]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:list]
[INFO]
[INFO] The following files have been resolved:
[INFO]    antlr:antlr:jar:2.7.6:compile
[INFO]    asm:asm:jar:1.5.3:compile
[INFO]    asm:asm-attrs:jar:1.5.3:compile
[INFO]    cglib:cglib:jar:2.1_3:compile
[INFO]    commons-collections:commons-collections:jar:2.1.1:compile
[INFO]    commons-logging:commons-logging:jar:1.0.4:compile
[INFO]    dom4j:dom4j:jar:1.6.1:compile
[INFO]    javax.persistence:persistence-api:jar:1.0:compile
[INFO]    javax.transaction:jta:jar:1.0.1B:compile
[INFO]    junit:junit:jar:4.5:test
[INFO]    net.sf.ehcache:ehcache:jar:1.2:compile
[INFO]    org.hamcrest:hamcrest-all:jar:1.1:compile
[INFO]    org.hibernate:hibernate:jar:3.2.0.ga:compile
[INFO]    org.hibernate:hibernate-annotations:jar:3.2.0.ga:compile
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

mvn dependency:list

Displays a list of resolved dependencies

Visualizing Dependencies

- From the command line
  - View your dependencies




```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building babble-core
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:tree]
[INFO] com.sonatype.training:babble-core:jar:1.0-SNAPSHOT
[INFO] +- org.hibernate:hibernate:jar:3.2.0.ga:compile
[INFO] |  +- net.sf.ehcache:ehcache:jar:1.2:compile
[INFO] |  +- javax.transaction:jta:jar:1.0.1B:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] |  +- asm:asm-attrs:jar:1.5.3:compile
[INFO] |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  +- antlr:antlr:jar:2.7.6:compile
[INFO] |  +- cglib:cglib:jar:2.1_3:compile
[INFO] |  +- asm:asm:jar:1.5.3:compile
[INFO] |  \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] +- org.hibernate:hibernate-annotations:jar:3.2.0.ga:compile
[INFO] |  \- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- junit:junit:jar:4.5:test
[INFO] \- org.hamcrest:hamcrest-all:jar:1.1:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

Visualizing Dependencies

- From the command line
  - Optimize your dependencies
  - Find unused dependencies
  - Declare important dependencies more precisely

mvn dependency:analyze

```
$ mvn dependency:analyze
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building babble-core
[INFO]    task-segment: [dependency:analyze]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing dependency:analyze
...
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]    javax.persistence:persistence-api:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]    org.hibernate:hibernate-annotations:jar:3.2.0.ga:compile
[WARNING]    org.hibernate:hibernate:jar:3.2.0.ga:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Mon Mar 30 14:19:04 NZDT 2009
[INFO] Final Memory: 13M/26M
[INFO] ------------------------------------------------------------------------
```

JPA annotations are used but not directly declared

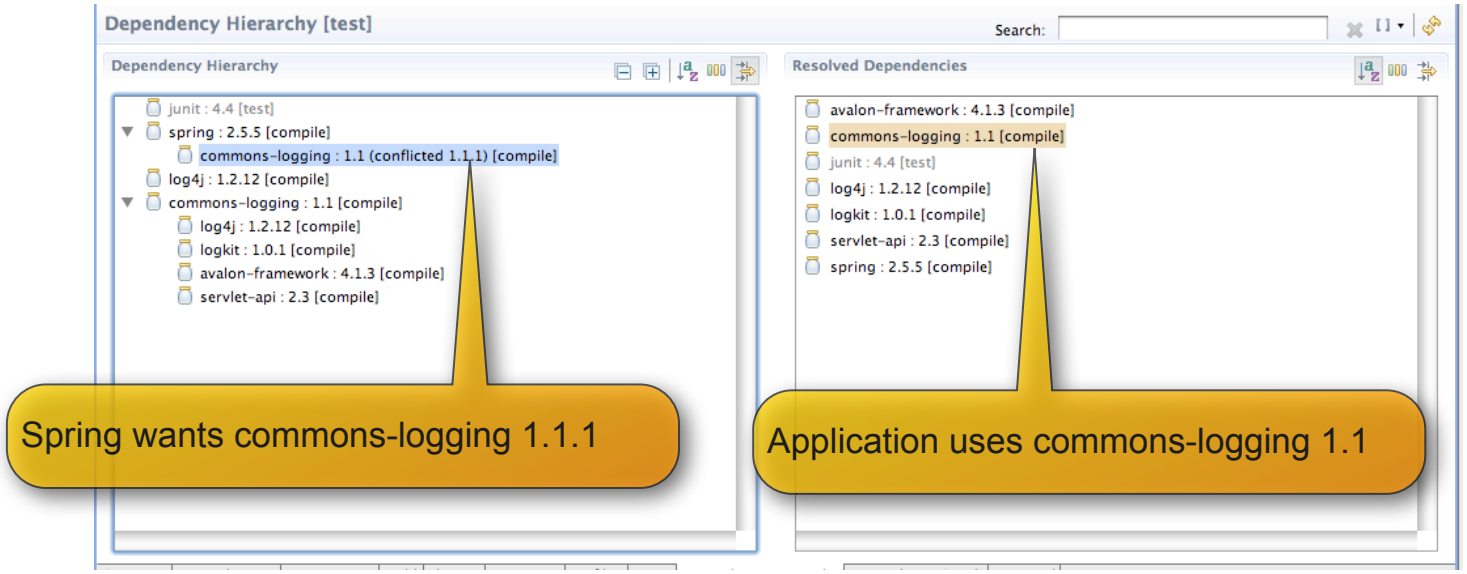
Hibernate libraries are declared but not used

Dependency Conflicts

- Dependency Conflicts
  - Different libraries require diversion versions of the same dependency
  - By default:
    - The nearest dependency to the top wins
    - The first dependency declared at a given level wins
  - Sometimes, we need to override the default behavior

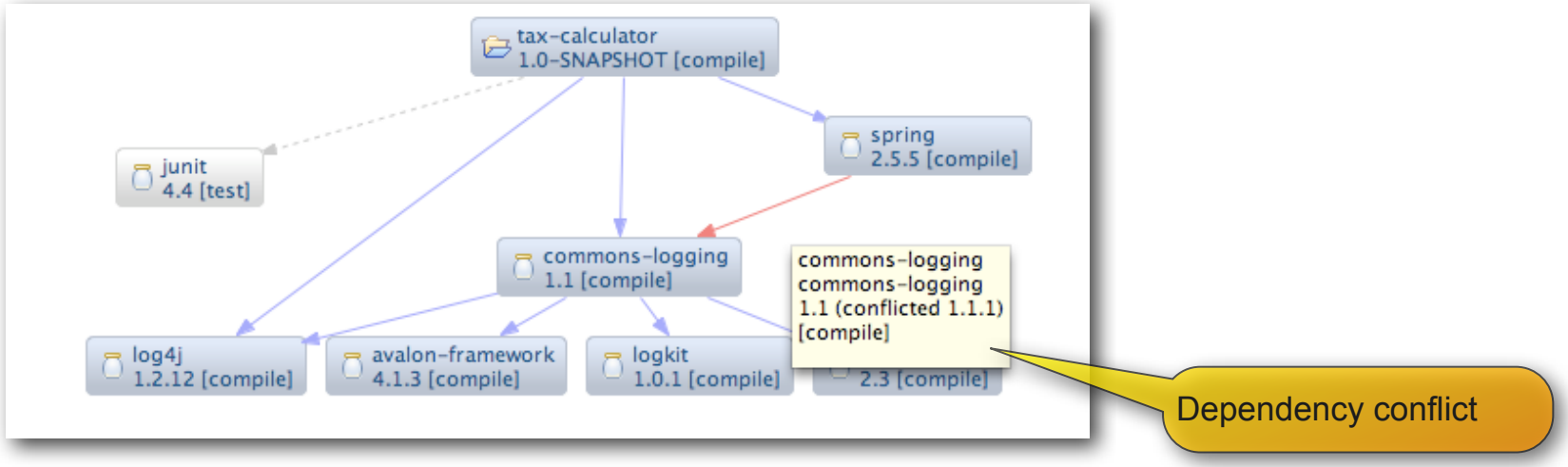
Dependency Conflicts

- Dependency Conflicts
  - You can visualize conflicts in the Dependency Hierarchy view



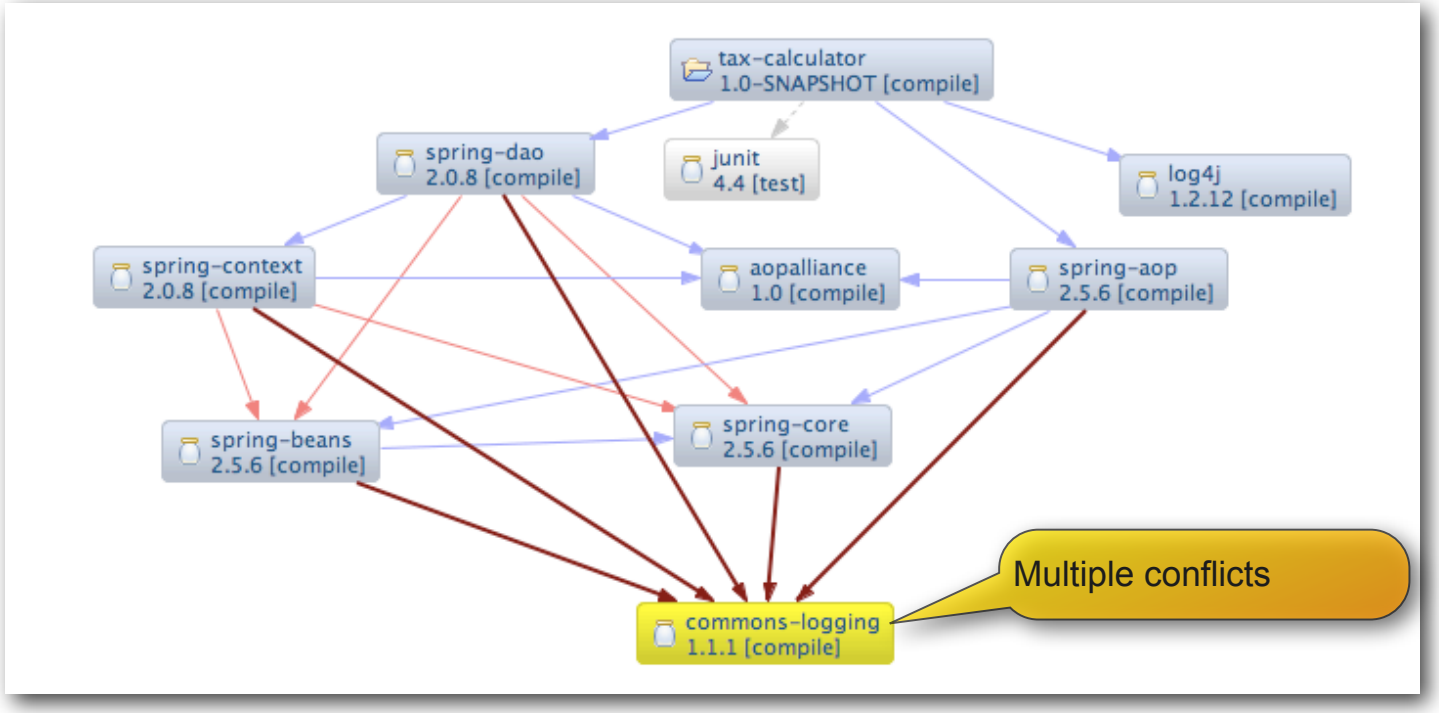
Dependency Conflicts

- Dependency Conflicts
  - Or in the Dependency Graph



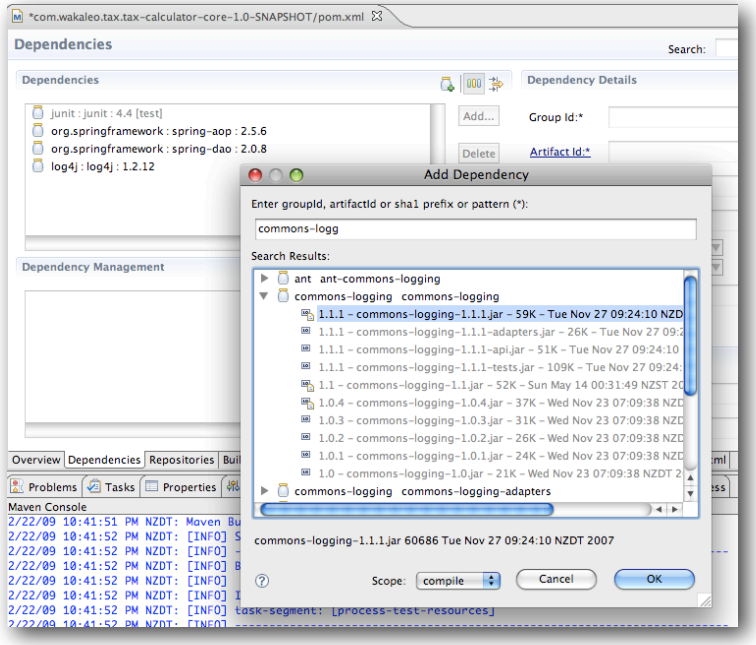
Dependency Conflicts

- Dependency Conflicts
  - A more complicated case:



Dependency Conflicts

- Dependency Conflict quick fix:
  - Declare the correct version in your POM file



  - Or exclude the unwanted version explicitly

Excluding Transitive Dependencies

- Excluding Dependencies lets you
  - Override normal transitive dependency management
  - Exclude certain libraries that would normally be transitively included

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
    <version>2.5.5</version>
    <exclusions>
     <exclusion>
      <groupId>javax.jms</groupId>
      <artifactId>jms<artifactId>
    </exclusion>
   </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jms_1.1_spec</artifact>
    <version>1.1</version>
  </dependency>
<dependencies>
```

Exclude the javax.jms dependency

Use the Apache Geronimo JMS specs instead

Grouping Dependencies

- Possibility to group dependencies together
  - logically grouped dependencies
  - define in a separate POM project
  - declare a dependency on this POM artifact

Grouping Dependencies

- Define logically grouped dependencies together

```
<project>
  <description>Project requiring JDBC</description>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>se.devoteam.maven.jfokus</groupId>
      <artifactId>persistence-deps</artifactId>
      <version>1.0</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</project>
```

```
<project>
  <groupId>se.devoteam.maven.jfokus</groupId>
  <artifactId>persistence-deps</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.2.5.ga</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-hibernate3</artifactId>
      <version>2.0.6</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1</version>
    </dependency>
  </dependencies>
</project>
```

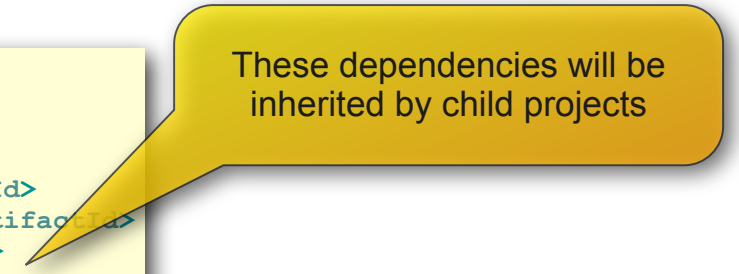
Grouping Dependencies

- Kind of a hack
- The drawback:
  - The dependencies are pushed down one level
  - Could affect conflict resolution
  - *mvn dependency:analyze* will not work

Inherited Behavior

- Inheriting common dependencies
  - Shared dependencies can be placed in the parent `pom.xml`
    - More consistent dependencies
    - Reduced repetition
    - Easier to maintain

```
<project...>
    ...
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.5</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.15</version>
        </dependency>
    </dependencies>
    ...
</project>
```



Inherited Behavior

- Using DependencyManagement to inherit dependencies
  - Use `<dependencyManagement>` to centralizes version numbers
    - Declare the official version numbers in a parent POM file
    - Only declare the artifacts in the child projects

```
<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</artif
        <version>5.1.2</version>
      </dependency>
      ...
```

Official version number

Only declare the artifact here

```
<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>
```

Inherited Behavior

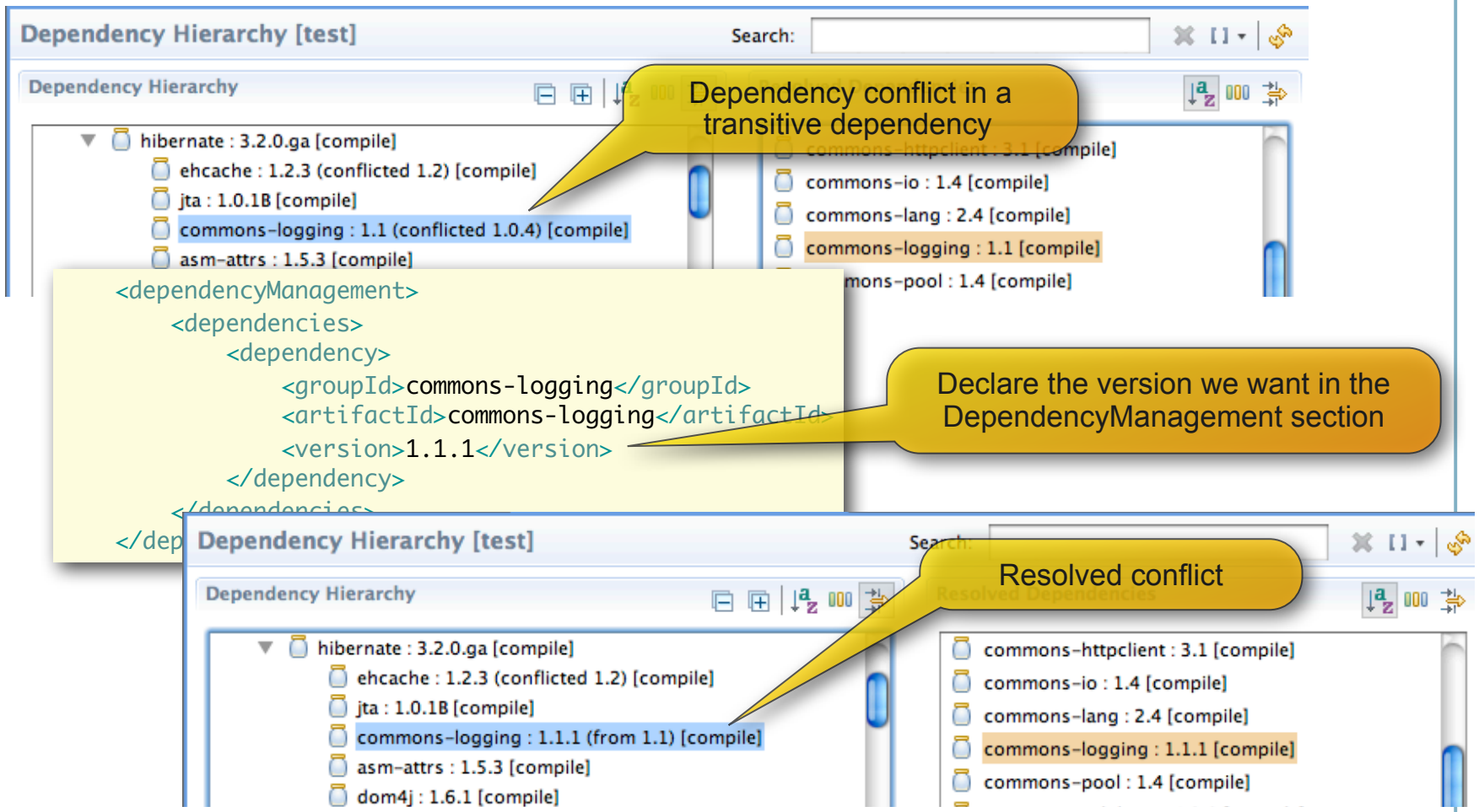
- Using DependencyManagement to inherit dependencies
  - The `<dependencyManagement>` section lists dependency version numbers
  - It does *not* add any dependencies to the project

Overriding Transitive Dependencies

- DependencyManagement with Transitive Dependencies
  - `<dependencyManagement>` also applies to Transitive Dependencies
    - This can be used to override versions of dependencies you don't directly depend on.
    - For example, if a specific version of a logger conflicts with your application server or has a known bug.

Overriding Transitive Dependencies

- DependencyManagement with Transitive Dependencies



Demo

- Dependency Management

Dependency Scope

- Import scope
  - Only works with Maven 2.0.9 onwards 
  - Import dependencies in the `<dependencyManagement>` section of another project
  - Lets you import dependencyManagement info from several sources
  - Only used by project type pom (e.g. parent projects)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>se.devoteam.maven.jfokus</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactI
        <version>5.1.2</version>
      </dependency>
      ...
    <dependencies>
  </dependencyManagement>
```

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>se.devoteam.maven.jfokus</groupId>
  <artifactId>b-parent</artifactId>
  <version>1.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>se.devoteam.maven.jfokus</groupId>
        <artifactId>a-parent</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      ...
    <dependencies>
  </dependencyManagement>
```

Enforce Correct Dependencies

- Through a Maven Repository Manager
  - enforce dependencies centrally
- Maven Enforcer Plugin
  - control through rules in the build
  - `<bannedDependencies>`

Adapting the build process

Part 2 - Lifecycle Customization

A Standardized Lifecycle

- Maven provides standardized lifecycles for projects
  - Reduced learning curve between projects
  - Allows for standardized, repeatable builds across projects
  - Customized through choices in POMs

A Standardized Lifecycle

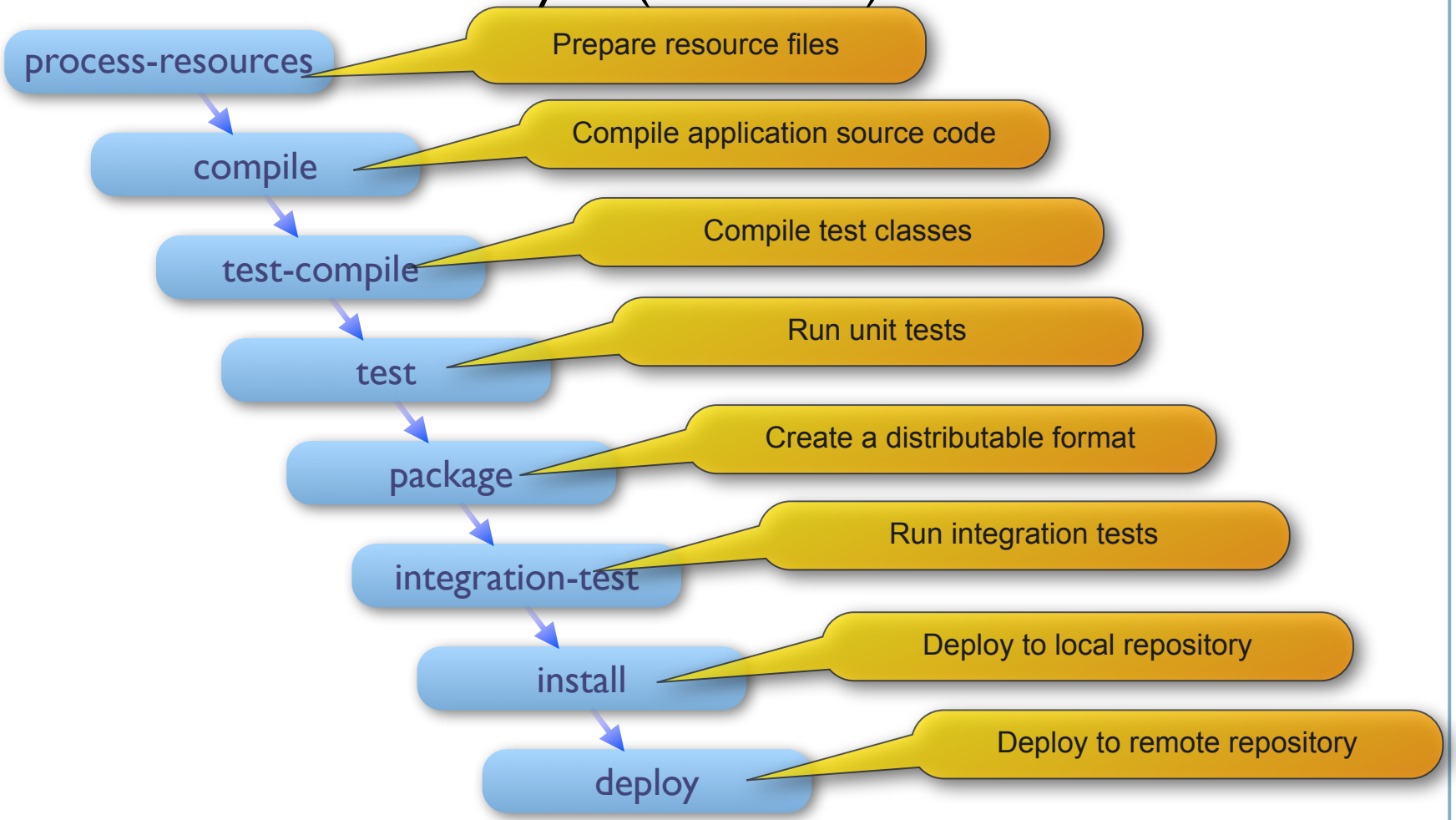
- Provides for many typical development steps:
  - Preparing source code for compilation
  - Compiling code
  - Running unit tests
  - Packaging applications
  - Running integration tests
  - Deploying to local and remote repositories

The Maven Lifecycle

- Maven provides the following lifecycles:
  - default lifecycle - project deployment
  - clean lifecycle - project cleaning
  - site lifecycle - project site creation

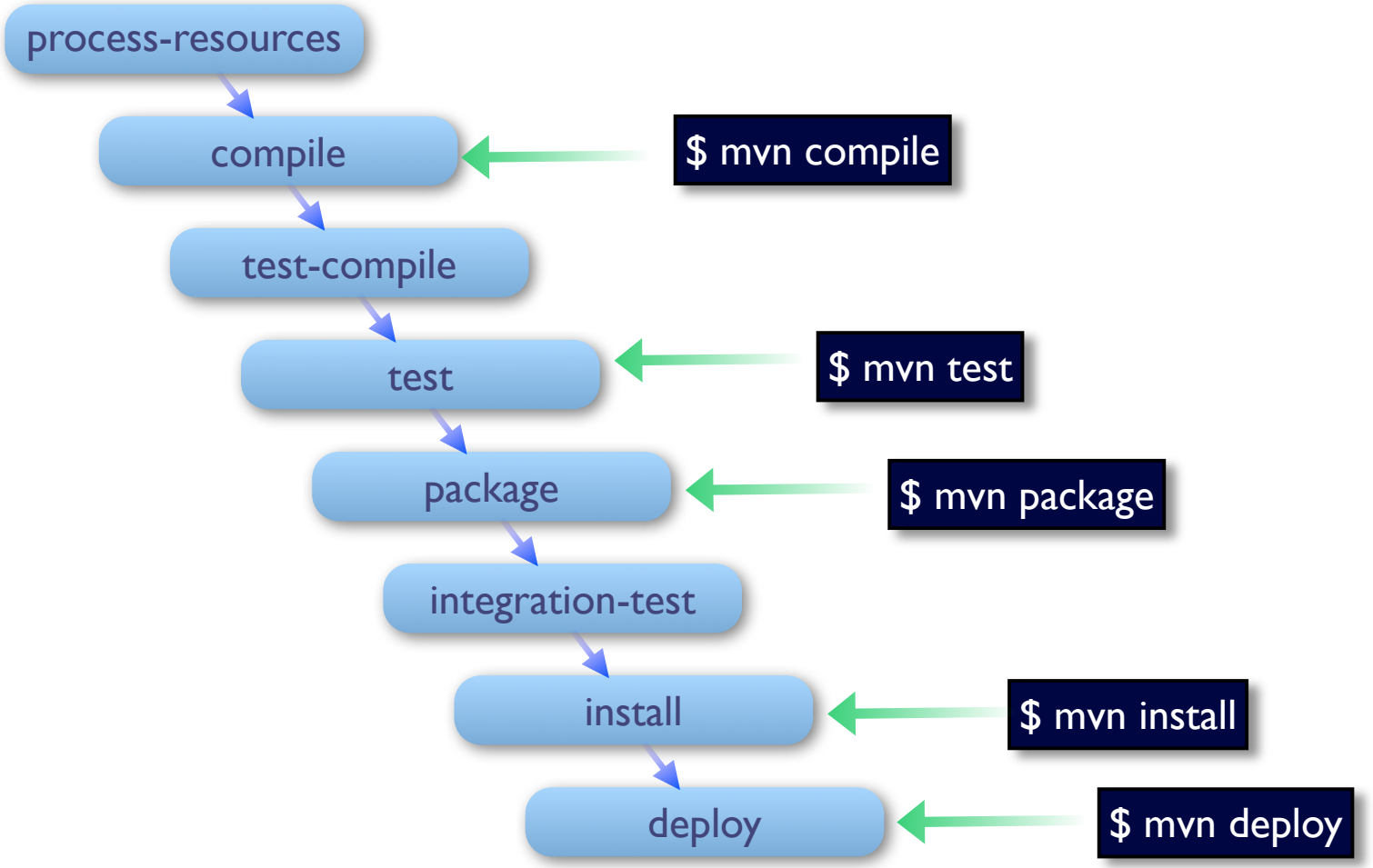
A Standardized Lifecycle

- The standard Maven lifecycle (an extract)



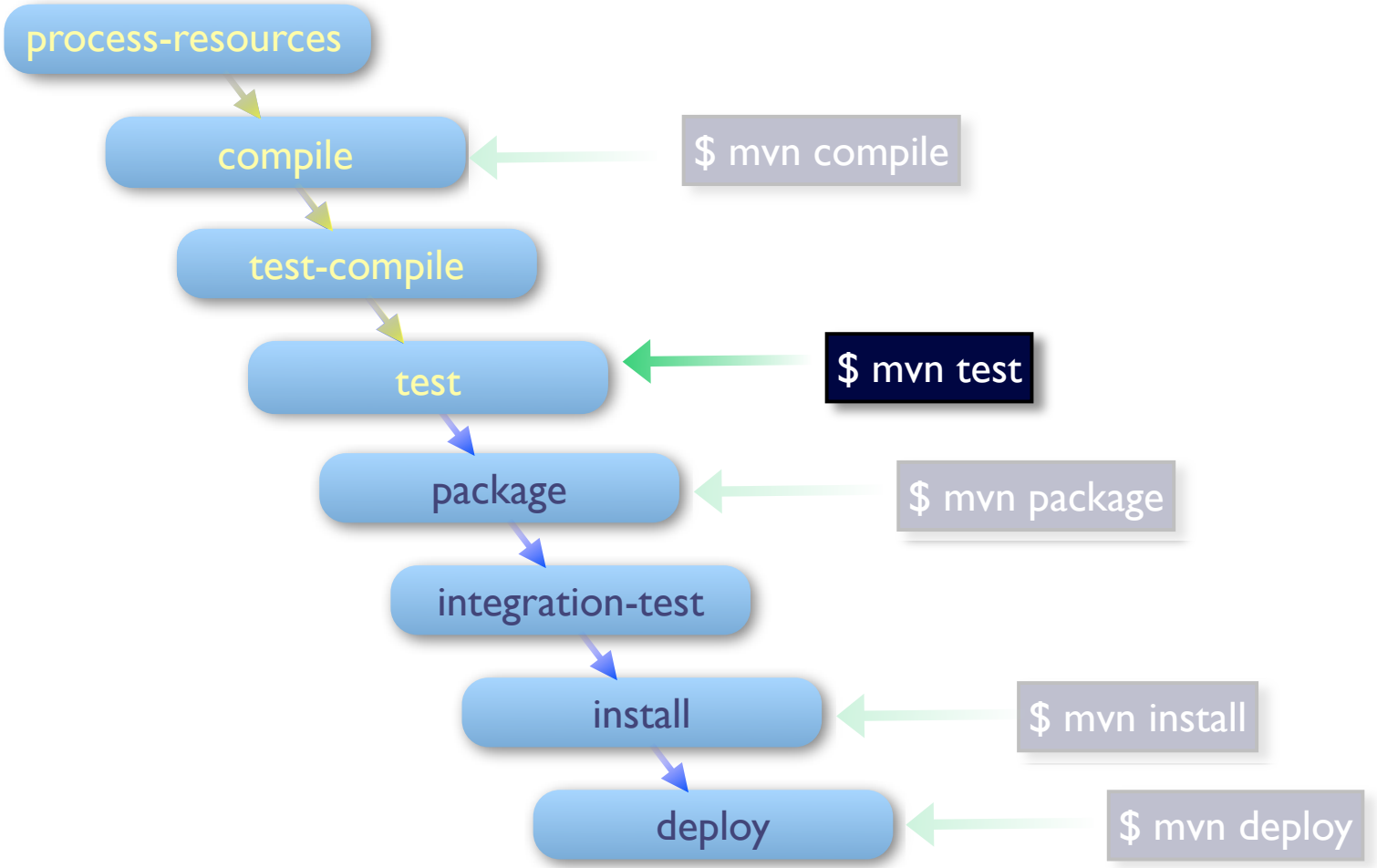
A Standardized Lifecycle

- You can invoke lifecycle phases directly...



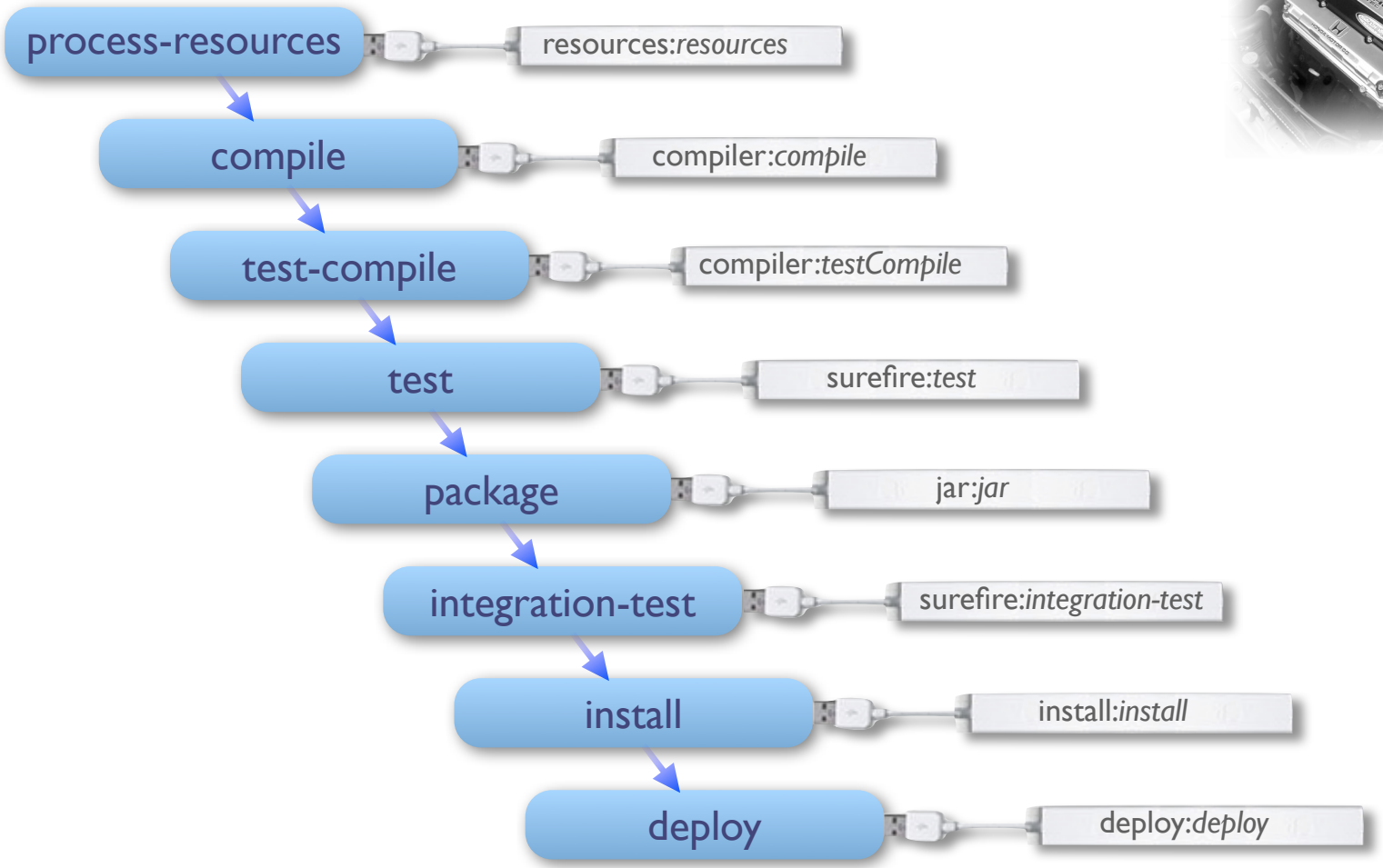
A Standardized Life Cycle

- Invoking a lifecycle phase will also invoke the previous phases



Maven Plugins and Goals

- Each lifecycle phase is implemented using plugins



Maven Plugins and Goals

- You can invoke a plugin in two ways:
  - Invoking a lifecycle phase

```
$ mvn compile
```

Invoke the **compile** lifecycle phase

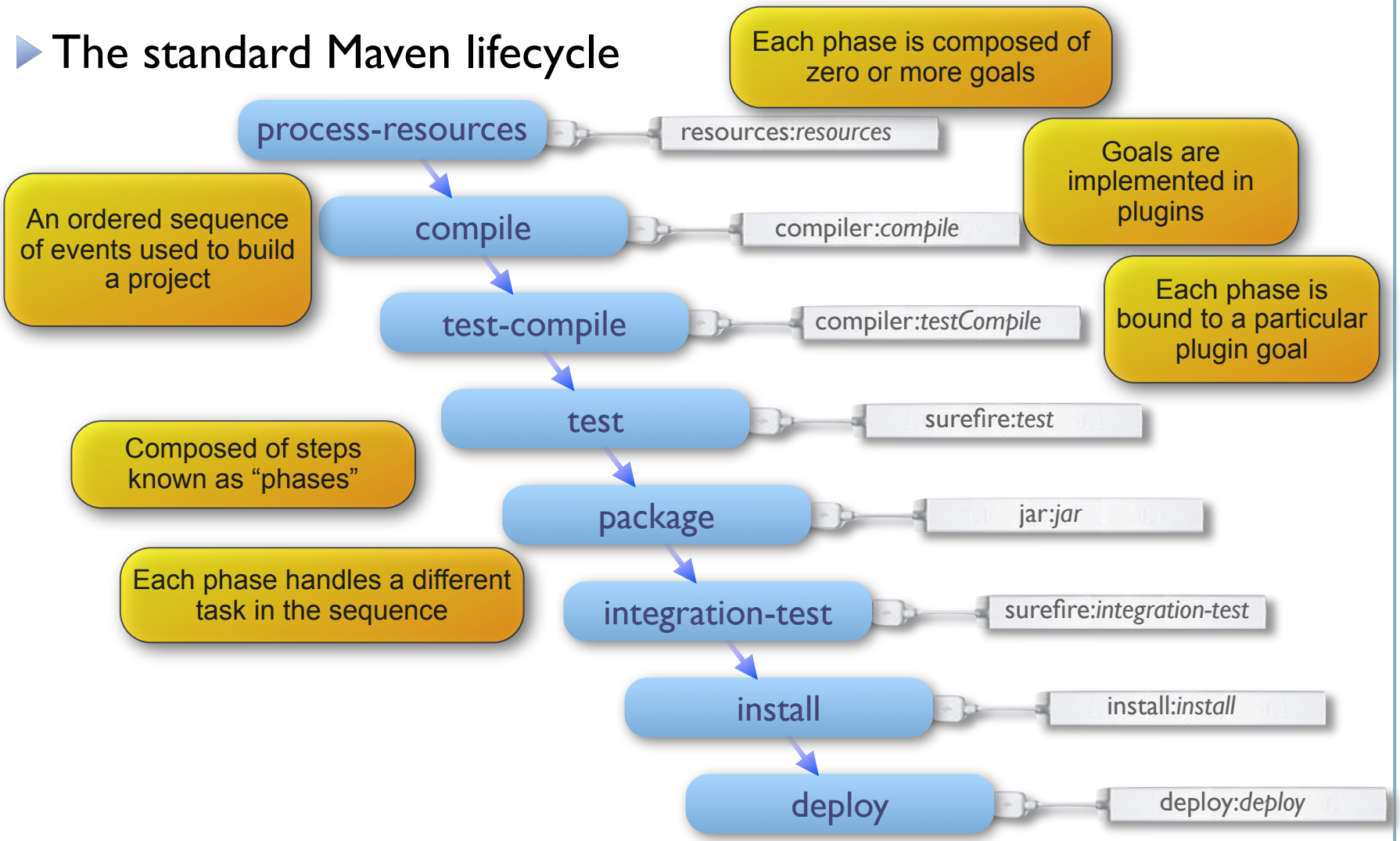
  - Invoking a plugin goal directly

```
$ mvn jar:test-jar
```

Invoke the **test-jar** goal of the **jar** plugin

Customizing the lifecycle

- The standard Maven lifecycle



Package-Specific Lifecycle Bindings

- There are specific lifecycle bindings for the following package types:
  - EAR
  - EJB
  - JAR
  - Maven Plugin
  - POM
  - WAR

Package-Specific Lifecycle Bindings

- The POM Lifecycle bindings
  - A project with packaging ear has a different set of default goals from a project with a packaging of jar or war

Lifecycle Phase	Goal
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

Package-Specific Lifecycle Bindings

- The JAR Lifecycle bindings
  - A project with packaging jar has a different set of default goals from a project with a packaging of war or ear

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

Package-Specific Lifecycle Bindings

- The WAR Lifecycle bindings
  - A project with packaging war has a different set of default goals from a project with a packaging of jar or ear

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

Package-Specific Lifecycle Bindings

- The EAR Lifecycle bindings
  - A project with packaging ear has a different set of default goals from a project with a packaging of jar or war

Lifecycle Phase	Goal
generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

Package-Specific Lifecycle Bindings

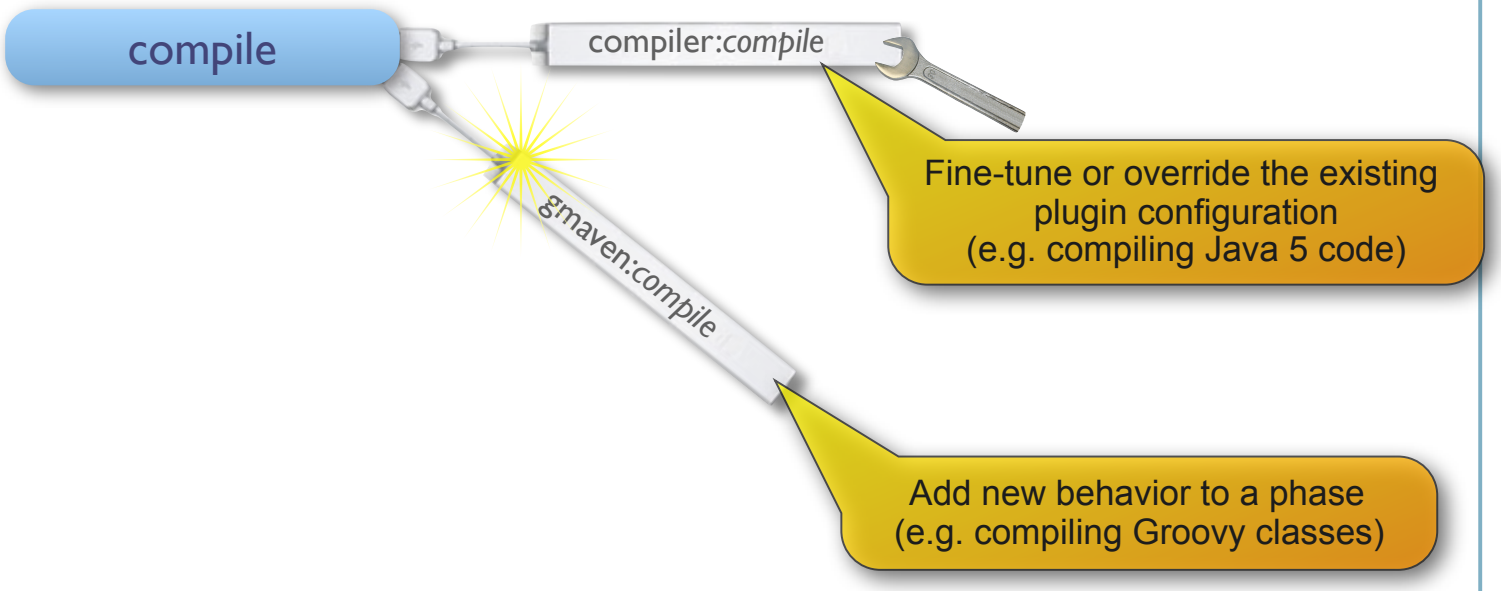
- Possible to create custom packaging types
- Define your own specific lifecycle bindings

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>se.devoteam.maven.jfokus</groupId>
  <artifactId>demo-sar</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jboss-sar</packaging>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jboss-packaging-maven-plugin</artifactId>
        <version>2.1.1</version>
        <!-- Enable packaging types and lifecycle bindings. -->
        <extensions>true</extensions>
      </plugin>
      ...
    </plugins>
  </build>
<project>
```

This configuration enables the custom package type

Customizing the lifecycle

- You can customize the lifecycle in two ways
  - Configure the standard plugin associated with a phase
  - Add a new plugin to add extra behavior to a phase



Customizing the lifecycle

- Customizing an existing configuration - compiling Java 5 code
  - Java compilation is done by the maven-compiler-plugin
  - This plugin is configurable:
    - Compiles sources for projects
    - Supports multiple compilers
    - Supports compiler options
    - Support pinning the compiler to a particular source and target version

Customizing the lifecycle

- An example of customization - compiling for Java 5

```
<project>
   ...
     <build>
       <pluginManagement>
         <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>
       <pluginManagement>
     </build>
   ...
</project>
```

Plugin configuration always goes in the <configuration> block

Here, compile for Java 5 code

Customizing the lifecycle

- Adding new behavior - executing a Groovy script
  - Maven and Groovy integrate well with the gmaven-plugin plugin
  - In this example we want to execute a Groovy script during the compile phase:

```
<plugin>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <executions>
        <execution>
            <phase>compile</phase>
            <goals>
                <goal>execute</goal>
            </goals>
            <configuration>
                <source>${pom.basedir}/src/main/script/myscript.groovy</source>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Add the gmaven-plugin

During the *compile* phase...

Call the plugin's *execute* goal

Plugin-specific configuration

Demo

- Lifecycle customization

Controlling the plugins

Part 3 - Plugin Management

Binding Inheritance

- Plugin bindings are inherited
  - similar to how dependencies work
  - convention - it is configurable

```
<plugin>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.2</version>
    <inherited>false</inherited>
    <executions>
        <execution>
            <phase>compile</phase>
            <goals>
                <goal>execute</goal>
            </goals>
            <configuration>
                ...
            </configuration>
        </execution>
    </executions>
</plugin>
```

This plugin binding is not inherited

Binding Inheritance

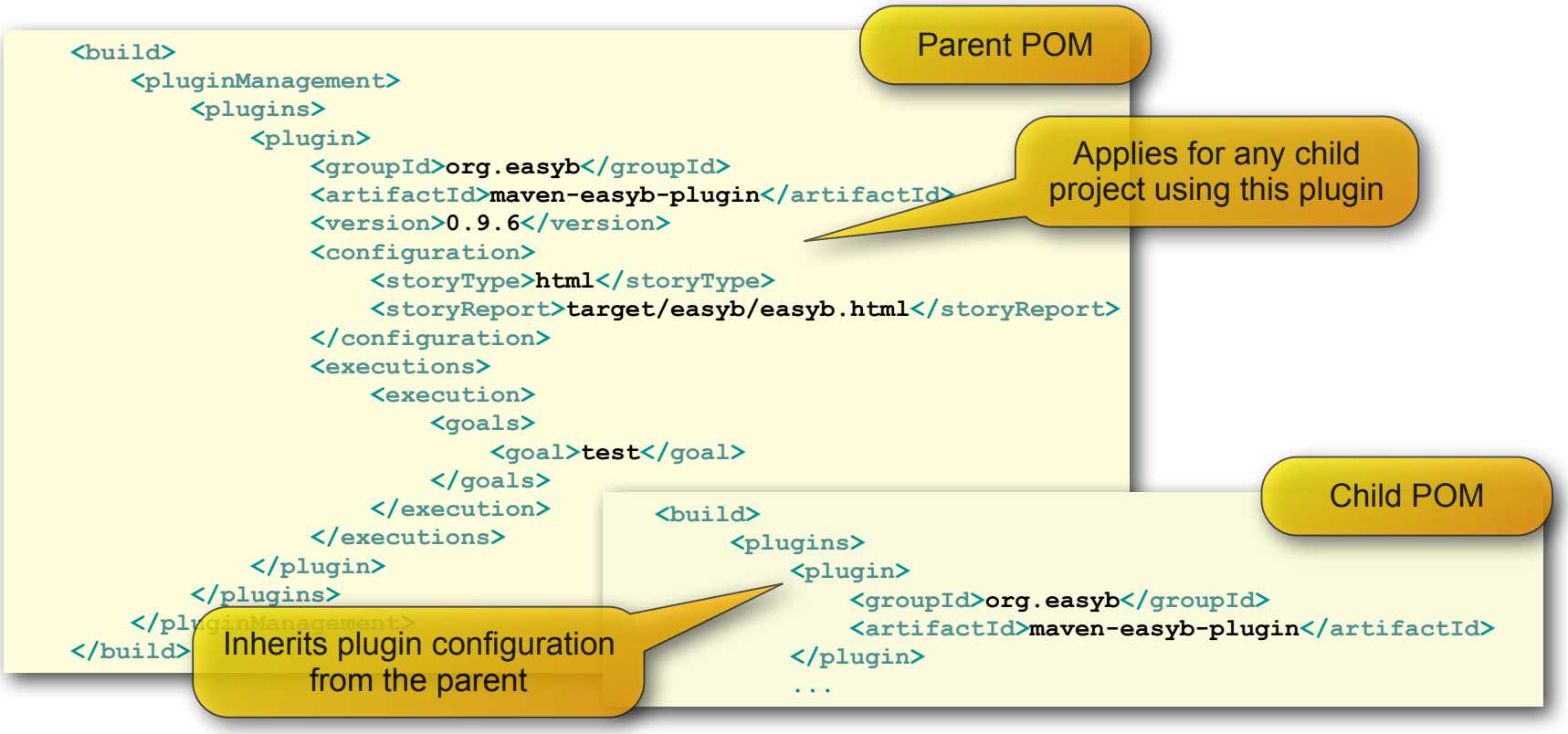
- Plugin binding inheritance is configurable
  - Where the binding is declared
  - Not where inherited (i.e. the child)

Demo

- Plugin binding inheritance

Plugin Management

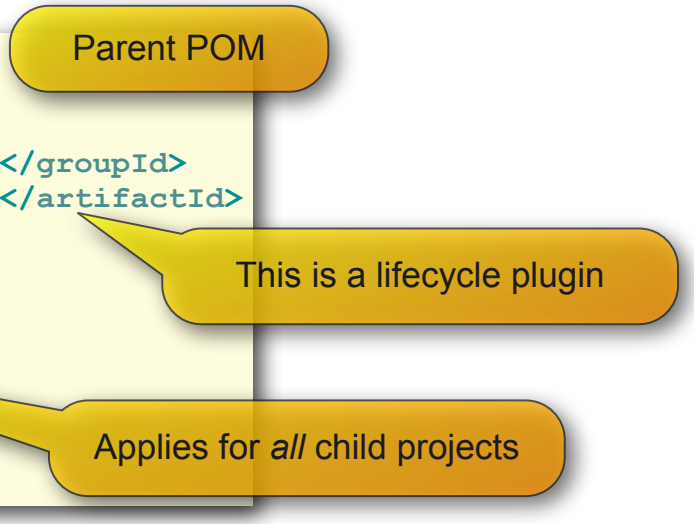
- Optimizing plugin dependencies
  - Similar to the `<dependencyManagement>` section
  - It does *not* add any new plugins to the project



Plugin Management

- Optimizing plugin dependencies
  - Lifecycle-related plugins apply to *all* child projects

```
<pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.1</version>
            <configuration>
                <source>1.5</source>
                <target>1.5</target>
            </configuration>
        </plugin>
    </plugins>
</pluginManagement>
```



Demo

- Plugin Management

Plugin Configuration

- Plugin configuration possible on two levels
  - plugin level
  - execution level

```
<plugin>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <configuration>
    <!-- Plugin level config -->
    <!-- This configuration applies to all executions -->
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>...</goal>
      </goals>
    <configuration>
      <!-- Execution level config -->
      <!-- Configuration specific to this execution -->
    </configuration>
    </execution>
  </executions>
</plugin>
```

Demo

- Plugin configuration