Maven Fundamentals

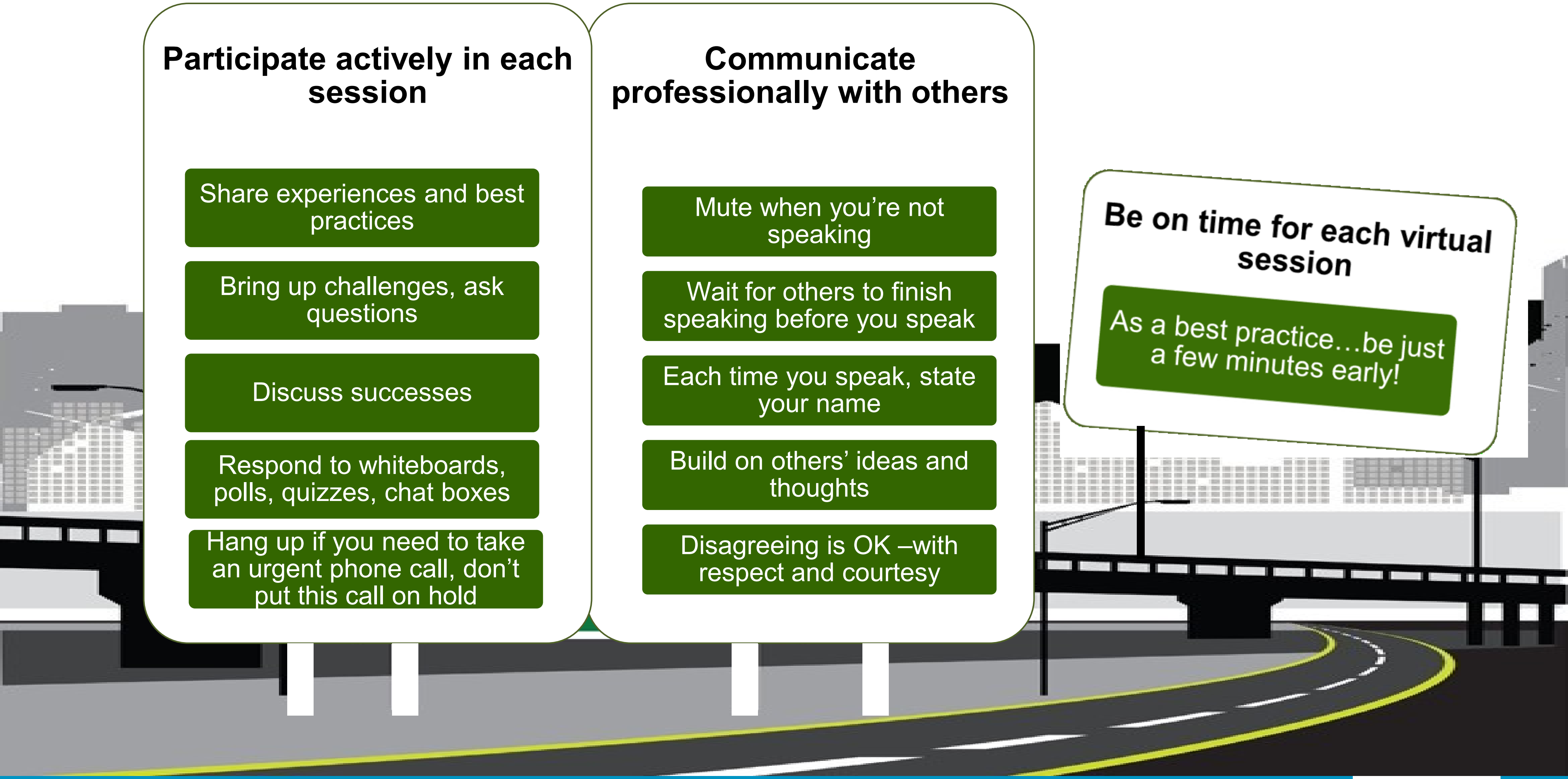
Feb 2016

People matter, results count.

Ground Rules for Face-to-face Classrooms

- Everyone participates
- Respect individual opinions and diversities
- Be open and honest
- Give headlines, be concise
- One speaker at a time
- Make language a non-issue
- Stick to time contracts
- Seek first to understand, and then to be understood
- Clean desk / room policy
- No mobile phone, No computer (except for surveys, polls etc)
- Clients & Leadership are often around, please remember that
- Maintain a spirit of fun and enthusiasm

Ground Rules for Virtual Classrooms



Participate actively in each session

- Share experiences and best practices
- Bring up challenges, ask questions
- Discuss successes
- Respond to whiteboards, polls, quizzes, chat boxes
- Hang up if you need to take an urgent phone call, don't put this call on hold

Communicate professionally with others

- Mute when you're not speaking
- Wait for others to finish speaking before you speak
- Each time you speak, state your name
- Build on others' ideas and thoughts
- Disagreeing is OK –with respect and courtesy

Be on time for each virtual session

- As a best practice…be just a few minutes early!

Module at a Glance

SME to provide the details required in the table.

Target Audience:	
Course Level:	*Basic*
Duration (in hours):	30 mins
Pre-requisites, if any:	NA
Post-requisites, if any:	*Submit Session Feedback*
Relevant Certifications:	None

Introductions (for Virtual Classrooms)

SME to provide the photos and names of the facilitators.

Business Photo

Business Photo

Facilitator
Name
Role

Moderator
Name
Role

Agenda

[redacted]

Module Objectives

Note to the SME : Please provide the module Objectives or validate the partially updated content

What you will learn

At the end of this module, you will learn:

- What is Maven

What you will be able to do

At the end of this module, you be able to:

- Understand what is Maven
- Differentiate between Ant and Maven
- State the key concepts of Maven
- Describe Dependencies in Maven
- List the Plug-ins used in Maven

Introduction to Maven

Outline

[redacted]

Maven High Level Overview

What is Maven?

At its simplest, Maven is a build tool.

- It always produces one artifact (component).
- It helps us manage dependencies.

It can also be used as a project management tool.

- It handles versioning and releases.
- Describes what your project is doing or what it produces.
- Can easily produce Javadocs as well as other site information.

Who Owns It?

- Maven is managed by the Apache Software Foundation
- Maven sites are built with Maven
- Open Source

Why do You Want to Use It?

Repeatable builds

- We can recreate our build for any environment

Transitive dependencies

- Downloading a dependency with also pull other items it needs

Contains everything it needs for your environment

Works with a local repo

Works with your IDE, but also standalone

The preferred choice for working with build tools like Jenkins or Cruise Control

Ant Vs. Maven

Ant

Ant was developed to be a replacement for a build tool called Make

Designed to be cross platform

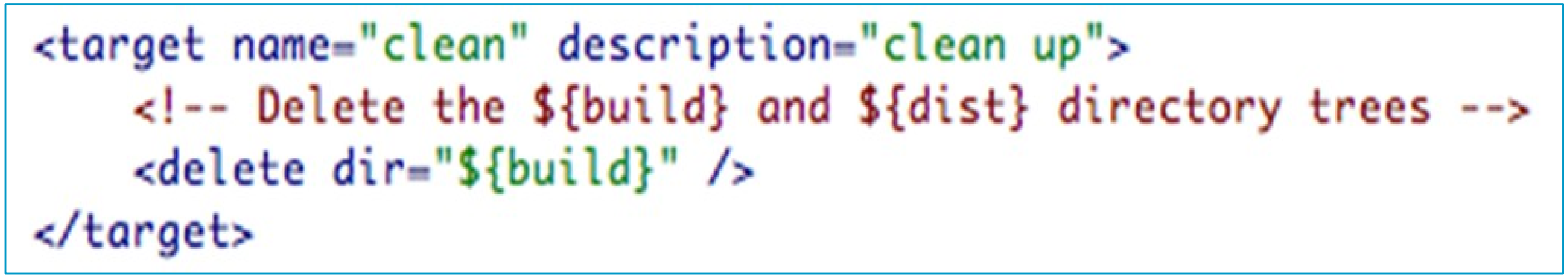
Built on top of Java and XML

Very procedural

Ant

Ant really isn't a build tool as much as it is a scripting tool.

You have to explicitly do everything in Ant.



```
<target name="clean" description="clean up">
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}" />
</target>
```

We have to define everything that we want to do clean, clear, cleanup, etc.

A lot of tribal knowledge, nothing carries over.

Maven

Maven is a full fledged build tool.

A lot of implicit functionality.

Consistency across projects.

Also capable to achieve inheritance in projects.

Transitive dependencies (can be achieved using Ivy with Ant though).

Built around versioning.

Pros and Cons

Pros	Cons
• Maven can be a black box • Steeper learning curve • Convention over configuration • Better IDE integration • Let overheard through use of repos • Different mindset, steepest learning curve is not making Maven act like Ant	• You can trace through Ant files fairly easily • Quicker to learn, but very copy-paste intensive • Larger project size in SCM, artifacts stored with project

Ant build.xml

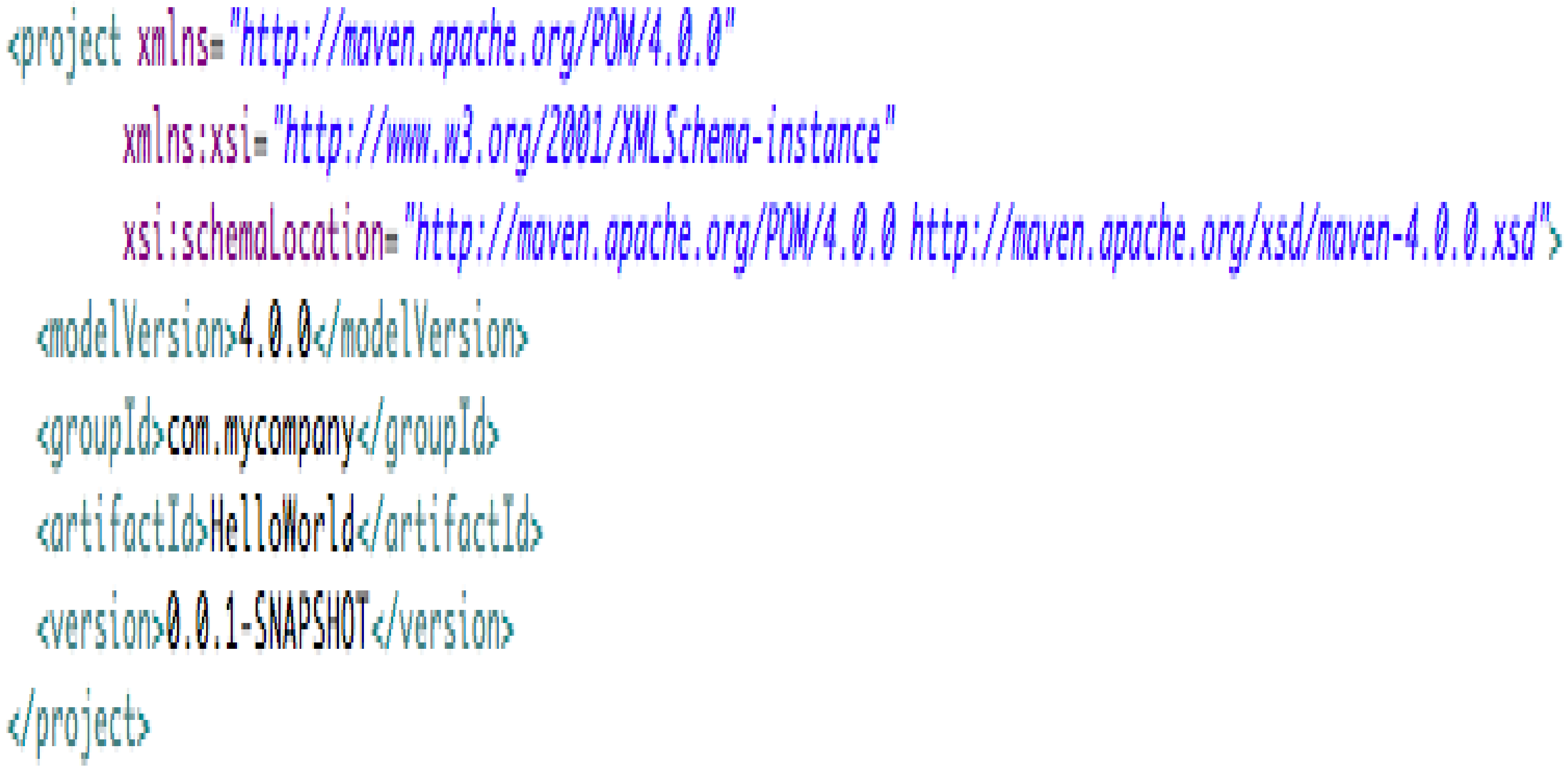
```
<project>
    <target name="clean">
        <delete dir="build"/>
    </target>

    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>

    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="oata.HelloWorld"/>
            </manifest>
        </jar>
    </target>

    <target name="run">
        <java jar="build/jar/HelloWorld.jar" fork="true"/>
    </target>
</project>
```

Maven pom.xml



```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>HelloWorld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

Ant vs. Maven

Ant	Maven
• Ant is very declarative. • Ant is maybe easier to learn, but it is only beneficial as a scripting tool.	• Maven follows a convention over configuration model. • Maven is really centered around managing your entire project's lifecycle.

Summary

What is Maven

How is Maven different from other tools

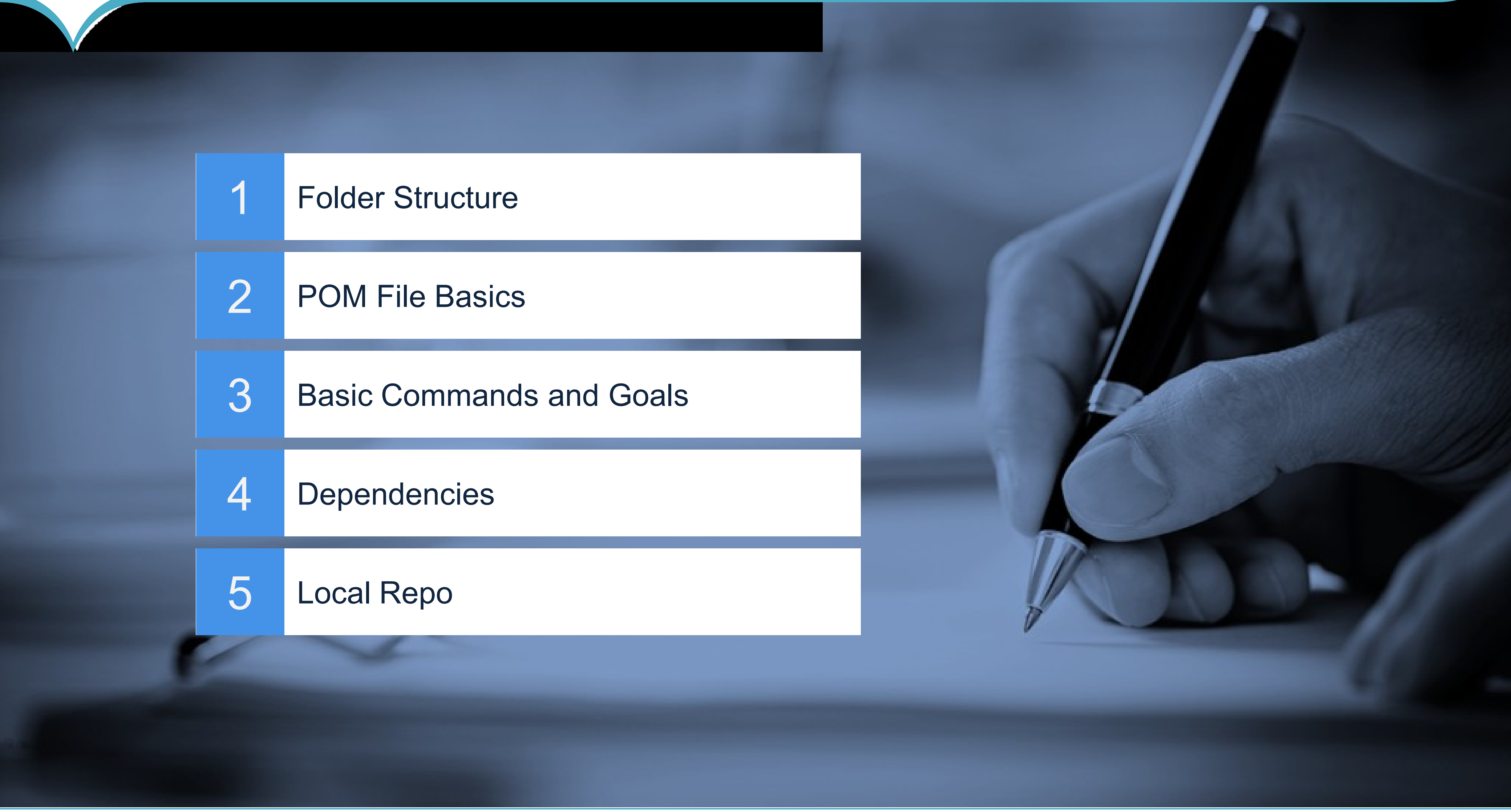
- Its not just a scripting tool

Where to get it and how to install it

A Hello World application

Structure

Outline

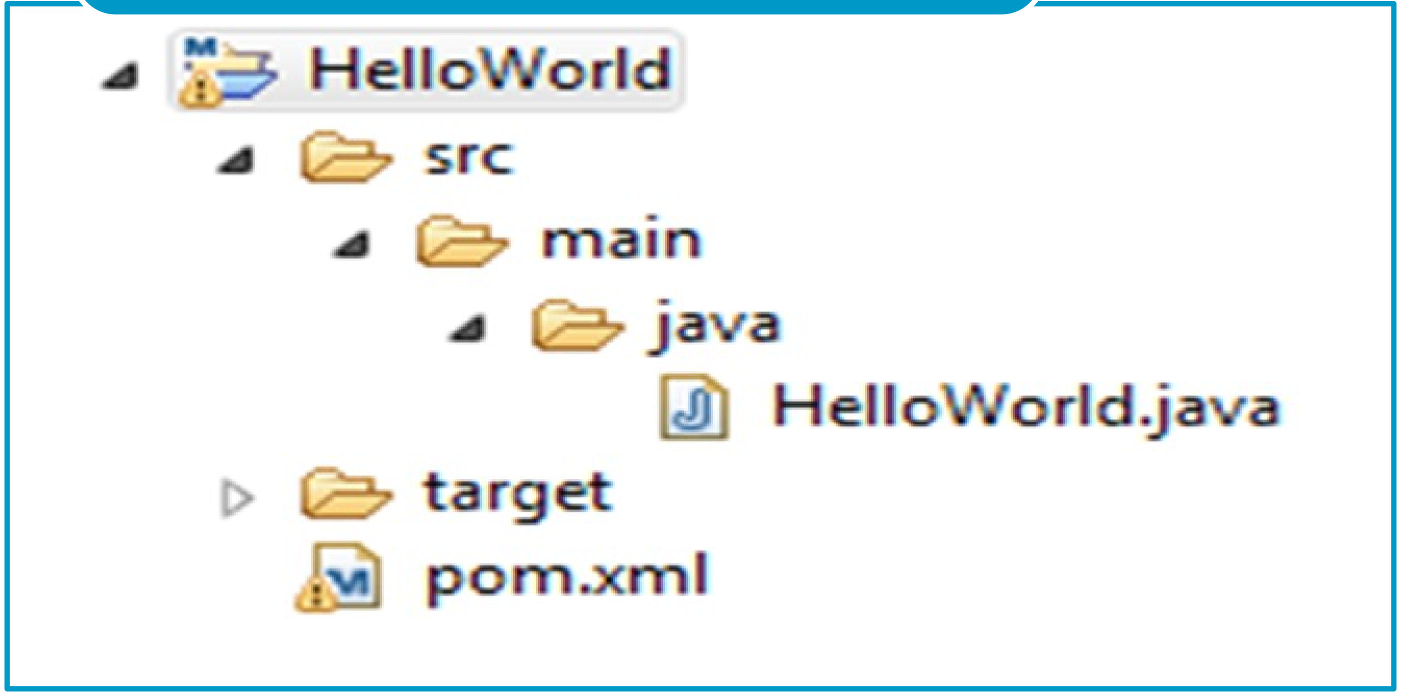


src/main/what?

src/main/java

target

pom.xml



```
◢ HelloWorld
    ◢ src
        ◢ main
            ◢ java
                HelloWorld.java
    ▷ target
      pom.xml
```

src/main/java

Where do we store our Java code?

The beginning of our package declaration:

- com.yourcompanyname.division

What about other languages

- src/main/groovy

What about testing

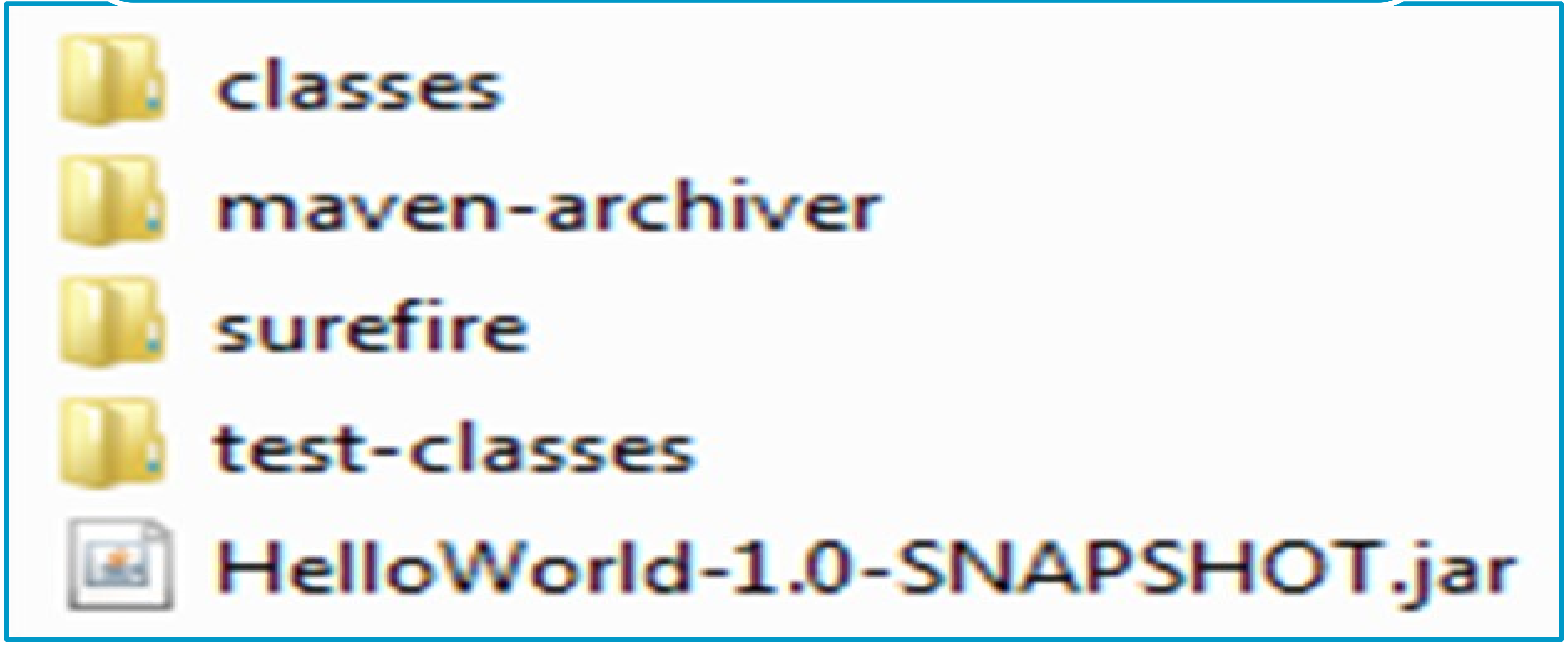
- src/test/java

Target

Where everything gets compiled to.

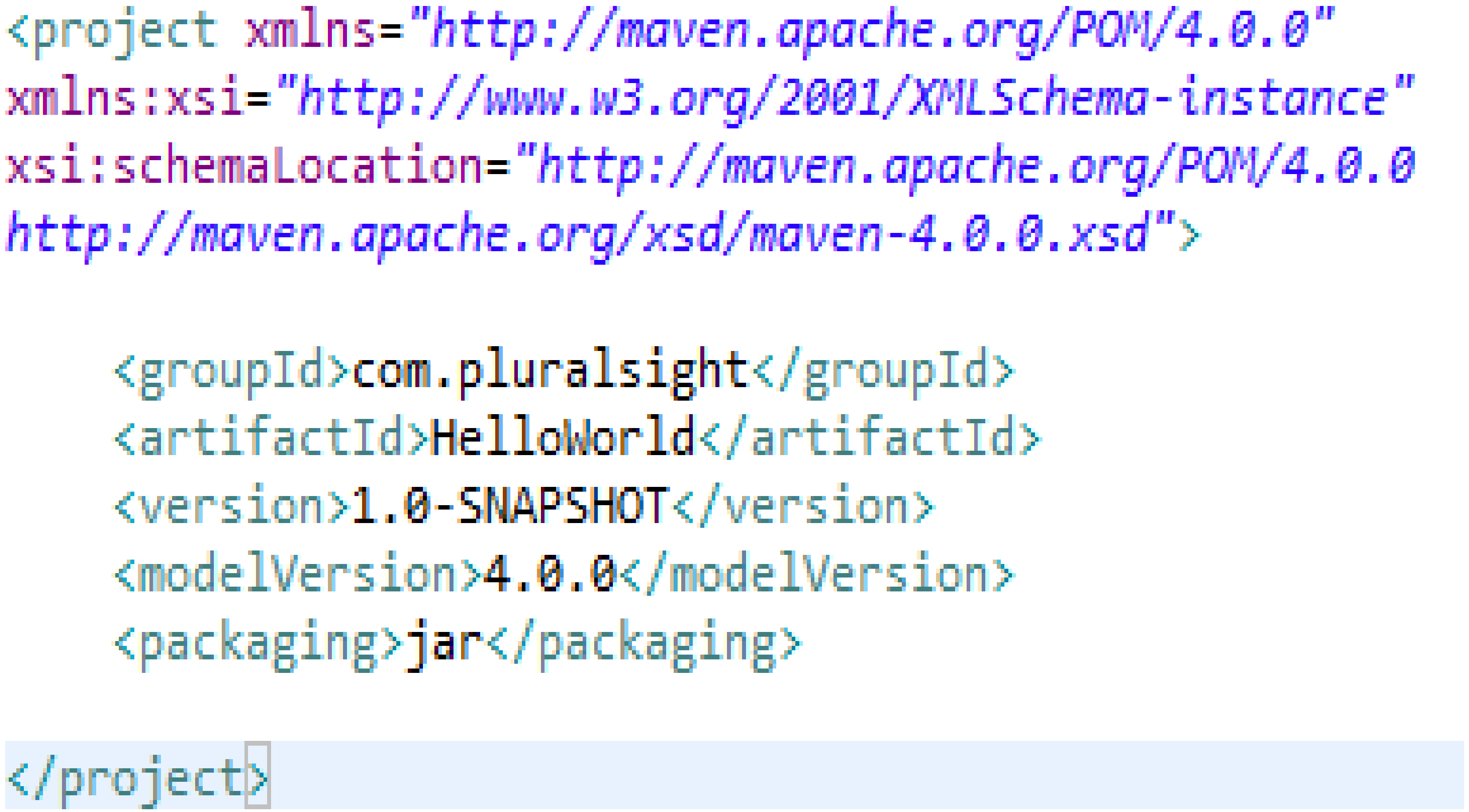
Also where tests get run from.

Contents in this directory get packaged into a jar, war, ear, etc.



classes
maven-archiver
surefire
test-classes
HelloWorld-1.0-SNAPSHOT.jar

pom.xml



```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <groupId>com.pluralsight</groupId>
    <artifactId>HelloWorld</artifactId>
    <version>1.0-SNAPSHOT</version>
    <modelVersion>4.0.0</modelVersion>
    <packaging>jar</packaging>

</project>
```

pom.xml (contd.)

Can be divided into 4 basic parts:

Project Information
- groupId
- artifactId
- Version
- Packaging

Dependencies
- Direct dependencies used in our application

Build
- Plugins
- Directory Structure

Repositories
- Where we download the artifacts from

Dependencies

What we want to use in our application.

Dependencies are imported by their naming convention.

- Often considered the most confusing part of Maven

We have to know the groupId, artifactId, and the version of what we are looking for.

Added to a dependencies section to our pom file.

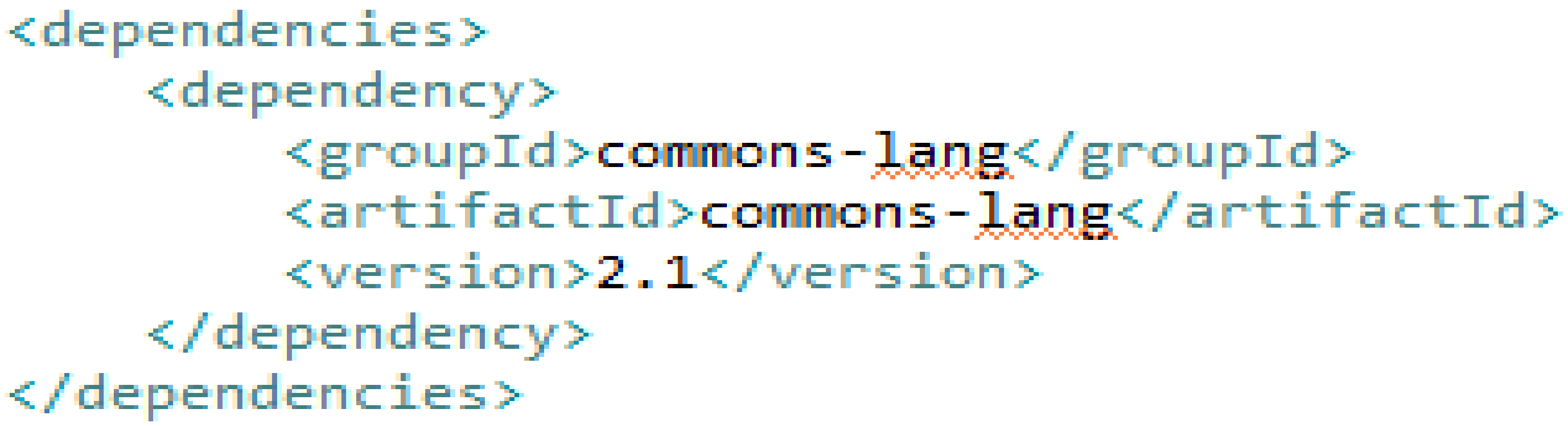
Dependencies (contd.)

Just list the dependency that we want.

- **Transitive dependencies will be pulled in by Maven.**

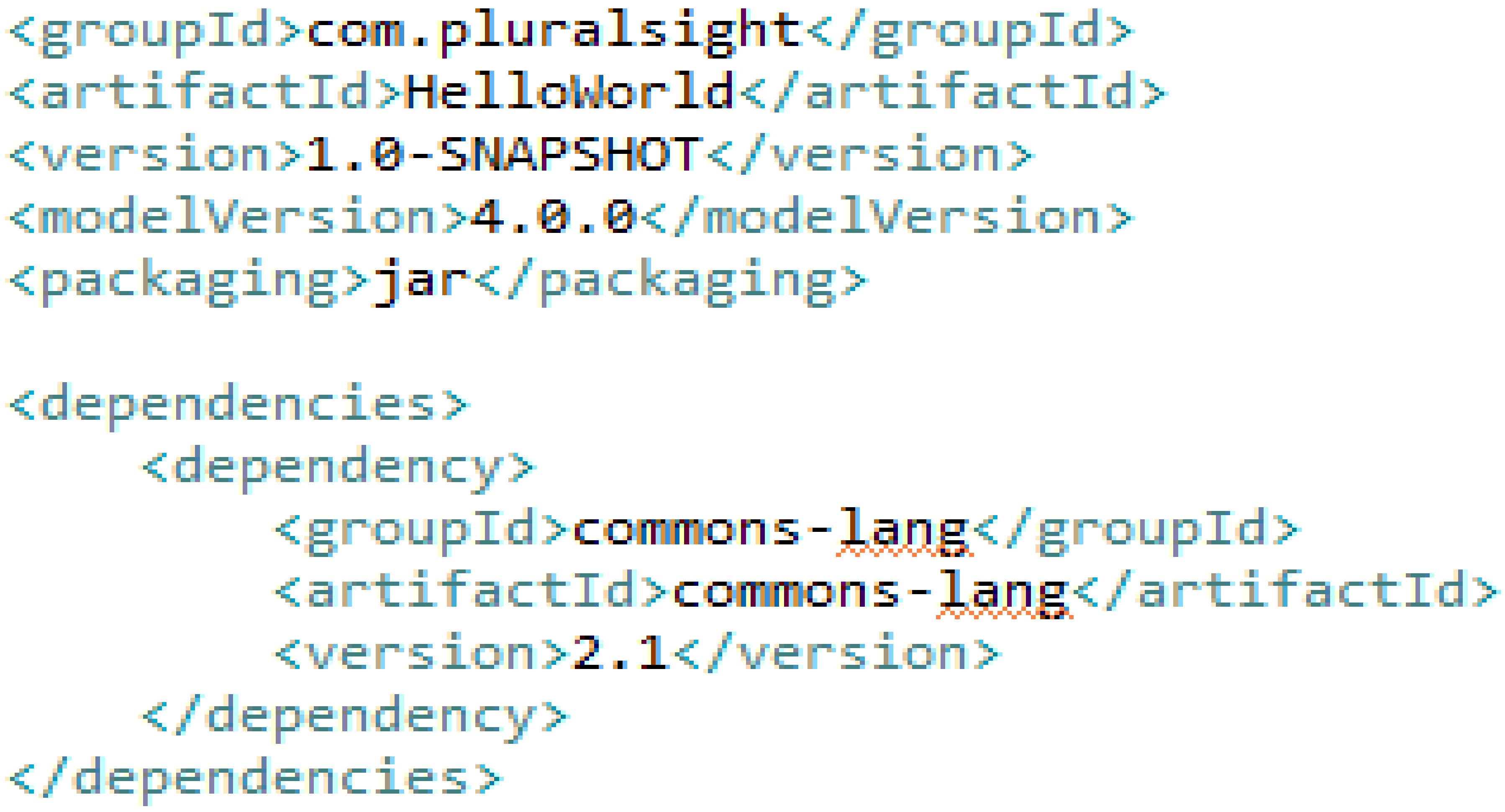
Need at a minimum 3 things:

- **groupId**
- **artifactId**
- **version**



```
<dependencies>
    <dependency>
        <groupId>commons-lang</groupId>
        <artifactId>commons-lang</artifactId>
        <version>2.1</version>
    </dependency>
</dependencies>
```

pom.xml With Our New Dependency



```xml
<groupId>com.pluralsight</groupId>
<artifactId>HelloWorld</artifactId>
<version>1.0-SNAPSHOT</version>
<modelVersion>4.0.0</modelVersion>
<packaging>jar</packaging>

<dependencies>
    <dependency>
        <groupId>commons-lang</groupId>
        <artifactId>commons-lang</artifactId>
        <version>2.1</version>
    </dependency>
</dependencies>
```

Goals

Clean	• Deletes the target directory and any generated resources.
Compile	• Compiles all source code, generates any files, copies resources to our classes directory.
Package	• Runs the compile command first, runs any tests, packages the app based off of its packaging type.
Install	• Runs the package command and then installs it in your local repo.
Deploy	• Runs the install command and then deploys it to a corporate repo. • Often confused with deploying to a web server.

Local Repo

Where Maven stores everything it downloads

Installs in your home directory\.m2

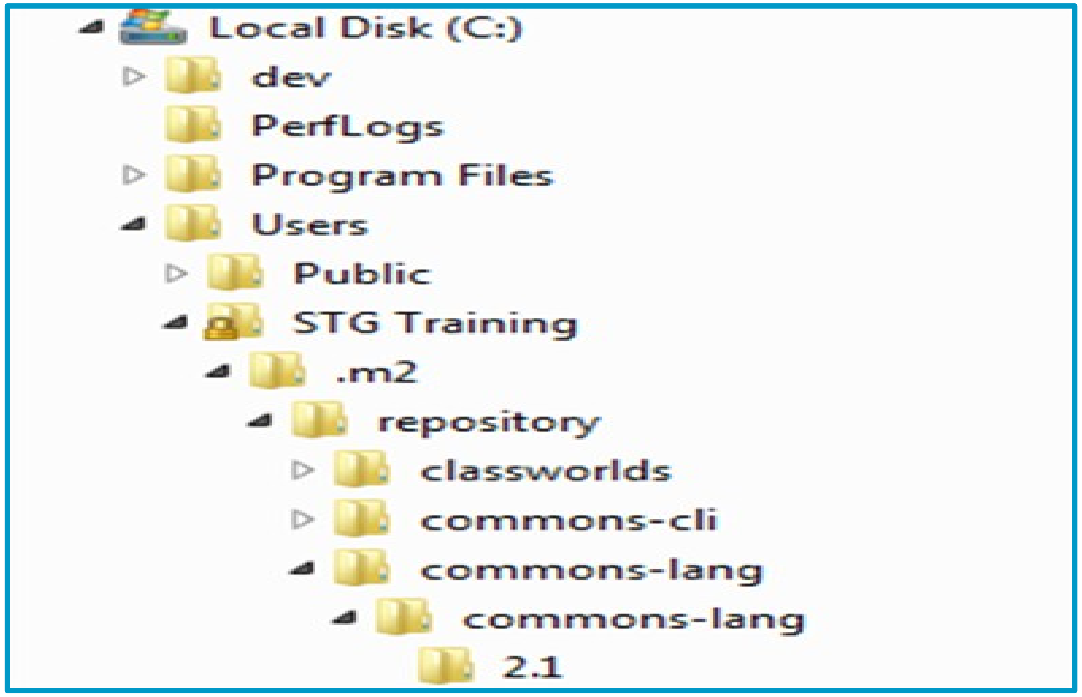
- C:\Users\<yourusername>\.m2\repository

Stores artifacts using the information that you provided for artifactId, groupId, and version

- C:\Users\<yourusername>\.m2\repository\commons-lang\commons-lang\2.1\commons-lang-2.1.jar

Local Repo (contd.)

Avoids duplication by copying it in every project and storing it in your SCM.



Defaults

These are all the defaults that Maven has, but how do we override them?

- The build section!
- Let’s Demo the options for structure changes.

Summary

Source code goes in src/main/java

Everything is compiled to our target directory.

The pom has 4 major parts.

Introduction of goals You can chain goals.

Basic example of a dependency.

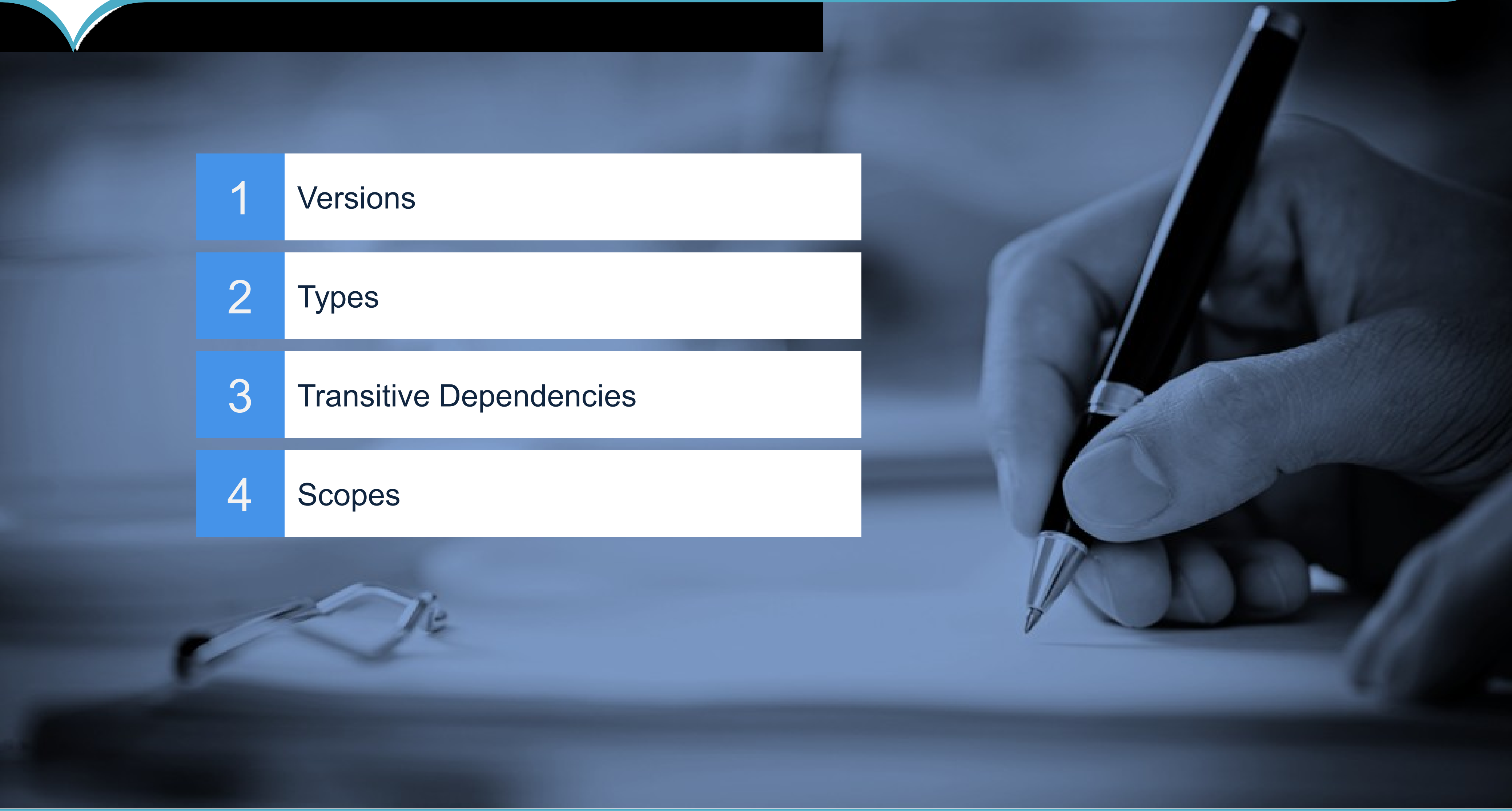
Where things are stored in your local repo.

How we can override the default behavior.

Dependencies

Outline

[redacted]



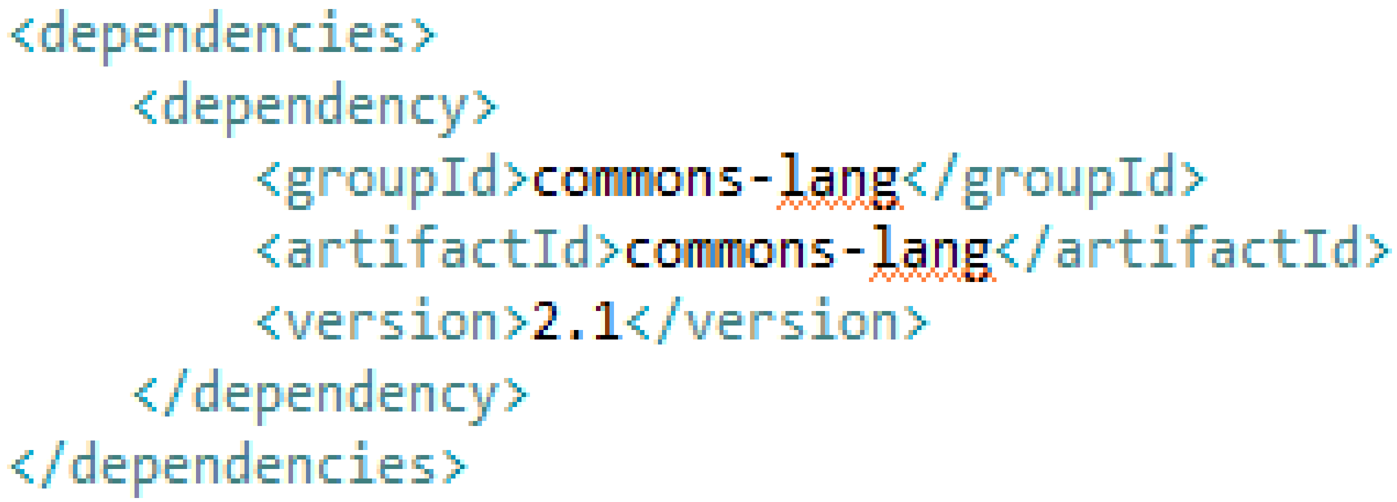
Dependency

Just list the dependency that we want.

- **Transitive dependencies will be pulled in by Maven.**

Need at a minimum 3 things:

- **groupId**
- **artifactId**
- **version**



```
<dependencies>
    <dependency>
        <groupId>commons-lang</groupId>
        <artifactId>commons-lang</artifactId>
        <version>2.1</version>
    </dependency>
</dependencies>
```

Versions

Development starts off as a SNAPSHOT

- myapp-1.0-SNAPSHOT.jar
- Changes always downloaded
- Saves you from rereleasing versions for development
- Never deploy to production with a SNAPSHOT

A release doesn't have a specific naming convention

- myapp-1.0.jar
- myapp-1.0.1.jar

Industry common terms, but don't affect maven

- myapp-1.0-M1.jar (milestone release)
- myapp-1.0-RC1.jar (release candidate)
- myapp-1.0-RELEASE.jar (release)
- myapp-1.0-Final.jar (release)

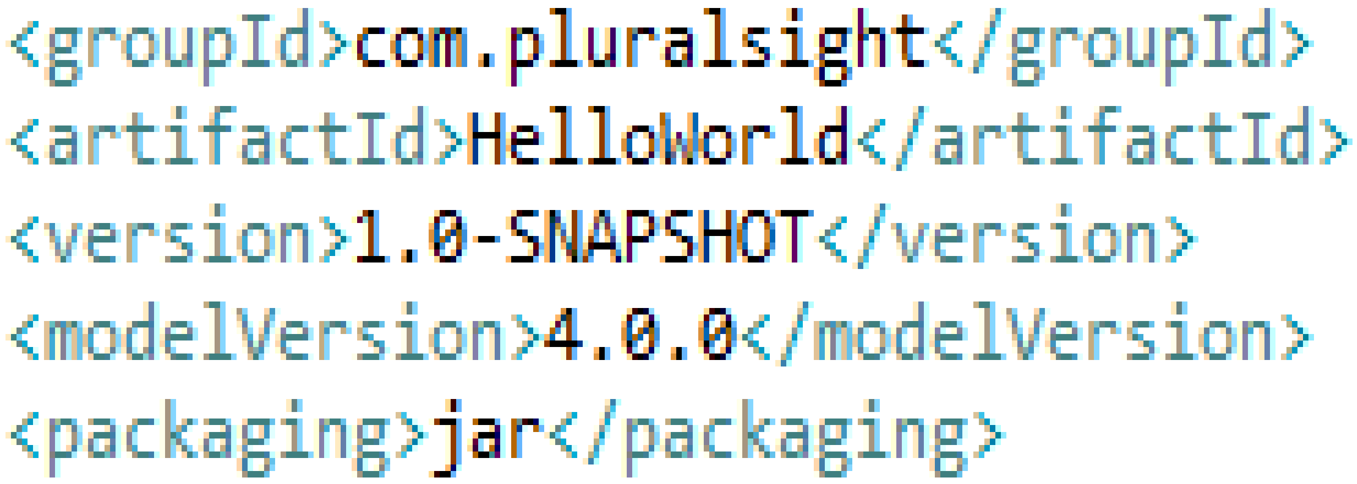
Types

Current core packaging types are:

- **pom, jar, maven-plugin, ejb, war, ear, rar, par**
- The default packaging type is jar.

The type of pom is referred to as a dependency pom.

- **Downloads dependencies from that pom**

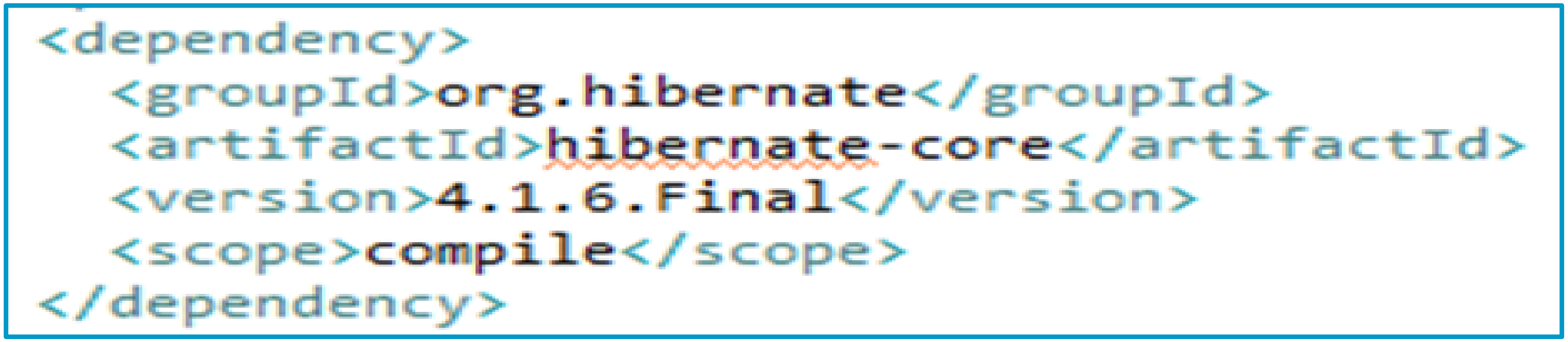


```
<groupId>com.pluralsight</groupId>
<artifactId>HelloWorld</artifactId>
<version>1.0-SNAPSHOT</version>
<modelVersion>4.0.0</modelVersion>
<packaging>jar</packaging>
```

Transitive Dependencies

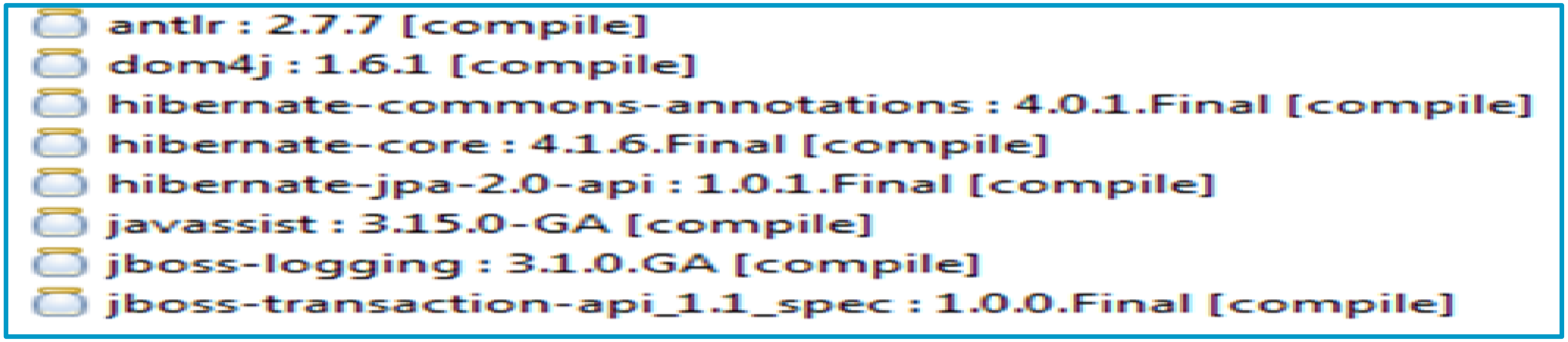
The main reason people begin using maven

If we add a dependency:



```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.1.6.Final</version>
  <scope>compile</scope>
</dependency>
```

If we add a dependency it downloads it's transitive dependencies



```
antlr : 2.7.7 [compile]
dom4j : 1.6.1 [compile]
hibernate-commons-annotations : 4.0.1.Final [compile]
hibernate-core : 4.1.6.Final [compile]
hibernate-jpa-2.0-api : 1.0.1.Final [compile]
javassist : 3.15.0-GA [compile]
jboss-logging : 3.1.0.GA [compile]
jboss-transaction-api_1.1_spec : 1.0.0.Final [compile]
```

Scopes

There are 6 scopes available for dependencies: compile – default scope, artifacts available everywhere.

provided	• Like compile, means that the artifact is going to be provided where it is deployed servlet-api.jar.
xml-apis	• Available in all phases, but not included in final artifact.
runtime	• Not needed for compilation, but needed for execution Not available for compilation, but included in all other phases. Not included in final artifact.
test	• Only available for the test compilation and execution phase.
system	• Similar to provided, you specify a path to the jar on your file system. Very brittle and defeats the purpose of maven, don’t use!
import	• Advanced topic, deals with dependency Management sections.

Demo

Let’s look at what the pom.xml will look like when we:

- Add a dependency
- Reference transitive dependencies
- Use scopes

Summary

Version numbers are up to your corporate strategy. SNAPSHOT has a unique meaning.

Supported types and what they mean to your application.

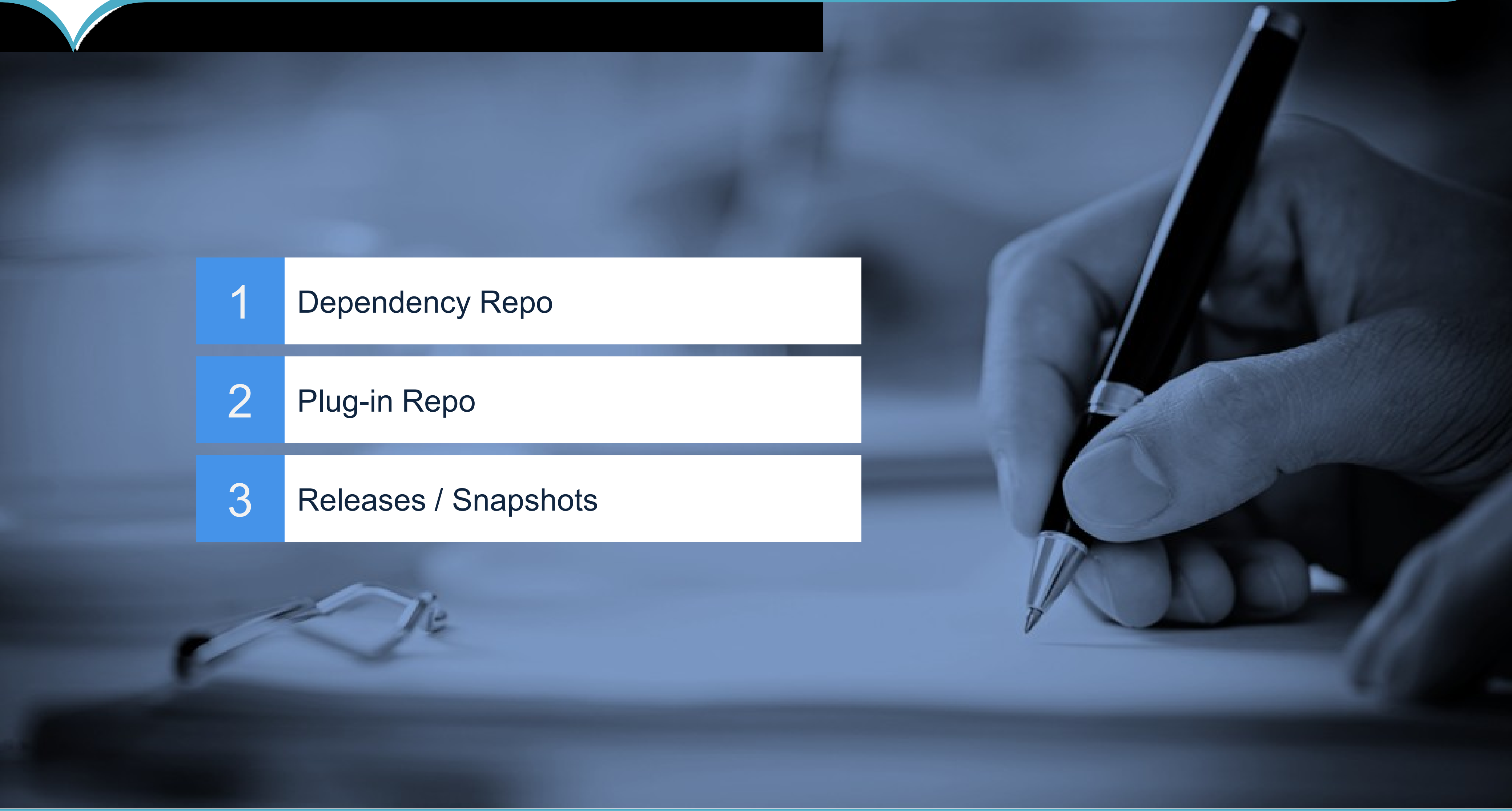
Transitive dependencies and how they are downloaded.

Scopes allow us to compile or test our code, but not include artifacts that shouldn’t be in the packaged code.

Repositories

Outline

[redacted]



Local Repo

Where Maven stores everything it downloads Installs in your home directory\.m2

- C:\Users\<yourusername>\.m2\repository

Stores artifacts using the information that you provided for artifactId, groupId, and version.

- C:\Users\<yourusername>\.m2\repository\commons-lang\commons-lang\2.1\commons-lang-2.1.jar

Avoids duplication by copying it in every project and storing it in your SCM.

Local Repo (contd.)

Where Maven stores everything it downloads. Installs in your home directory\.m2

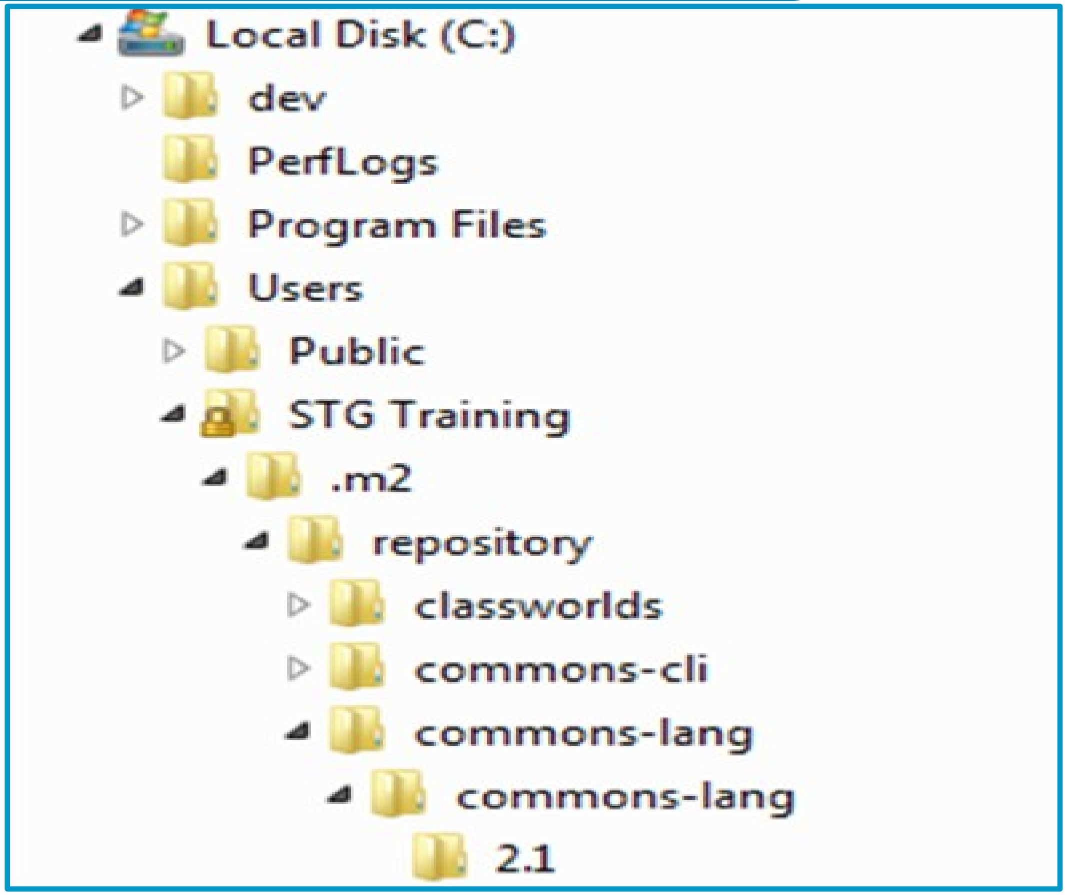
- **C:\Users\<yourusername>\.m2\repository**

Stores artifacts using the information that you provided for artifactId, groupId, and version.

- **C:\Users\<yourusername>\.m2\repository\commons-lang\commons-lang\2.1\commons-lang-2.1.jar**

Local Repo (contd.)

Avoids duplication by copying it in every project and storing it in your SCM.



```
Local Disk (C:)
  dev
  PerfLogs
  Program Files
  Users
    Public
    STG Training
      .m2
        repository
          classworlds
          commons-cli
          commons-lang
            commons-lang
              2.1
```

Repositories

Simply just a http accessible location that you download files from:

Super pom.xml

- Default with the Maven installation

Default location

- http://repo.maven.apache.org/maven2

Multiple repositories allowed

Corporate Repository

- Nexus (this is what the default repo is built on)
- Artifactory

Dependency Repository

Where we download all of our dependencies from:

- **Can contain just releases and/or snapshots**
- **Not uncommon to have them in separate repositories**

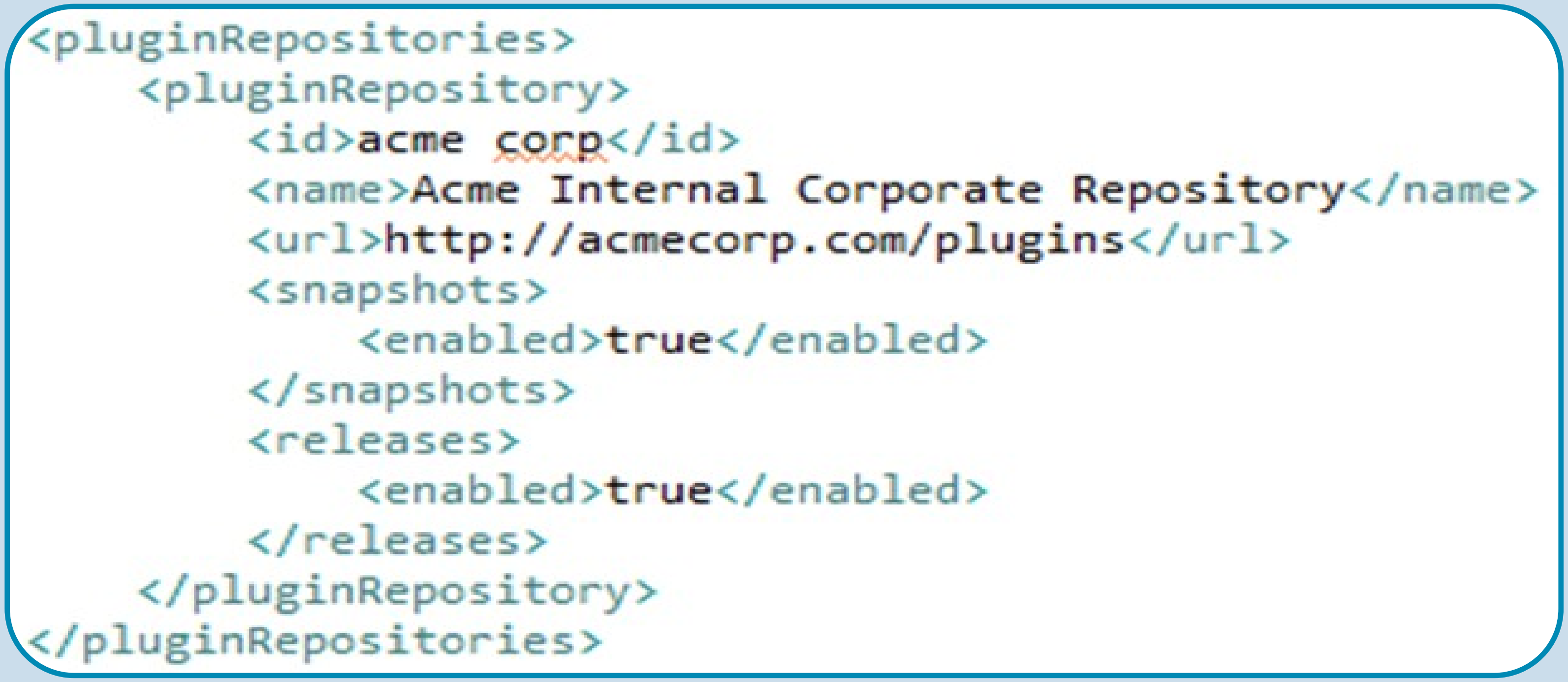
How do we specify our own repository

```
<repositories>
    <repository>
        <id>spring-snapshot</id>
        <name>Spring Maven SNAPSHOT Repository</name>
        <url>http://repo.springsource.org/libs-snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <releases>
            <enabled>false</enabled>
        </releases>
    </repository>
</repositories>
```

Plugin Repository

Identical to Dependency Repositories, just deals with Plugins.

Will only look for Plugins, by design usually a separate repository.



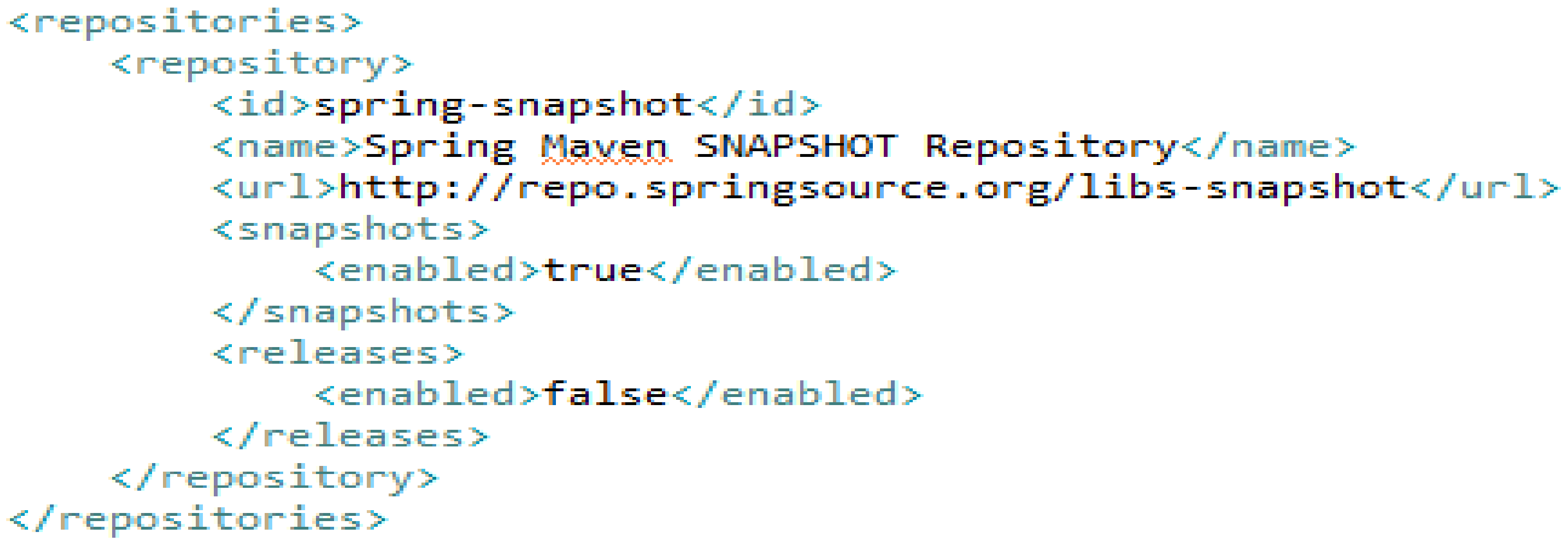
```
<pluginRepositories>
    <pluginRepository>
        <id>acme corp</id>
        <name>Acme Internal Corporate Repository</name>
        <url>http://acmecorp.com/plugins</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <releases>
            <enabled>true</enabled>
        </releases>
    </pluginRepository>
</pluginRepositories>
```

Releases / Snapshots

Snapshots and releases can come from the same repo.

Why would projects not upload everything to the central repo:

- **Snapshots**
- **Milestones**
- **Release Candidate**
- **Release policies**



```xml
<repositories>
    <repository>
        <id>spring-snapshot</id>
        <name>Spring Maven SNAPSHOT Repository</name>
        <url>http://repo.springsource.org/libs-snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <releases>
            <enabled>false</enabled>
        </releases>
    </repository>
</repositories>
```

Summary

Dependency repositories and Plug-in repositories can be separate or the same repository.

Projects will often not upload their SNAPSHOT code up to the central repo even though their release project is hosted there.

Plug-ins are usually in the same repo as dependencies.

Companies should use a corporate repository internally to help lighten the load on the central repo.

Plug-ins

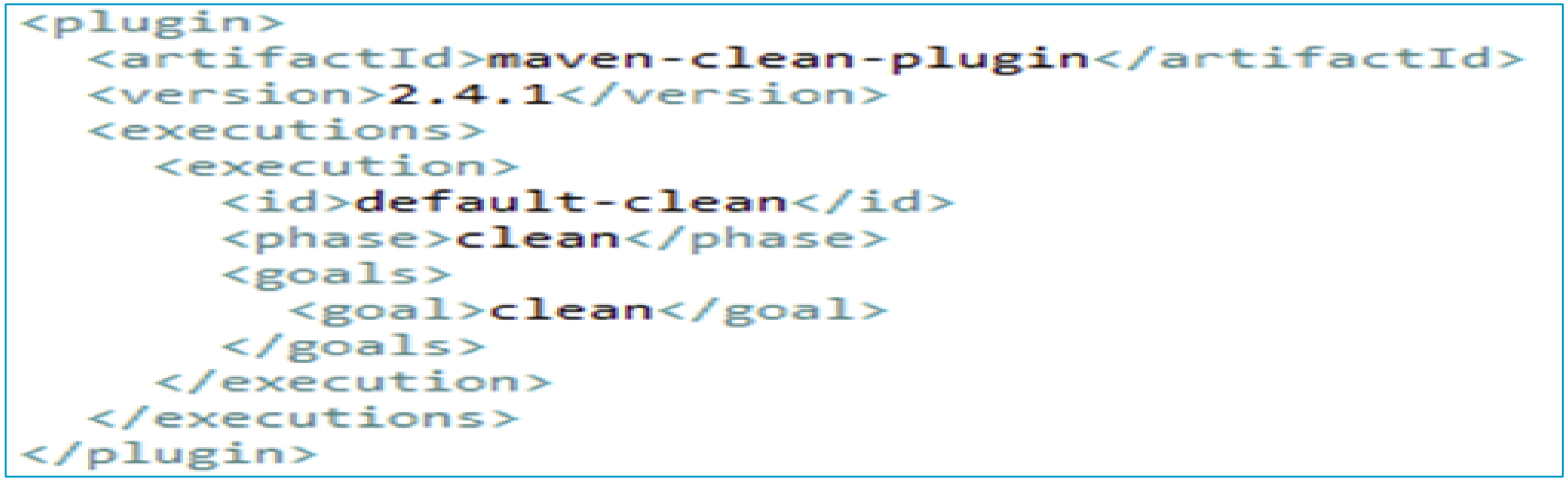
Outline

Goals

The default goals are plugins configured in the maven install.

- **clean, compile, test, package, install, deploy**

Super pom has these goals defined in it, which are added to your effective pom:



```
<plugin>
  <artifactId>maven-clean-plugin</artifactId>
  <version>2.4.1</version>
  <executions>
    <execution>
      <id>default-clean</id>
      <phase>clean</phase>
      <goals>
        <goal>clean</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Goals are tied to a phase.

Phases

Validate	• Validate the project is correct and all necessary information is available.
Compile	• Compile the source code of the project.
Test	• Test the compiled source code.
Package	• Packages the code in its defined package, such as a JAR.
Integration-test	• Deploy and run integration tests.
Verify	• Run checks against package to verify integrity.
Install	• Install the package in our local repo.
Deploy	• Copy final package to a remote repository.

Compiler Plugin

Used to compile code and test code.

http://maven.apache.org/plugins/maven-compiler-plugin/index.html

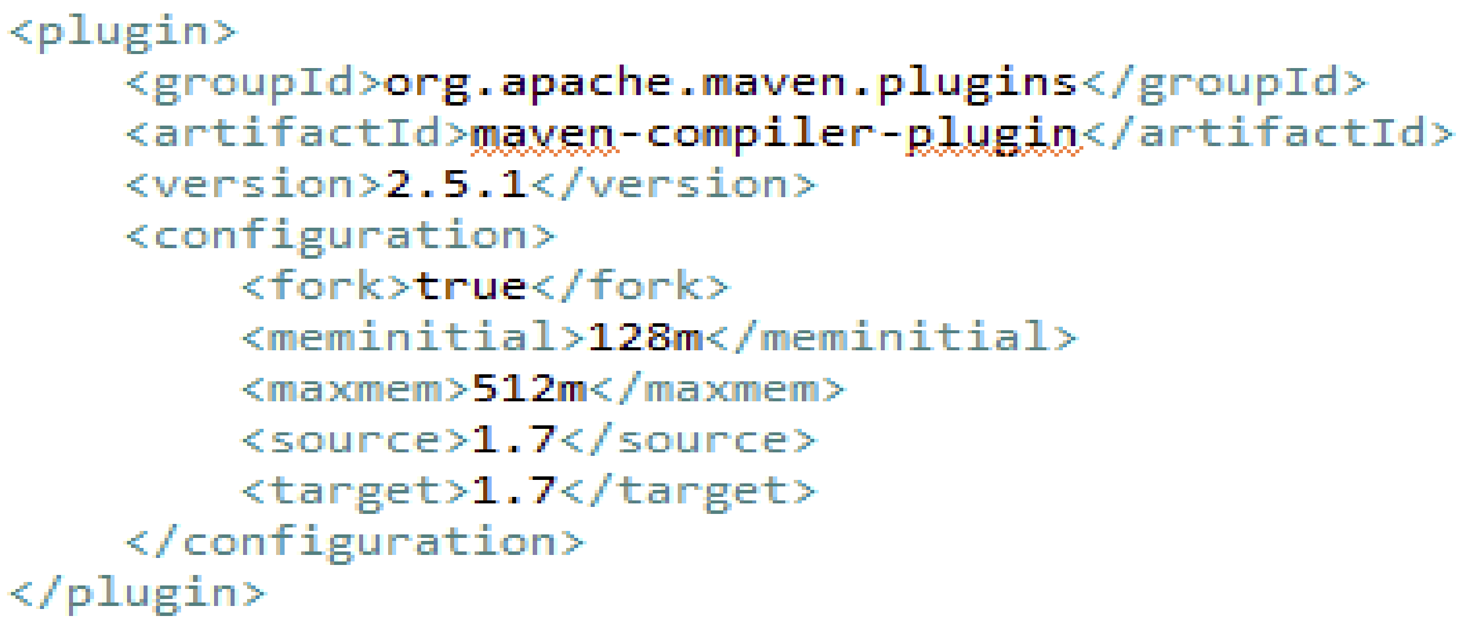
Invokes Javac, but with the classpath set from the dependencies.

Defaults to Java 1.5 regardless of what JDK is installed.

Compiler Plugin (contd.)

Configuration section allows customization.

- **Includes/Excludes**
- **Fork**
- **Memory**
- **Source/target**



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.5.1</version>
    <configuration>
        <fork>true</fork>
        <meminitial>128m</meminitial>
        <maxmem>512m</maxmem>
        <source>1.7</source>
        <target>1.7</target>
    </configuration>
</plugin>
```

Jar Plugin

Used to package code into a jar.

http://maven.apache.org/plugins/maven-jar-plugin/index.html

Tied to the package phase.

Configuration section allows customization.

- Includes/Excludes
- Manifest

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                <useDefaultManifestFile>true</useDefaultManifestFile>
            </configuration>
        </plugin>
    </plugins>
</build>
```

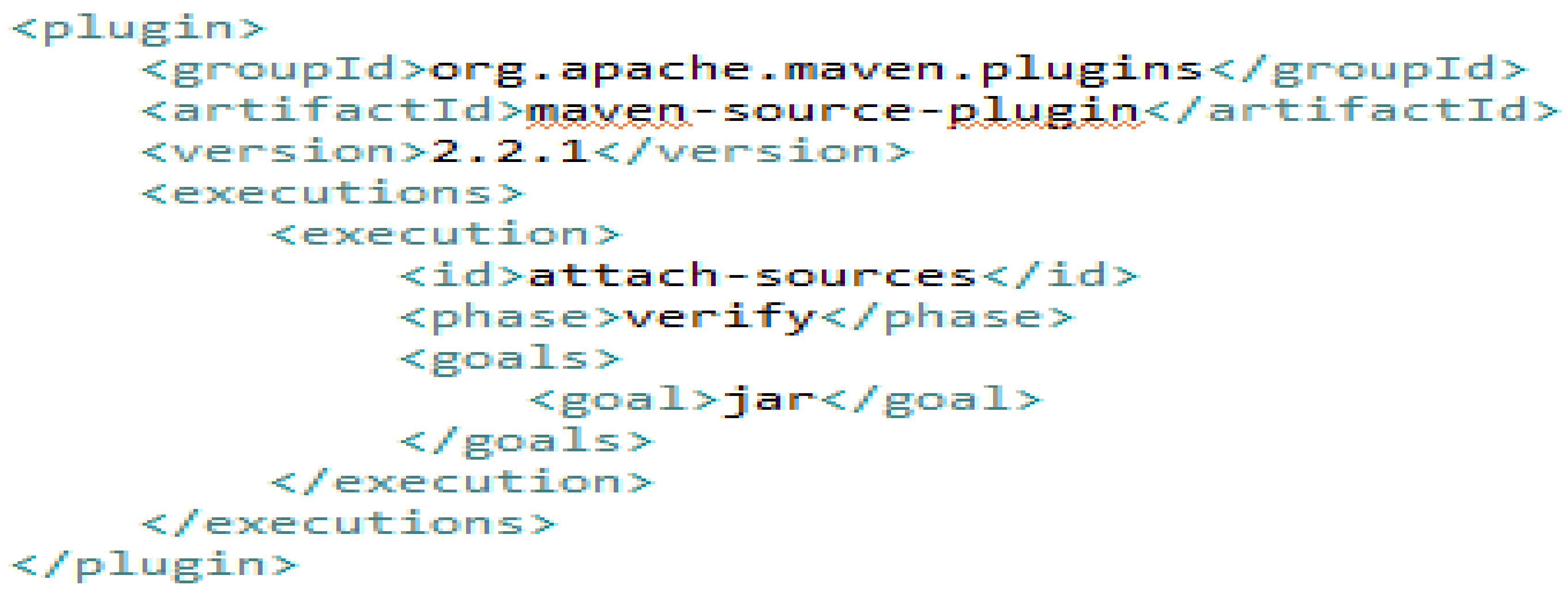
Source Plugin

Used to attach source code to a jar.

http://maven.apache.org/plugins/maven-source-plugin/index.html

Tied to the package phase.

- **Often overridden to a later phase.**



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
        <execution>
            <id>attach-sources</id>
            <phase>verify</phase>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Javadoc Plugin

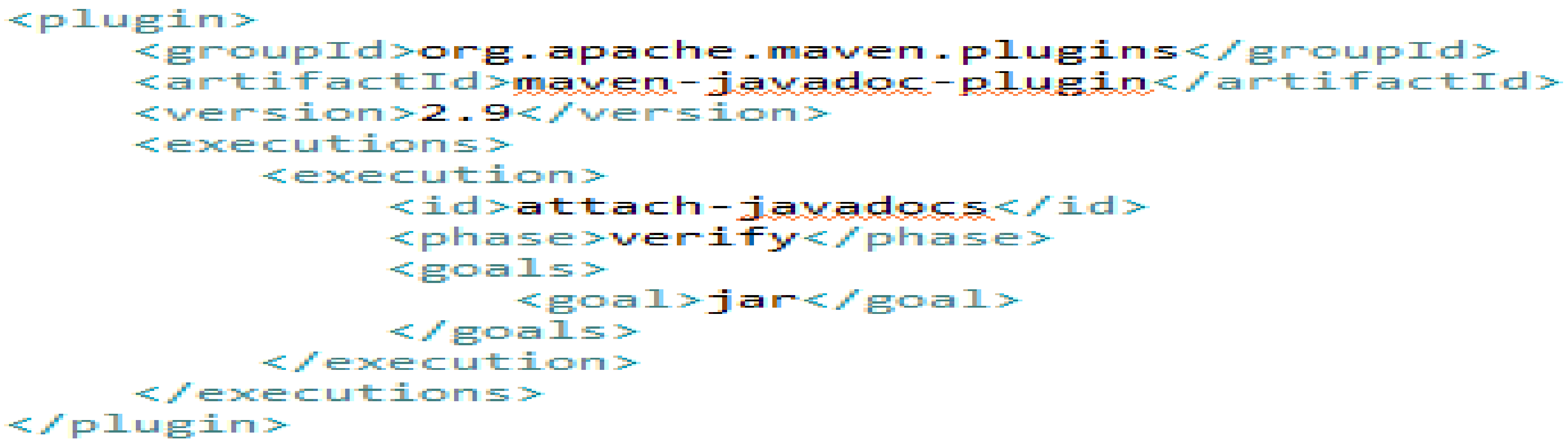
Used to attach Javadocs to a jar.

http://maven.apache.org/plugins/maven-javadoc-plugin/index.html

Tied to the package phase.

- Often overridden to a later phase.

Usually just use the defaults, but many customization options for Javadoc format.



```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>2.9</version>
    <executions>
        <execution>
            <id>attach-javadocs</id>
            <phase>verify</phase>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Summary

Goals are really just configured plugins in your application.

Plugins are tied to one of the defined phases, but can usually be overridden.

The compile plugin is already defined for you, but is often changed to use a specific version of Java.

The jar plugin is one of the default plugins and can be configured to produce artifacts to specific needs.

Source and Javadocs can easily be generated to be installed in your corporate repository for use by other developers.

Eclipse / Spring STS Integration

Outline

Installation

Eclipse / Spring STS doesn’t use the registry.

Java and Maven installed the same regardless of using an IDE.

Some IDEs do include a bundled version of Maven.

http://www.springsource.org/downloads/sts-ggts

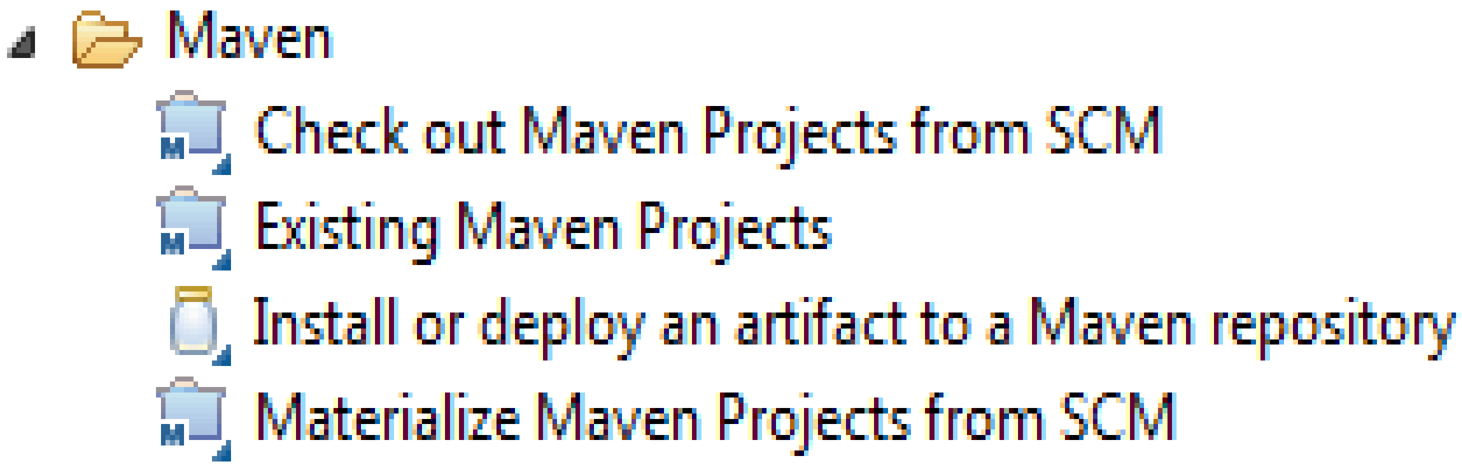
Importing Maven Projects

Modern IDEs have Maven integration built into them.

Maven integration will allow us to execute default maven goals within our IDE.

IDE configuration and Classpath will be set from Maven.

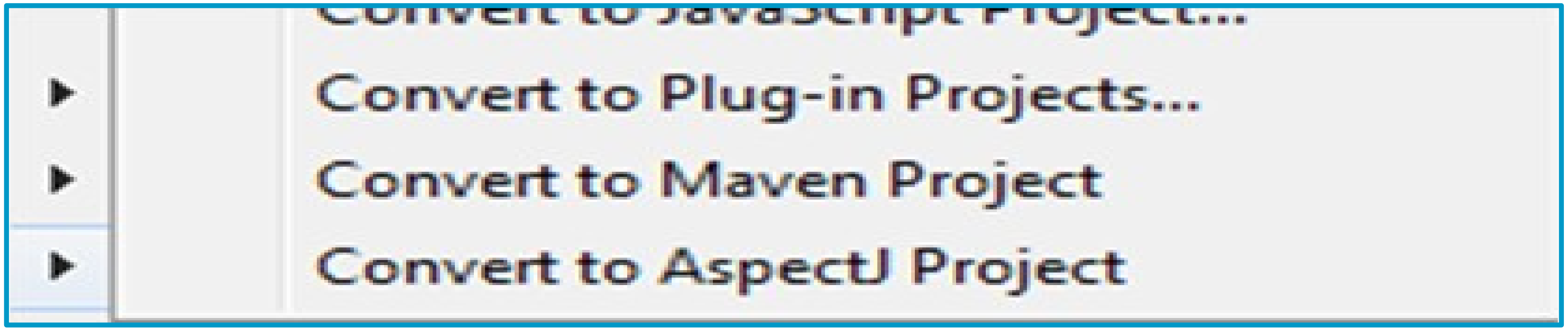
Right Click in the Package Explorer > Import > Maven > Existing Maven Projects.


Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM

Converting Existing Projects

If you have a pom.xml file, you can convert the project to a Maven project.

Right click on the project containing the pom.xml file > Configure > Convert to Maven Project



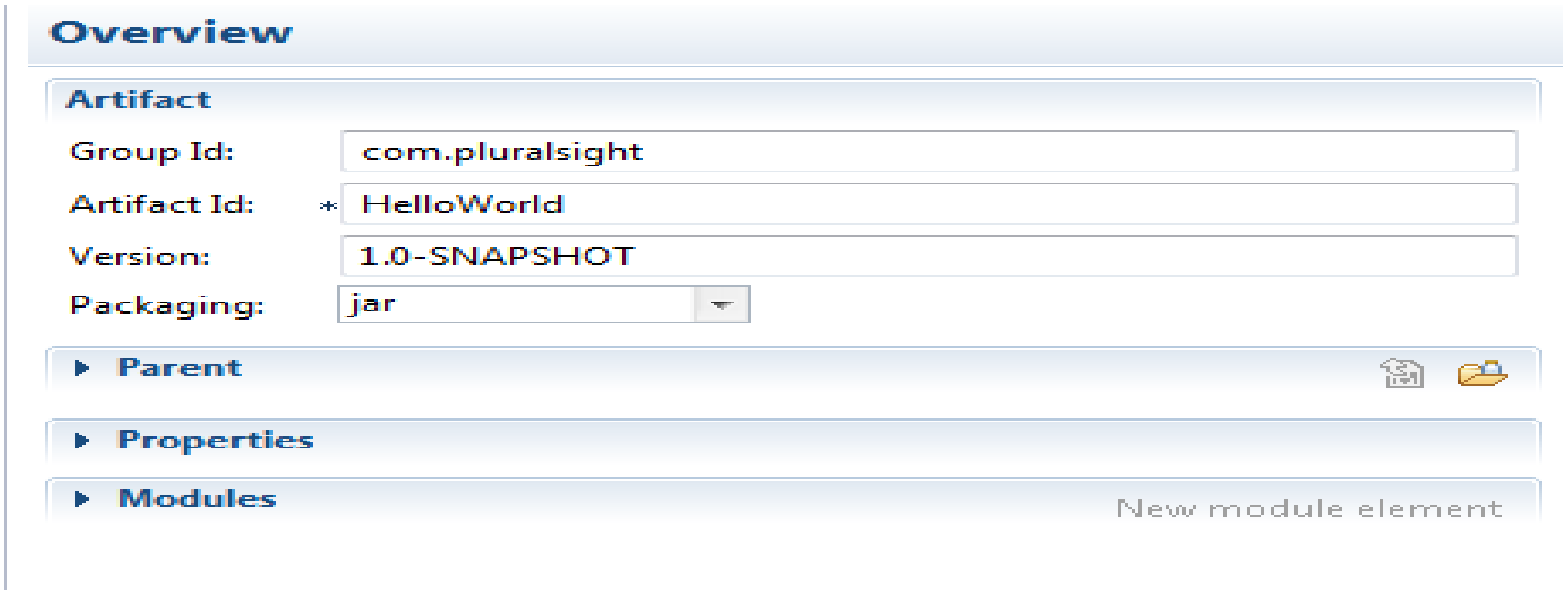
Once converted the project will set the classpath and automatically build the project.

Pom Viewer

Default view when you open the pom file.

Pom overview shows the high level elements of your project.

Changes made here are directly changing the source.



Overview

Artifact

Group Id: com.pluralsight

Artifact Id: * HelloWorld

Version: 1.0-SNAPSHOT

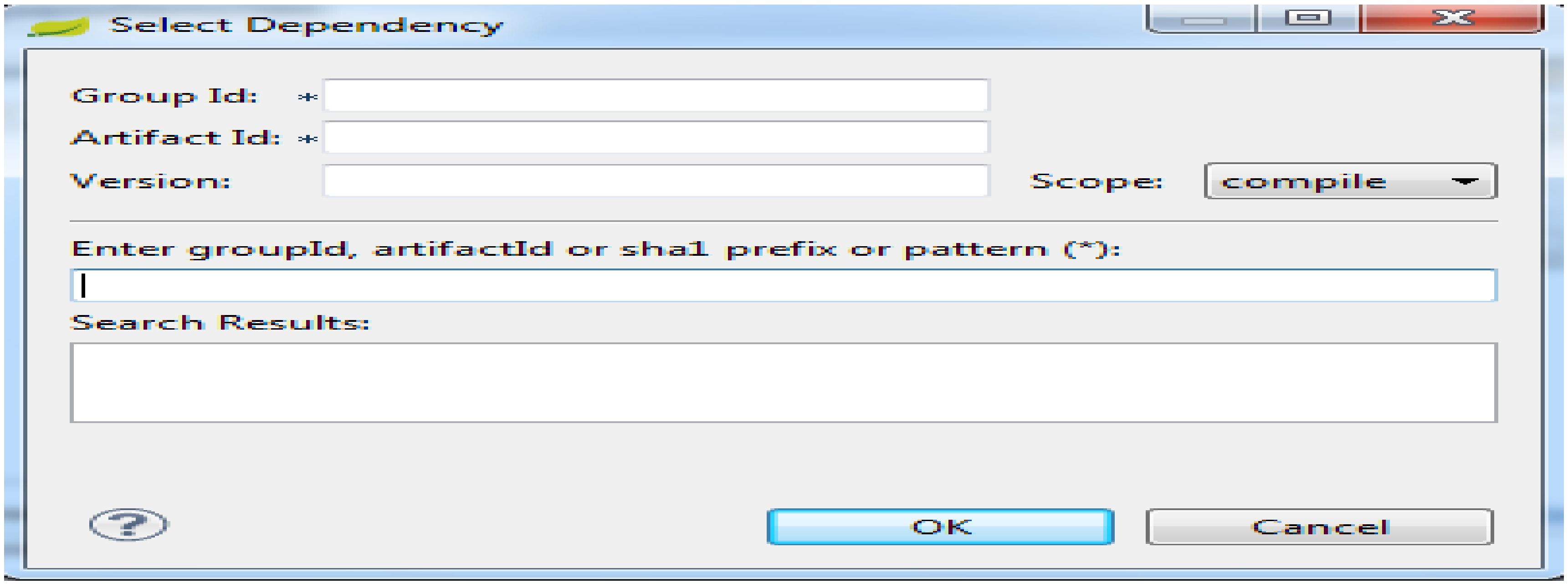
Packaging: jar

Dependencies

Shows which dependencies we have installed and allows us to manipulate dependencies too.

Dependency Management (advanced topic) is also displayed.

The add screen has searching capability.

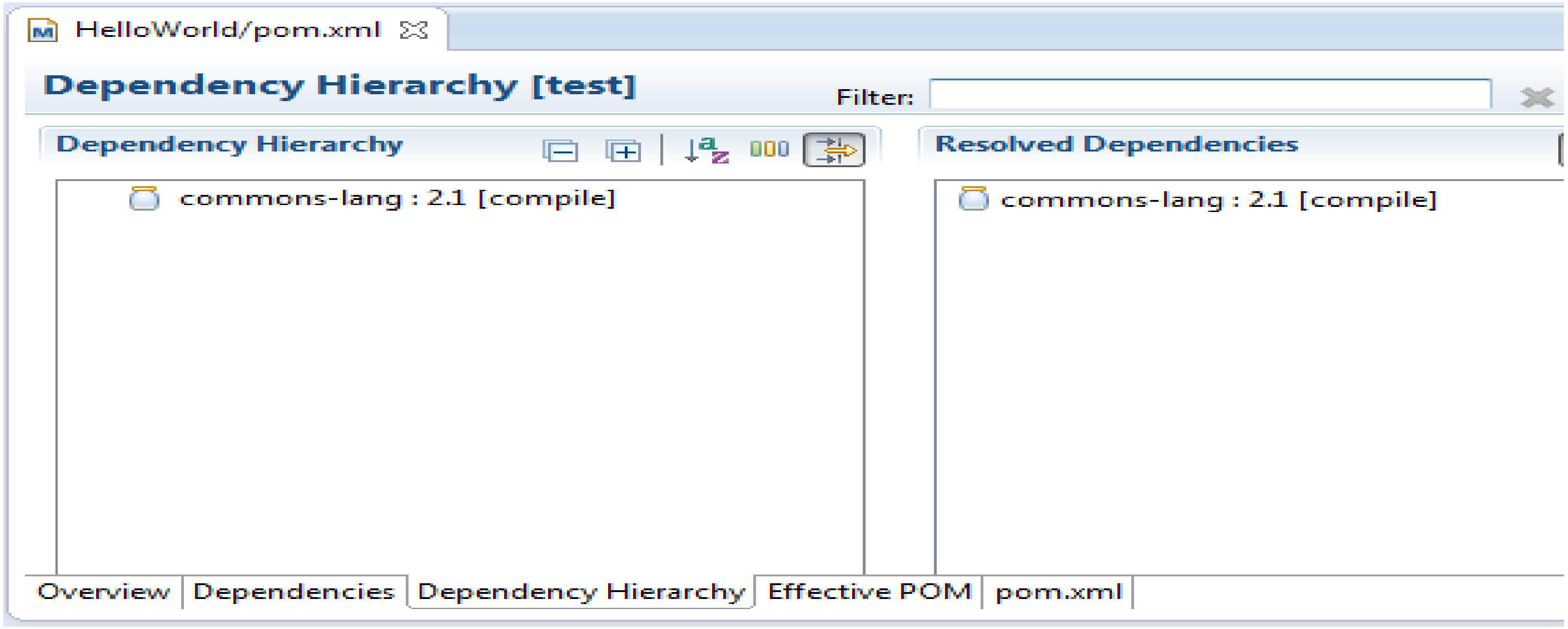


Group Id: *
Artifact Id: *
Version:
Scope: compile
Enter groupId, artifactId or sha1 prefix or pattern (*):
Search Results:

Dependency Hierarchy

Displays the complete dependency tree, including transitive dependencies as well overridden dependencies.

Scope of the resource is also displayed.



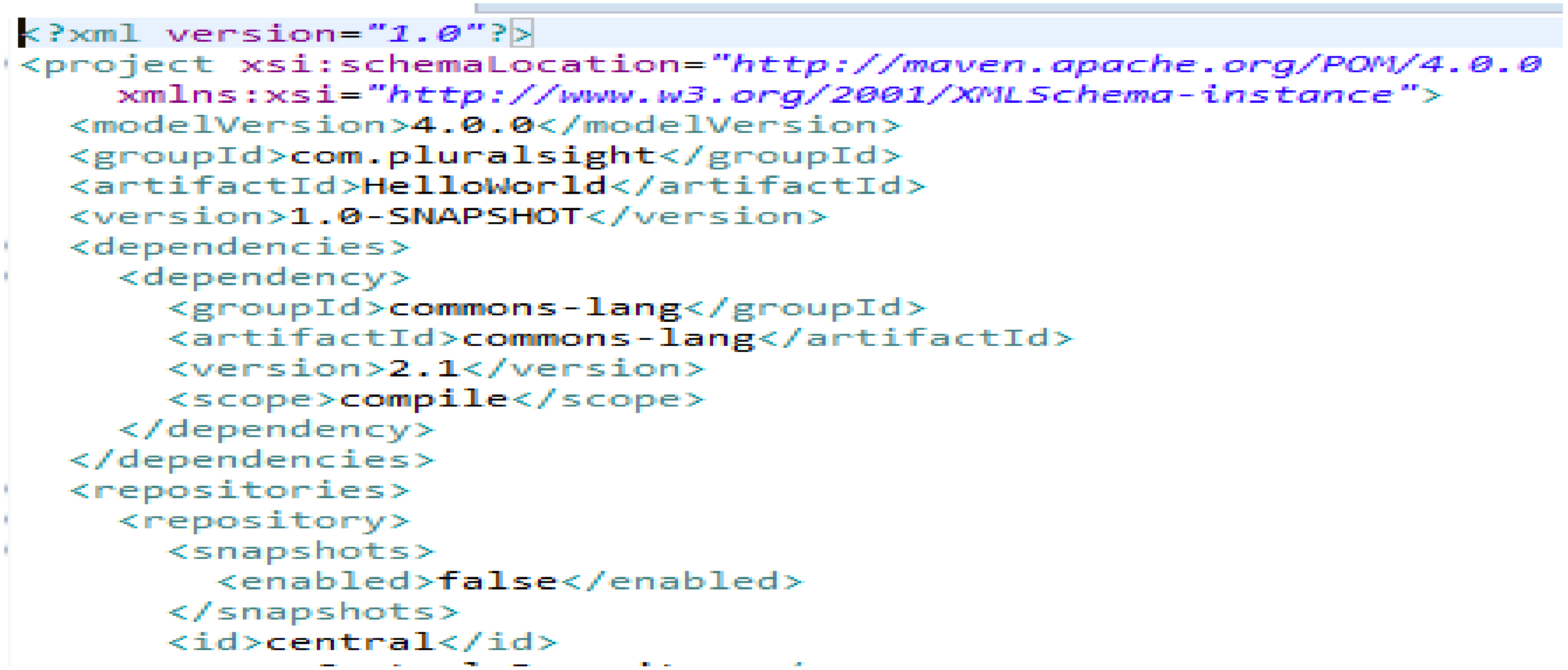
```
commons-lang : 2.1 [compile]
```

```
commons-lang : 2.1 [compile]
```

Effective Pom

The complete pom with everything inherited from the project pom, if we have a parent pom, and the default super pom.

More of a debugging tool to see what the pom is doing.



```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.pluralsight</groupId>
  <artifactId>HelloWorld</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.1</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  <repositories>
    <repository>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <id>central</id>
```

Summary

Eclipse / STS installation is really unzipping.

Existing projects can be imported easily and converting projects can be easily as well.

Adding dependencies inside the IDE can be easier using the searching tools.

Solving dependency resolution errors is more convenient in the IDE.

Configuring your IDE is also more convenient with Maven.

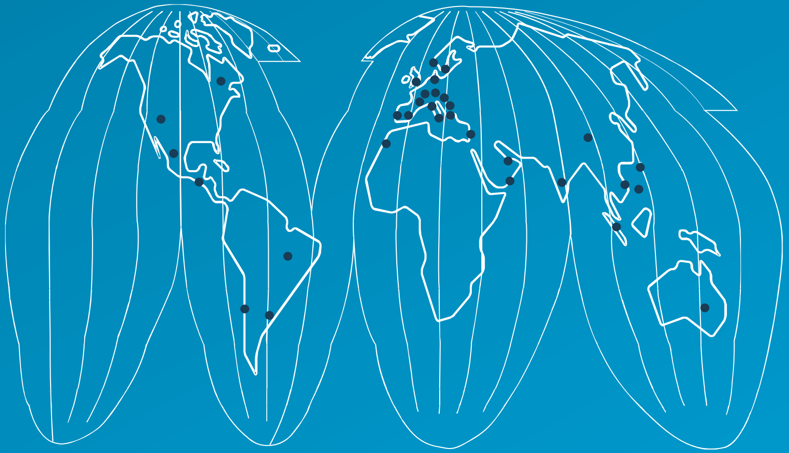
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

People matter, results count.

About Capgemini

With more than 145,000 people in 40 countries, Capgemini is one of the world's foremost providers of consulting, technology and outsourcing services. The Group reported 2014 global revenues of EUR 10.5 billion.
Together with its clients, Capgemini creates and delivers business and technology solutions that fit their needs and drive the results they want. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Rightshore® is a trademark belonging to Capgemini



www.capgemini.com