MAVEN – AN INTRODUCTION

NOTES

- This is a training **NOT** a presentation
- Please ask questions
- Prerequisites
  - Introduction to the Java Stack
  - Basic Java and XML skillz

OUTLINE

- Introduce Maven
- Basic Maven Pom File and Project Structure
- Dependencies

MAVEN BACKGROUND

- Is a Java build tool
  - “project management and comprehension tool”
- An Apache Project
  - Mostly sponsored by Sonatype

MAVEN BACKGROUND

- History
  - Maven 1 (2003)
    - Very Ugly
    - Used in Stack 1
  - Maven 2 (2005)
    - Complete rewrite
    - Not backwards Compatible
    - Used in Stack 2.0,2.1,2.2,3.0
  - Maven 3 (2010)
    - Same as Maven 2 but more stable
    - Used in Stack 2.3, 3.1

MAVEN FEATURES

- Dependency System
- Multi-module builds
- Consistent project structure
- Consistent build model
- Plugin oriented
- Project generated sites

THE MAVEN MINDSET

- All build systems are essentially the same:
  - Compile Source code
  - Copy Resource
  - Compile and Run Tests
  - Package Project
  - Deploy Project
  - Cleanup
- Describe the project and configure the build
  - You don’t script a build
  - Maven has no concept of a condition
  - **Plugins are configured**

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

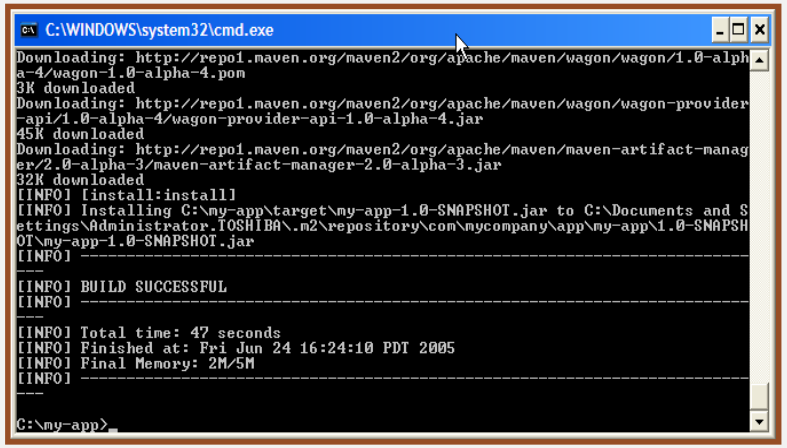
COMPARISON WITH ANT

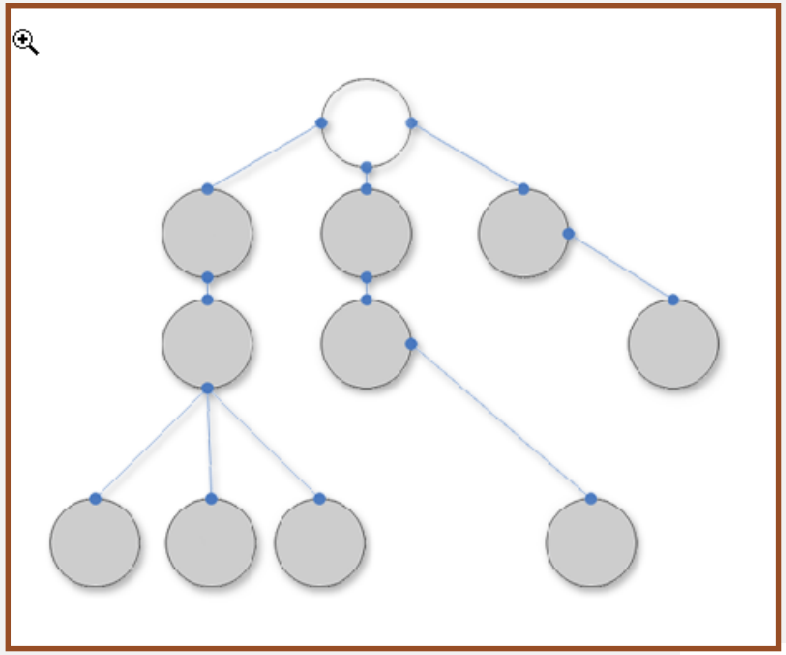
ANT	MAVEN
Target build.xml	Goal pom.xml

1. One level above ANT
2. Higher level of reusability between builds
3. Faster turn around time to set up a powerful build
4. Project website generation
5. Less maintenance
6. Greater momentum
7. Repository management
8. Automatic downloads

MAIN FEATURES OF MAVEN

- Build-Tool
- Dependency Management Tool
- Documentation Tool





MAVEN LEARNING RESOURCES

- Maven Homepage
  - http://maven.apache.org
    - Reference Documentation for Maven
    - Reference Documentation for core Plugins
- Sonatype Resources
  - http://www.sonatype.com/resource-center.html
    - Free Books
    - Videos

PROJECT CREATION IN MAVEN

mvn **archetype:generate**

-DgroupId = com.mycompany.app

-DartifactId = my-app

-DarchetypeArtifactId = maven-archetype-quickstart

-DinteractiveMode = false

CONTENTS OF THE CREATED PROJECT

- POM
- source tree for your application's sources
- source tree for your test sources

MAVEN POM

- Stands for Project Object Model
- Describes a project
  - Name and Version
  - Artifact Type
  - Source Code Locations
  - Dependencies
  - Plugins
  - Profiles (Alternate build configurations)
- Uses XML by Default
  - Not the way Ant uses XML

PROJECT NAME (GAV)

- Maven uniquely identifies a project using:
  - groupID: Arbitrary project grouping identifier (no spaces or colons)
    - Usually loosely based on Java package
  - artfiactId: Arbitrary name of project (no spaces or colons)
  - version: Version of project
    - Format {Major}.{Minor}.{Maintanence}
    - Add '-SNAPSHOT ' to identify in development
- GAV Syntax: groupId:artifactId:version

PROJECT NAME (GAV)

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

PROJECT OBJECT MODEL (POM)

- Metadata: Location of Directories, Developers/Contributors, Dependencies, Repositories
- Dependencies (Transitive Dependencies), Inheritance, and Aggregation
- Key Elements
  - Project
  - Model Version
  - Group ID
  - Packaging
  - Artifact ID
  - Version
  - Name
  - URL
  - Description

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>
```

PROJECT INHERITANCE

- Pom files can inherit configuration
  - groupId, version
  - Project Config
  - Dependencies
  - Plugin configuration
  - Etc.

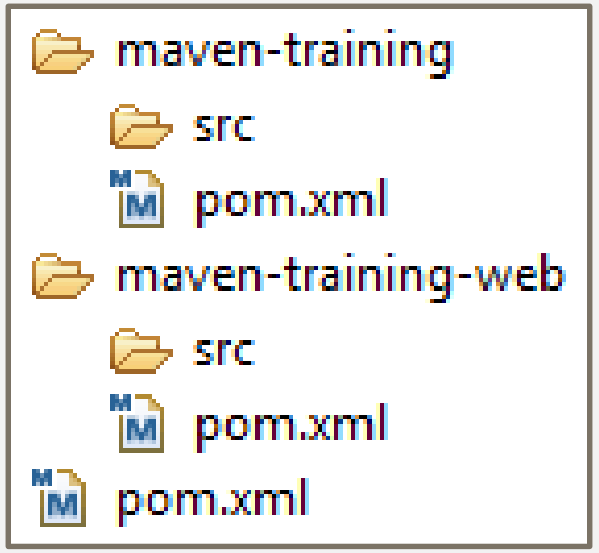
PROJECT INHERITANCE

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

MULTI MODULE PROJECTS

- Maven has 1st class multi-module support
- Each maven project creates 1 primary artifact
- A parent pom is used to group modules

```
<project>
    ...
    <packaging>pom</packaging>
    <modules>
        <module>maven-training</module>
        <module>maven-training-web</module>
    </modules>
</project>
```



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

MAVEN DIRECTORY STRUCTURE

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/main/scripts	Application/Library scripts
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site
LICENSE.txt	Project's license
NOTICE.txt	Notices and attributions required by libraries that the project depends on
README.txt	Project's readme

MAVEN BUILD LIFECYCLE

- A Maven build follow a lifecycle
- Default lifecycle
  - generate-sources/generate-resources
  - compile
  - test
  - package
  - integration-test (pre and post)
  - Install
  - deploy
- There is also a Clean lifecycle

EXAMPLE MAVEN GOALS

- To invoke a Maven build you set a lifecycle “goal”
- mvn install
  - Invokes generate* and compile, test, package, integration-test, install
- mvn clean
  - Invokes just clean
- mvn clean compile
  - Clean old builds and execute generate*, compile
- mvn compile install
  - Invokes generate*, compile, test, integration-test, package, install
- mvn test clean
  - Invokes generate*, compile, test then cleans

MAVEN AND DEPENDENCIES

- Maven revolutionized Java dependency management
  - No more checking libraries into version control
- Introduced the Maven Repository concept
  - Established Maven Central
- Created a module metadata file (POM)
- Introduced concept of transitive dependency
- Often include source and javadoc artifacts

ADDING A DEPENDENCY

- Dependencies consist of:
  - GAV
  - Scope: compile, test, provided (default=compile)
  - Type: jar, pom, war, ear, zip (default=jar)

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

MAVEN REPOSITORIES

- Dependencies are downloaded from repositories
  - Via http
- Downloaded dependencies are cached in a local repository
  - Usually found in ${user.home}/.m2/repository
- Repository follows a simple directory structure
  - {groupId}/{artifactId}/{version}/{artifactId}-{version}.jar
  - groupId '.' is replaced with '/'
- Maven Central is primary community repo
  - http://repo1.maven.org/maven2

PROXY REPOSITORIES

- Proxy Repositories are useful:
  - Organizationally cache artifacts
  - Allow organization some control over dependencies
  - Combines repositories
- Sapient uses – Nexus – May go to Artifactory
- All artifacts in Nexus go through approval process
  - License verified
  - Improve organizational reuse

DEFINING A REPOSITORY

- Repositories are defined in the pom
- Repositories can be inherited from parent
- Repositories are keyed by id
- Downloading snapshots can be controlled

DEFINING A REPOSITORY

```
<project>
    ...
    <repositories>
        <repository>
            <id>lds-main</id>
            <name>LDS Main Repo</name>
            <url>http://code.lds.org/nexus/content/groups/main-repo</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
</project>
```

TRANSITIVE DEPENDENCIES

- Transitive Dependency Definition:
  - A dependency that should be included when declaring project itself is a dependency
- ProjectA depends on ProjectB
- If ProjectC depends on ProjectA then ProjectB is automatically included
- Only compile and runtime scopes are transitive
- Transitive dependencies are controlled using:
  - Exclusions
  - Optional declarations

DEPENDENCY EXCLUSIONS

- Exclusions exclude transitive dependencies
- Dependency consumer solution

DEPENDENCY EXCLUSIONS

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>3.0.5.RELEASE</version>
            <exclusions>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
</project>
```

OPTIONAL DEPENDENCIES

- Don’t propagate dependency transitively
- Dependency producer solution
- Optional is under used

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>3.0.5.RELEASE</version>
            <optional>true</optional>
        </dependency>
    </dependencies>
</project>
```

DEPENDENCY MANAGEMENT

- What do you do when versions collide?
  - Allow Maven to manage it?
    - Complex and less predictable
  - Take control yourself
    - Manage the version manually
- In Java you cannot use both versions

DEPENDENCY MANAGEMENT

```
<project>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
                <version>3.0.5.RELEASE</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

USING DEPENDENCY MANAGEMENT

```
<project>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
                <version>3.0.5.RELEASE</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
        </dependency>  <!-- Look ma, no version! -->
    </dependencies>
</project>
```

USING DEPENDENCY MANAGEMENT

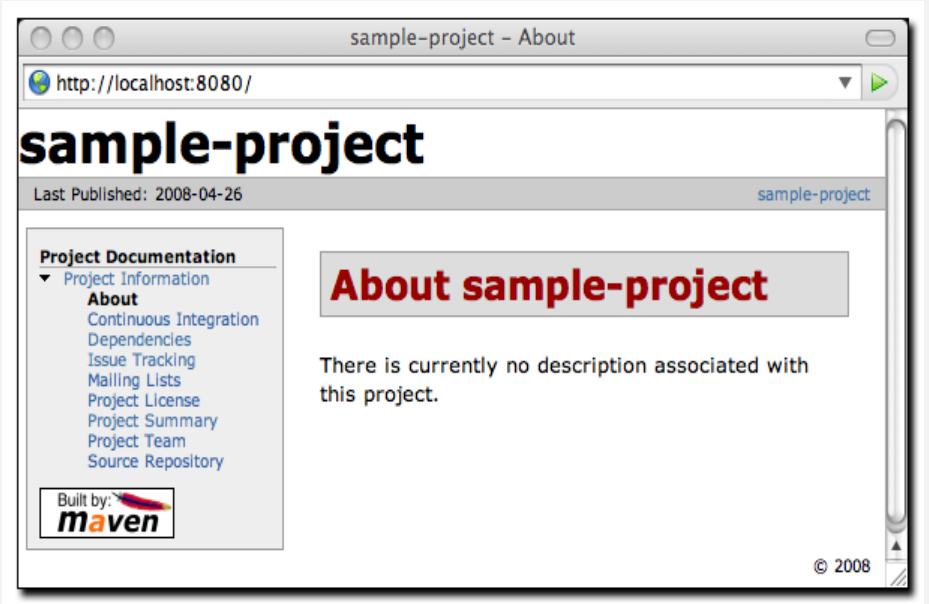
- Other uses for Dependency Management
  - Allowing parent pom to manage versions
  - Unify exclusions

DOCUMENTATION – BUILDING OWN SITE

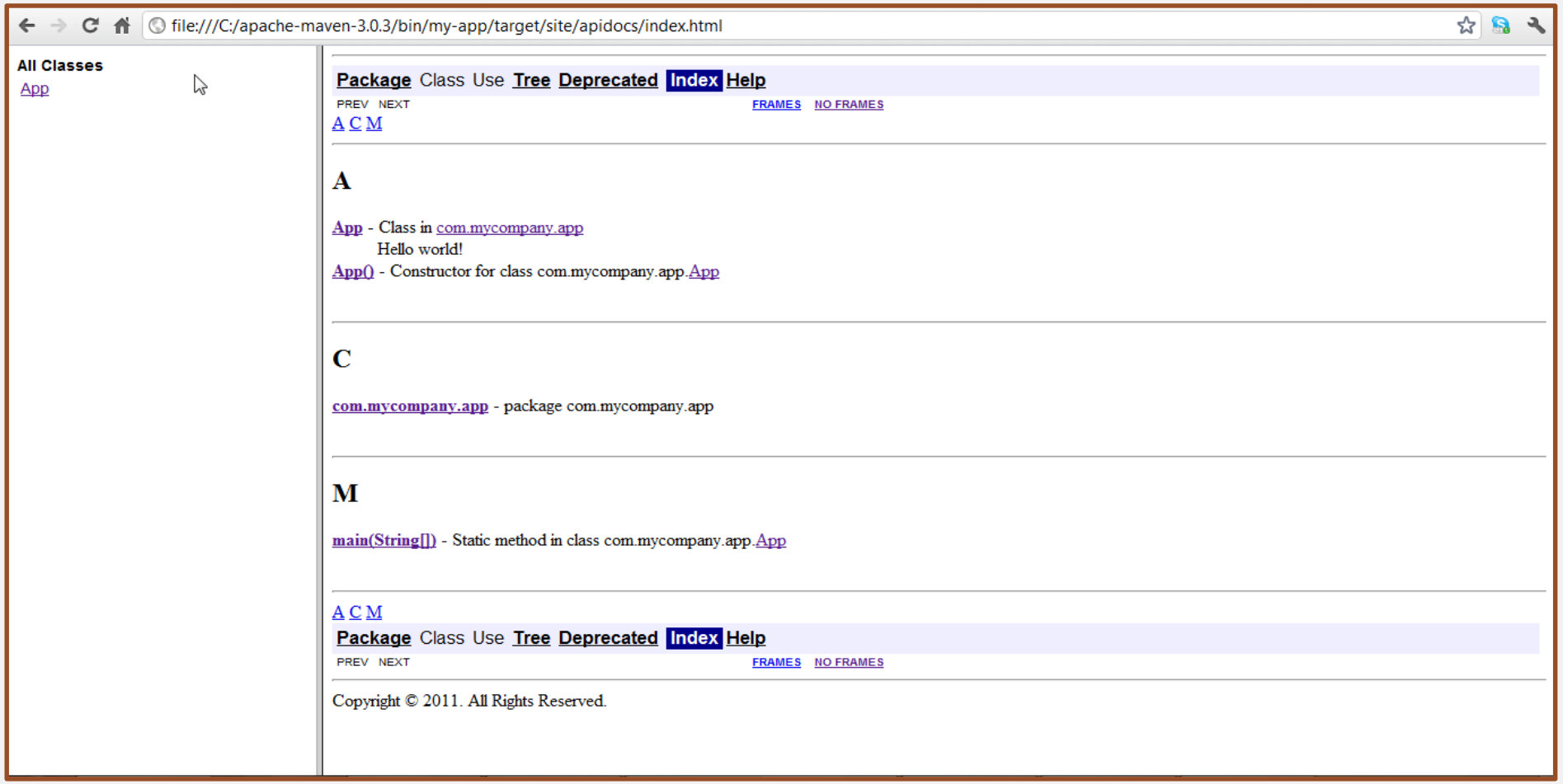
- mvn **site**
- pom.xml

```
<project> ...
<distributionManagement>
<site>
<id>website</id>
<url>scp://www.mycompany.com/www/docs/project/</url>
</site>
</distributionManagement> ...
</project>
```

- mvn **site-deploy**



REPORT GENERATION



BUILD LIFECYCLE

- Maven is based around the central concept of a build lifecycle.
  - The process for building and distributing a particular artifact (project) is clearly defined.
- For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.
- .

BUILD LIFECYCLE

- There are three built-in build lifecycles:
  - default,
  - clean and
  - site.
  - The default lifecycle handles your project deployment,
  - the clean lifecycle handles project cleaning,
  - while the site lifecycle handles the creation of your project's site documentation.

BUILD LIFECYCLE

- Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

BUILD LIFECYCLE

- 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 using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- package - take the compiled code and package it in its distributable format, such as a JAR.
- verify - run any checks on results of integration tests to ensure quality criteria are met
- install - install the package into the local repository, for use as a dependency in other projects locally
- deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

BUILD LIFECYCLES

- These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

BUILD LIFECYCLES

- A Phase is made up of plugin goals
- A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project.
- It may be bound to zero or more build phases.
- A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation.
- The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked.
- For example, consider the command below. The clean and package arguments are build phases, while the **dependency:copy-dependencies** is a goal (of a plugin).

BUILD LIFECYCLES

- What is being done here?

```
mvn clean dependency:copy-dependencies package
```

BUILD LIFECYCLES

- Moreover, if a goal is bound to one or more build phases, that goal will be called in all those phases.

- Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals

- (Note: In Maven 2.0.5 and above, multiple goals bound to a phase are executed in the same order as they are declared in the POM, however multiple instances of the same plugin are not supported. Multiple instances of the same plugin are grouped to execute together and ordered in Maven 2.0.11 and above).

BUILD LIFE CYCLES

- The phases named with hyphenated-words (pre-*, post-*, or process-*) are not usually directly called from the command line. These phases sequence the build, producing intermediate results that are not useful outside the build. In the case of invoking integration-test, the environment may be left in a hanging state.

- Code coverage tools such as Jacoco and execution container plugins such as Tomcat, Cargo, and Docker bind goals to the pre-integration-test phase to prepare the integration test container environment. These plugins also bind goals to the post-integration-test phase to collect coverage statistics or decommission the integration test container.

- Failsafe and code coverage plugins bind goals to integration-test and verify phases. The net result is test and coverage reports are available after the verify phase. If integration-test were to be called from the command line, no reports are generated. Worse is that the integration test container environment is left in a hanging state; the Tomcat webserver or Docker instance is left running, and Maven may not even terminate by itself.

EXAMPLES

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

HOW TO ADD THESE?

- Packaging
- Plugins

PLUGINS

- Plugins are artifacts that provide goals to Maven.
- Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code.

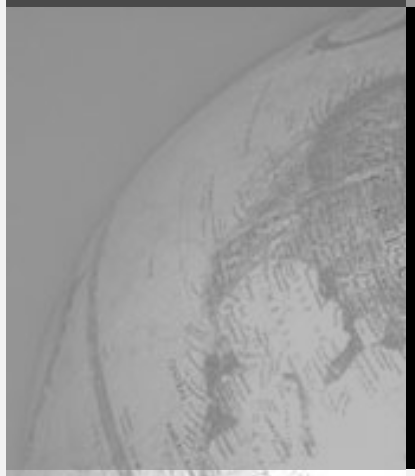
PLUGINS

- Plugins can contain information that indicates which lifecycle phase to bind a goal to.
- Note that adding the plugin on its own is not enough information - you must also specify the goals you want to run as part of your build.

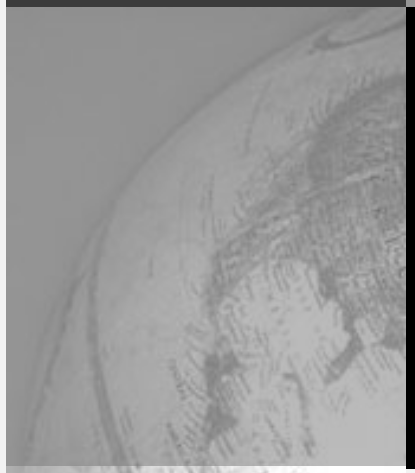
- The goals that are configured will be added to the goals already bound to the lifecycle from the packaging selected.
- If more than one goal is bound to a particular phase, the order used is that those from the packaging are executed first, followed by those configured in the POM.
- Note that you can use the <executions> element to gain more control over the order of particular goals.

SUMMARY

- Maven is a different kind of build tool
- It is easy to create multi-module builds
- Dependencies are awesome



Questions and Answers.



Thanks for listening!!!

Seshagiri Sriram
Email: SeshagiriSriram@gmail.com