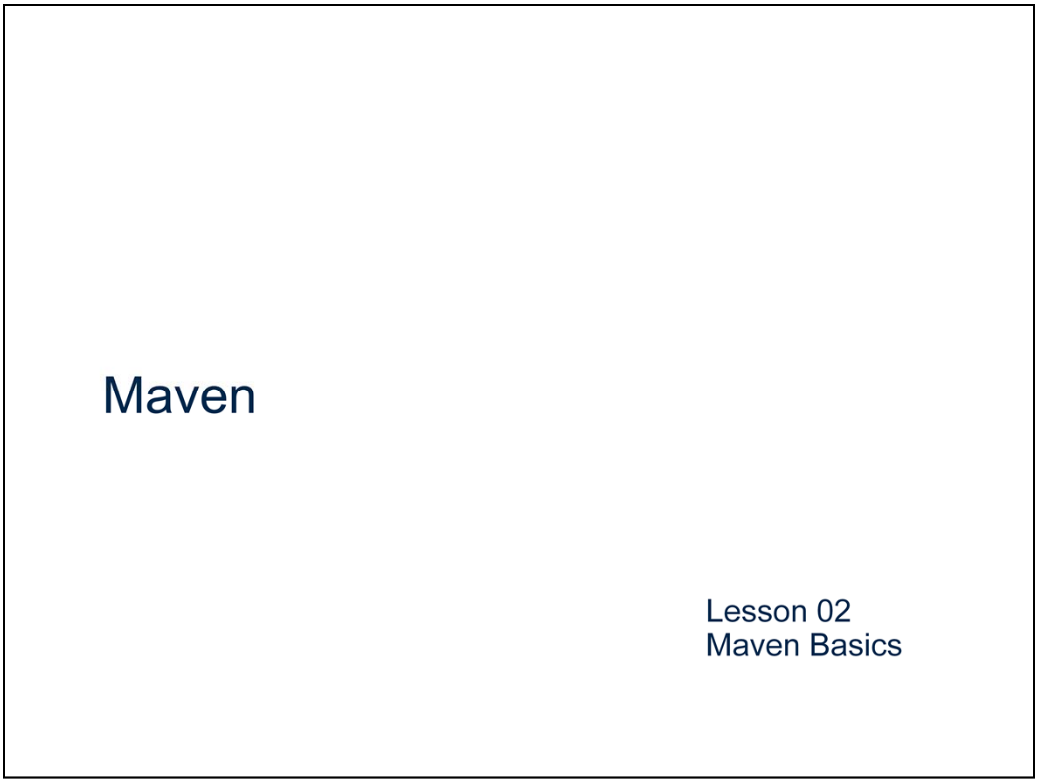
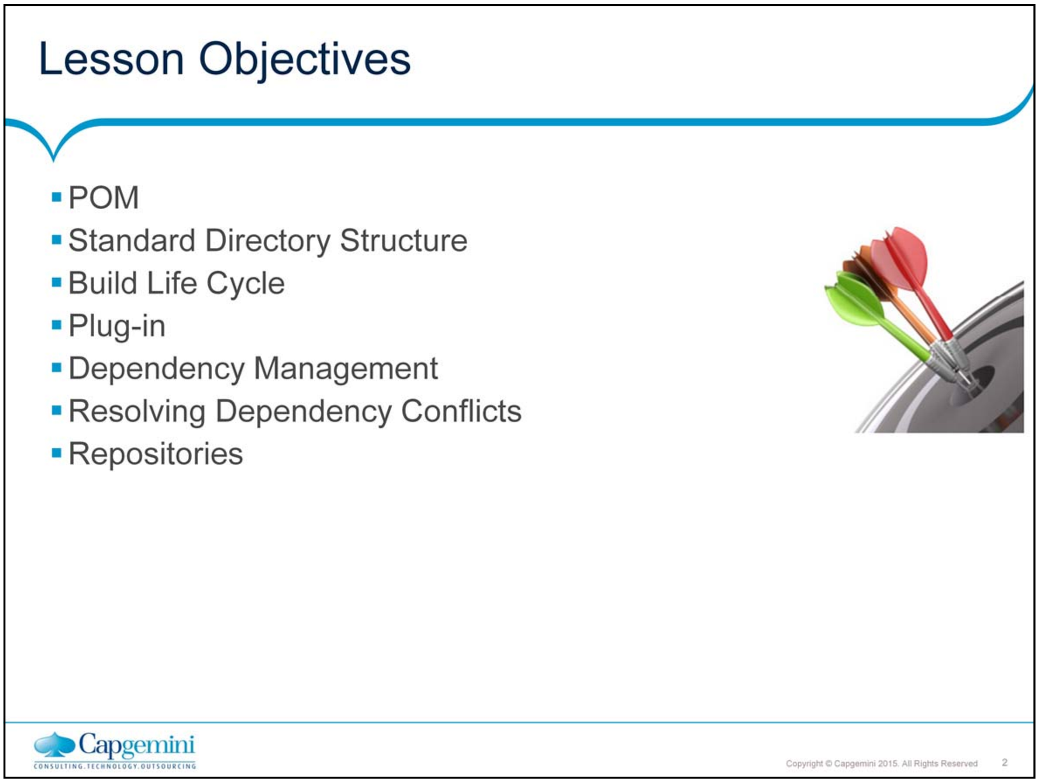


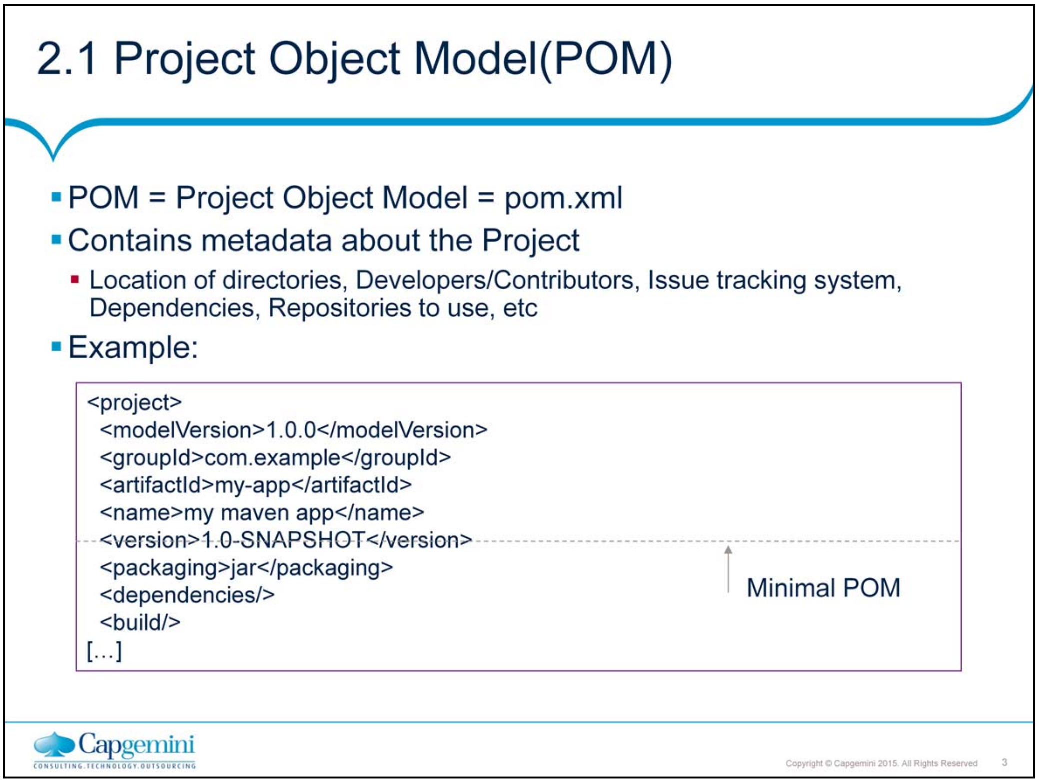
Maven

Lesson 02
Maven Basics



Lesson Objectives

- POM
- Standard Directory Structure
- Build Life Cycle
- Plug-in
- Dependency Management
- Resolving Dependency Conflicts
- Repositories

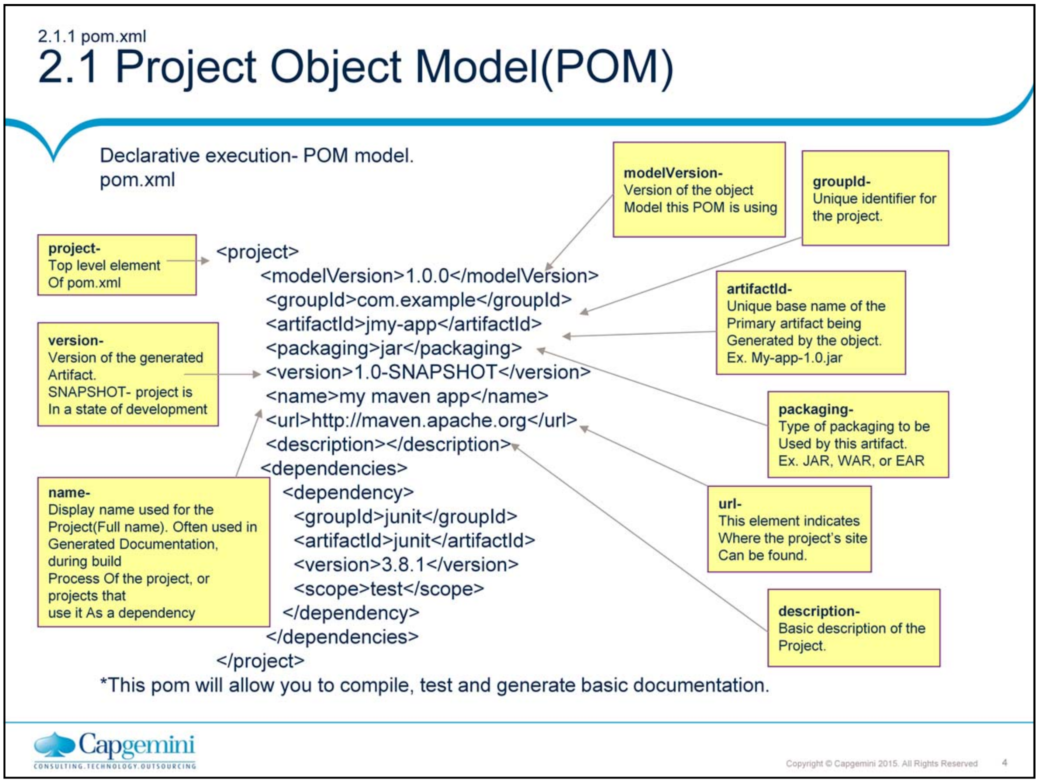


2.1 Project Object Model(POM)

- POM = Project Object Model = pom.xml
- Contains metadata about the Project
  - Location of directories, Developers/Contributors, Issue tracking system, Dependencies, Repositories to use, etc
- Example:

```
<project>
  <modelVersion>1.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <name>my maven app</name>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies/>
  <build/>
[…]
```

Minimal POM



2.1.1 pom.xml

2.1 Project Object Model(POM)

Declarative execution- POM model.
pom.xml

```
<project>
    <modelVersion>1.0.0</modelVersion>
     <groupId>com.example</groupId>
     <artifactId>jmy-app</artifactId>
     <packaging>jar</packaging>
     <version>1.0-SNAPSHOT</version>
     <name>my maven app</name>
     <url>http://maven.apache.org</url>
     <description></description>
    <dependencies>
       <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>3.8.1</version>
         <scope>test</scope>
       </dependency>
     </dependencies>
</project>
```

*This pom will allow you to compile, test and generate basic documentation.

project-
Top level element
Of pom.xml

version-
Version of the generated
Artifact.
SNAPSHOT- project is
In a state of development

name-
Display name used for the
Project(Full name). Often used in
Generated Documentation,
during build
Process Of the project, or
projects that
use it As a dependency

modelVersion-
Version of the object
Model this POM is using

groupId-
Unique identifier for
the project.

artifactId-
Unique base name of the
Primary artifact being
Generated by the object.
Ex. My-app-1.0.jar

packaging-
Type of packaging to be
Used by this artifact.
Ex. JAR, WAR, or EAR

url-
This element indicates
Where the project's site
Can be found.

description-
Basic description of the
Project.

Copyright © Capgemini 2015. All Rights Reserved 4

The **<project>** element is the root of the project descriptor.

<modelVersion> : Declares to which version of project descriptor this POM conforms.

<groupId> :A universally unique identifier for a project. It is normal to use a fully-qualified package name to distinguish it from other projects with a similar name (eg. org.apache.maven).

<artifactId > : The identifier for this artifact that is unique within the group given by the group ID. An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions, and WARs.

<packaging> : The type of artifact this project produces, for example jar war ear pom. Plugins can create their own packaging, and therefore their own packaging types, so this list does not contain all possible types.
Default value is: jar.

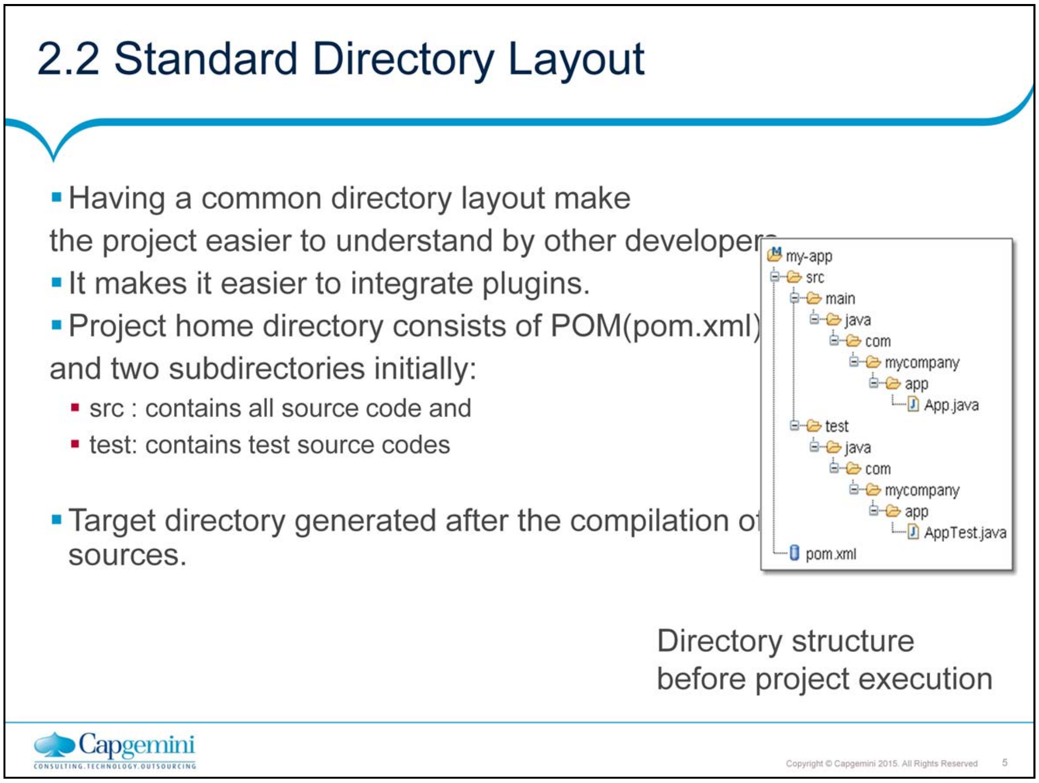
<Name> : The full name of the project.

<url> : The URL to the project's homepage.

<Version> : The current version of the artifact produced by this project.

<description> : A detailed description of the project, used by Maven whenever it needs to describe the project, such as on the web site.

<dependencies>: This element describes all of the dependencies associated with a project. These dependencies are used to construct a classpath for your project during the build process. They are automatically downloaded from the repositories defined in this project.



2.2 Standard Directory Layout

- Having a common directory layout make the project easier to understand by other developers.
- It makes it easier to integrate plugins.
- Project home directory consists of POM(pom.xml) and two subdirectories initially:
  - src : contains all source code and
  - test: contains test source codes
- Target directory generated after the compilation of sources.

```
my-app
├─src
│ ├─main
│ │ └─java
│ │   └─com
│ │     └─mycompany
│ │       └─app
│ │         └─App.java
│ └─test
│   └─java
│     └─com
│       └─mycompany
│         └─app
│           └─AppTest.java
└─pom.xml
```

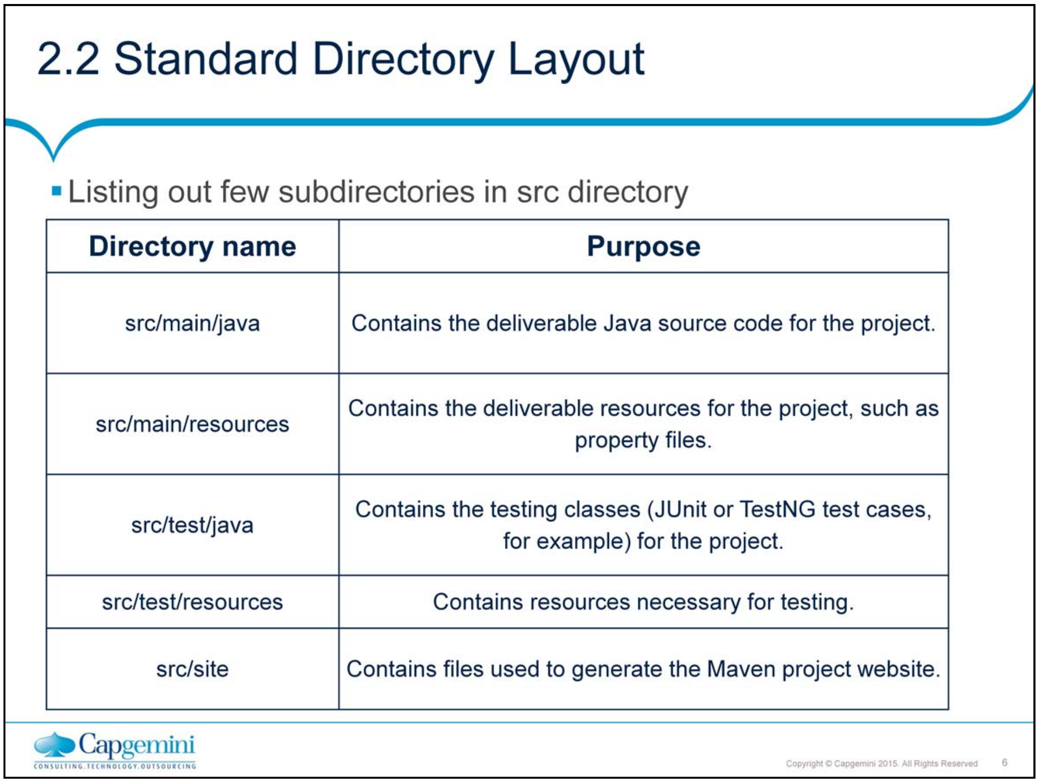
Directory structure
before project execution

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Advantages:

– A developer familiar with Maven will quickly get familiar with a new project

– No time wasted on re-inventing directory structures and conventions



2.2 Standard Directory Layout

- Listing out few subdirectories in src directory

Directory name	Purpose
src/main/java	Contains the deliverable Java source code for the project.
src/main/resources	Contains the deliverable resources for the project, such as property files.
src/test/java	Contains the testing classes (JUnit or TestNG test cases, for example) for the project.
src/test/resources	Contains resources necessary for testing.
src/site	Contains files used to generate the Maven project website.

The src directory has a number of subdirectories, each of which has a clearly defined purpose. Listing out few subdirectories in src directory:

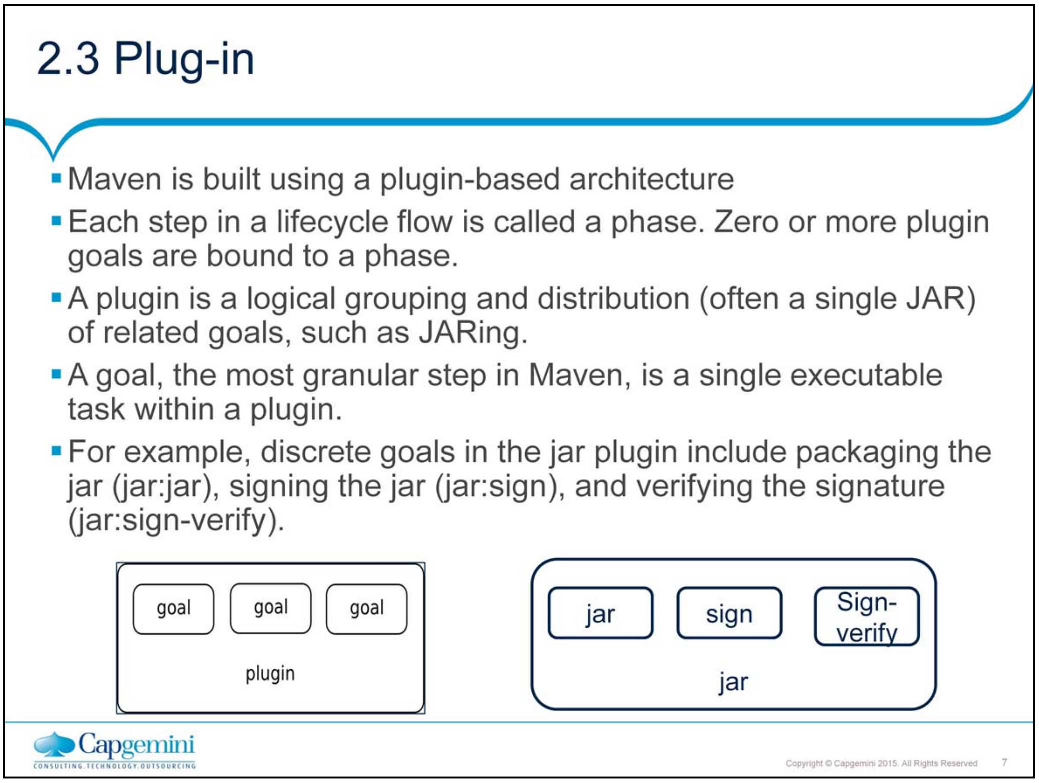
src/main/java - Contains the deliverable Java source code for the project.

src/main/resources - Contains the deliverable resources for the project, such as property files.

src/test/java - Contains the testing classes (JUnit or TestNG test cases, for example) for the project.

src/test/resources - Contains resources necessary for testing.

src/site - Contains files used to generate the Maven project website.

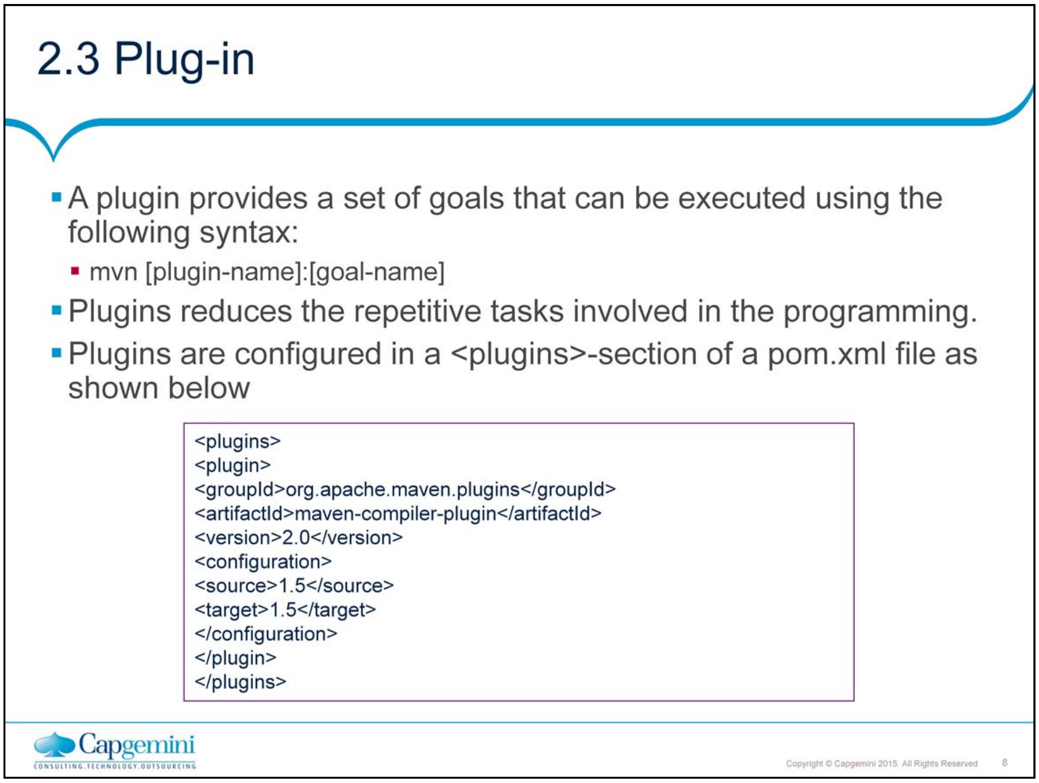


2.3 Plug-in

- Maven is built using a plugin-based architecture
- Each step in a lifecycle flow is called a phase. Zero or more plugin goals are bound to a phase.
- A plugin is a logical grouping and distribution (often a single JAR) of related goals, such as JARing.
- A goal, the most granular step in Maven, is a single executable task within a plugin.
- For example, discrete goals in the jar plugin include packaging the jar (jar:jar), signing the jar (jar:sign), and verifying the signature (jar:sign-verify).

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Plugin	Description
Core plugins	Plugins corresponding to default core phases (ie. clean, compile). They may have multiple goals as well.
clean	Clean up after the build.
compiler	Compiles Java sources.
deploy	Deploy the built artifact to the remote repository.
failsafe	Run the JUnit integration tests in an isolated classloader.
install	Install the built artifact into the local repository.
resources	Copy the resources to the output directory for including in the JAR.
site for Maven 2	Generate a site for the current project.
surefire	Run the JUnit unit tests in an isolated classloader.
verifier	Useful for integration tests - verifies the existence of certain conditions.



2.3 Plug-in

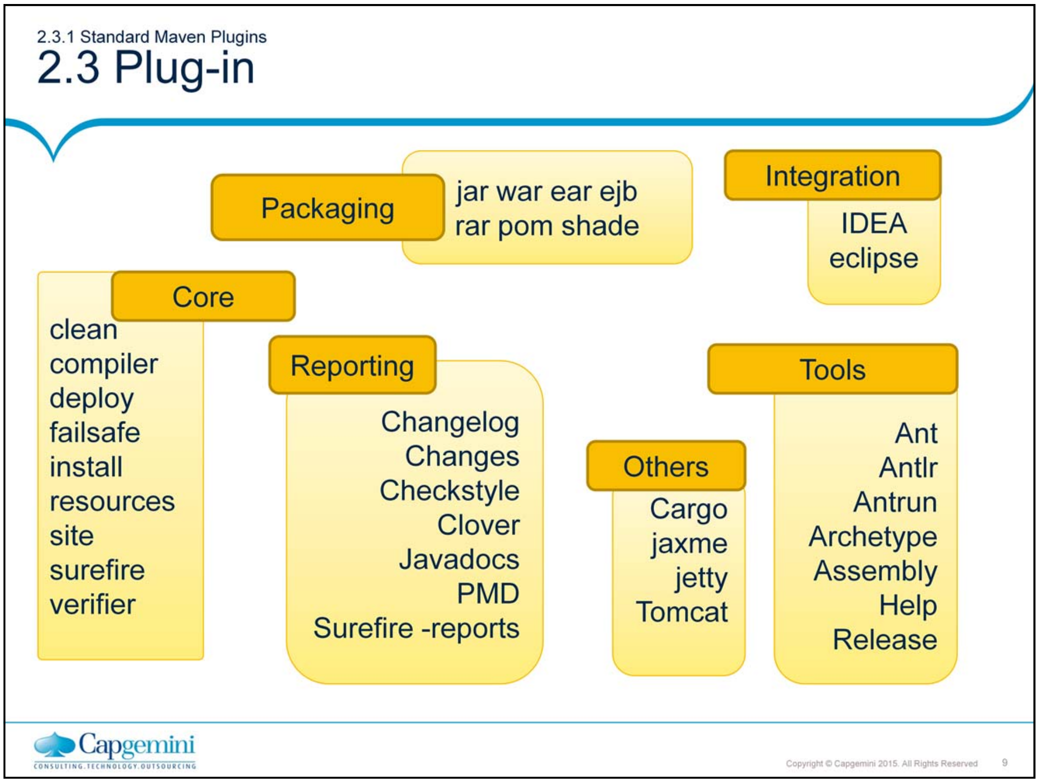
- A plugin provides a set of goals that can be executed using the following syntax:
  - mvn [plugin-name]:[goal-name]
- Plugins reduces the repetitive tasks involved in the programming.
- Plugins are configured in a <plugins>-section of a pom.xml file as shown below

```
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>2.0</version>
<configuration>
<source>1.5</source>
<target>1.5</target>
</configuration>
</plugin>
</plugins>
```

For example, configuring the Java compiler to allow JDK 5.0 sources in a project. This is as simple as adding this following to your POM:

```
<project>
...
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>2.0</version>
<configuration>
<source>1.5</source>
<target>1.5</target>
</configuration>
</plugin>
</plugins>
</build>
...
</project>
```

This plugin will be downloaded and installed automatically, if it is not present on your local system.



2.3.1 Standard Maven Plugins

2.3 Plug-in

Packaging
jar war ear ejb
rar pom shade

Integration
IDEA
eclipse

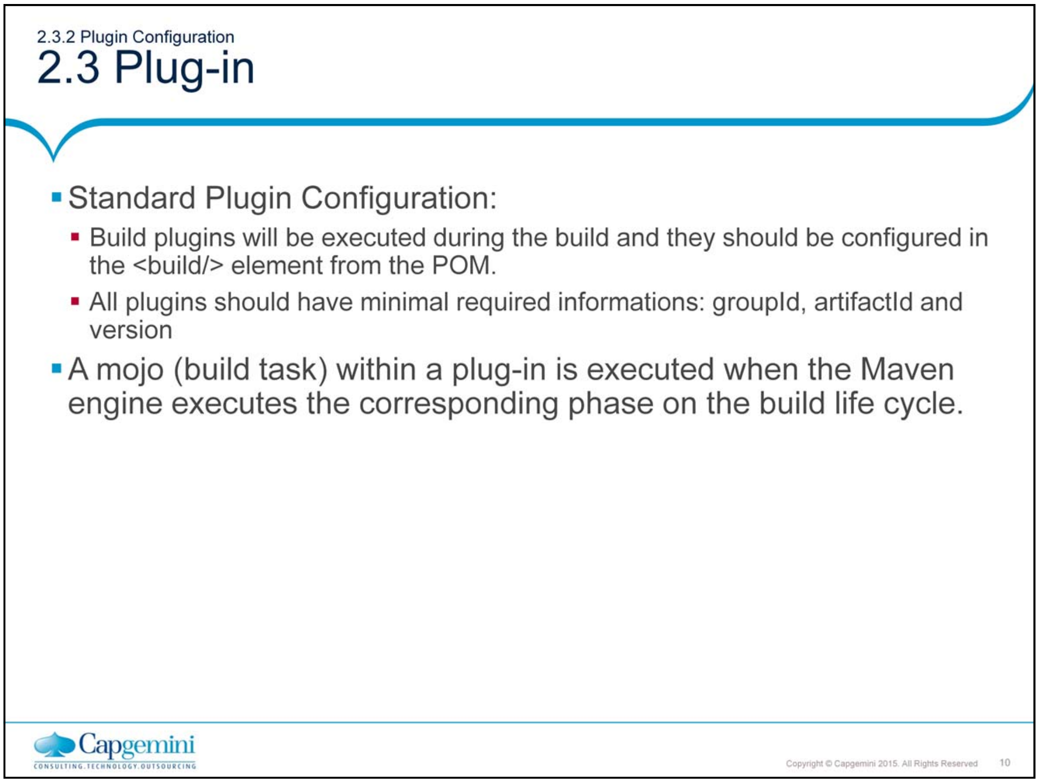
Core
clean
compiler
deploy
failsafe
install
resources
site
surefire
verifier

Reporting
Changelog
Changes
Checkstyle
Clover
Javadocs
PMD
Surefire -reports

Others
Cargo
jaxme
jetty
Tomcat

Tools
Ant
Antlr
Antrun
Archetype
Assembly
Help
Release

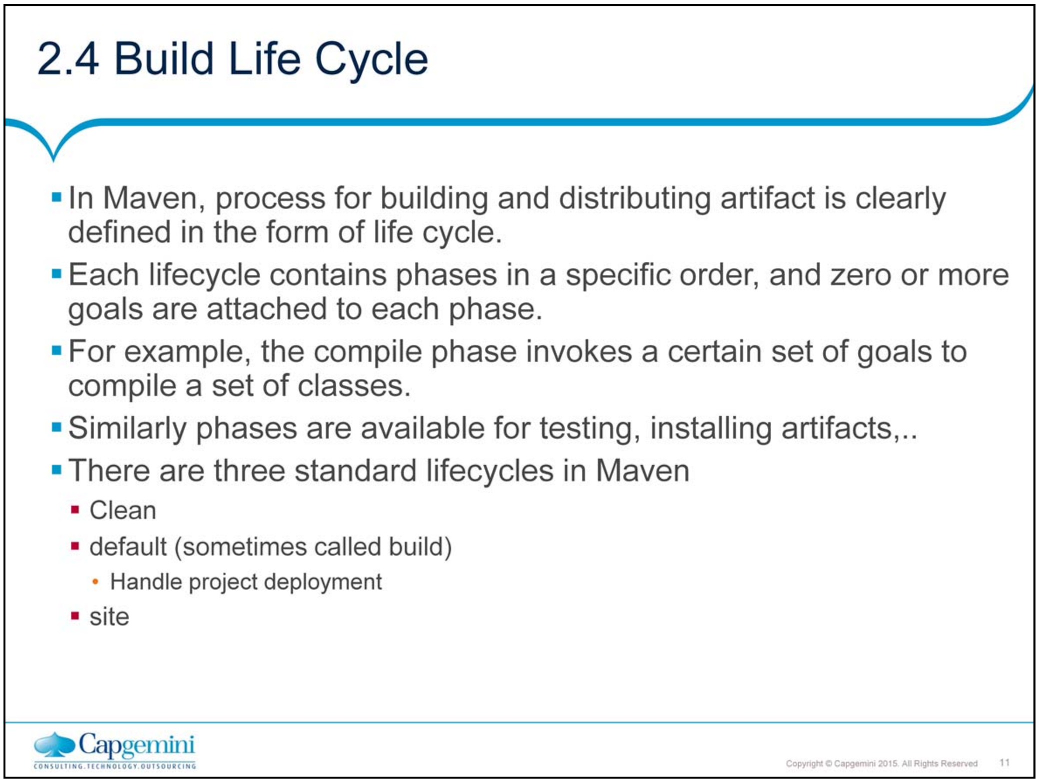
To see the most up-to-date list browse the Maven repository at http://repo1.maven.org/maven2/, specifically the org/apache/maven/plugins subfolder. *(Plugins are organized according to a directory structure that resembles the standard Java package naming convention)*



2.3.2 Plugin Configuration

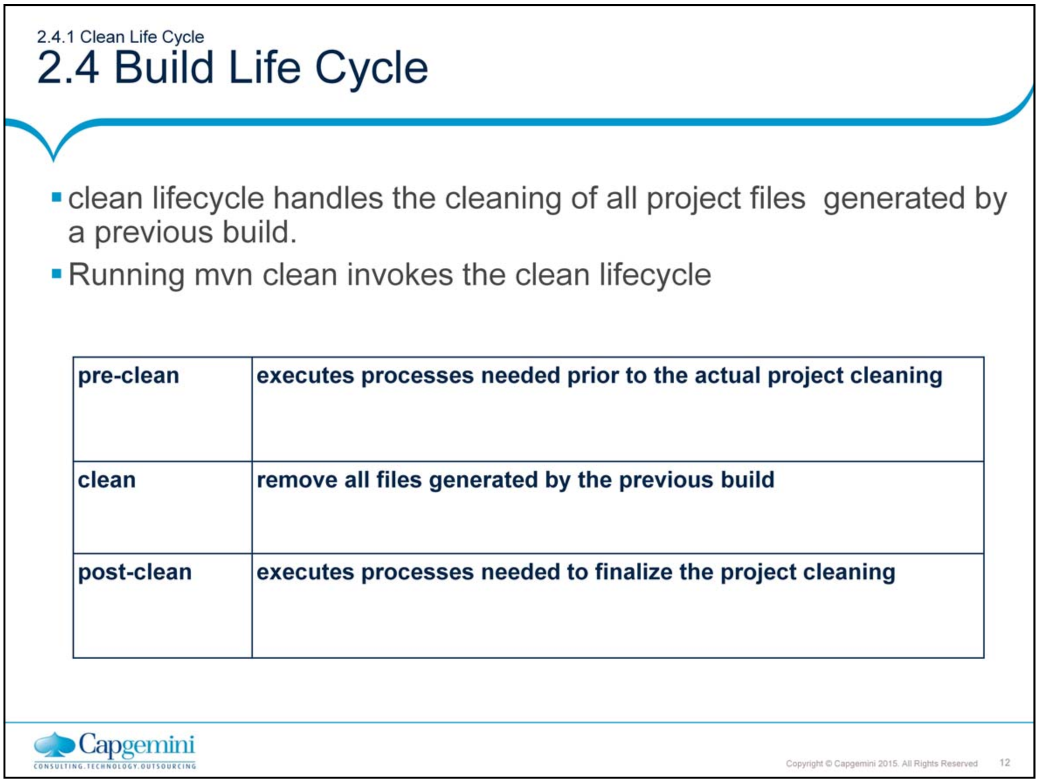
2.3 Plug-in

- Standard Plugin Configuration:
  - Build plugins will be executed during the build and they should be configured in the <build/> element from the POM.
  - All plugins should have minimal required informations: groupId, artifactId and version
- A mojo (build task) within a plug-in is executed when the Maven engine executes the corresponding phase on the build life cycle.



2.4 Build Life Cycle

- In Maven, process for building and distributing artifact is clearly defined in the form of life cycle.
- Each lifecycle contains phases in a specific order, and zero or more goals are attached to each phase.
- For example, the compile phase invokes a certain set of goals to compile a set of classes.
- Similarly phases are available for testing, installing artifacts,..
- There are three standard lifecycles in Maven
  - Clean
  - default (sometimes called build)
    - Handle project deployment
  - site



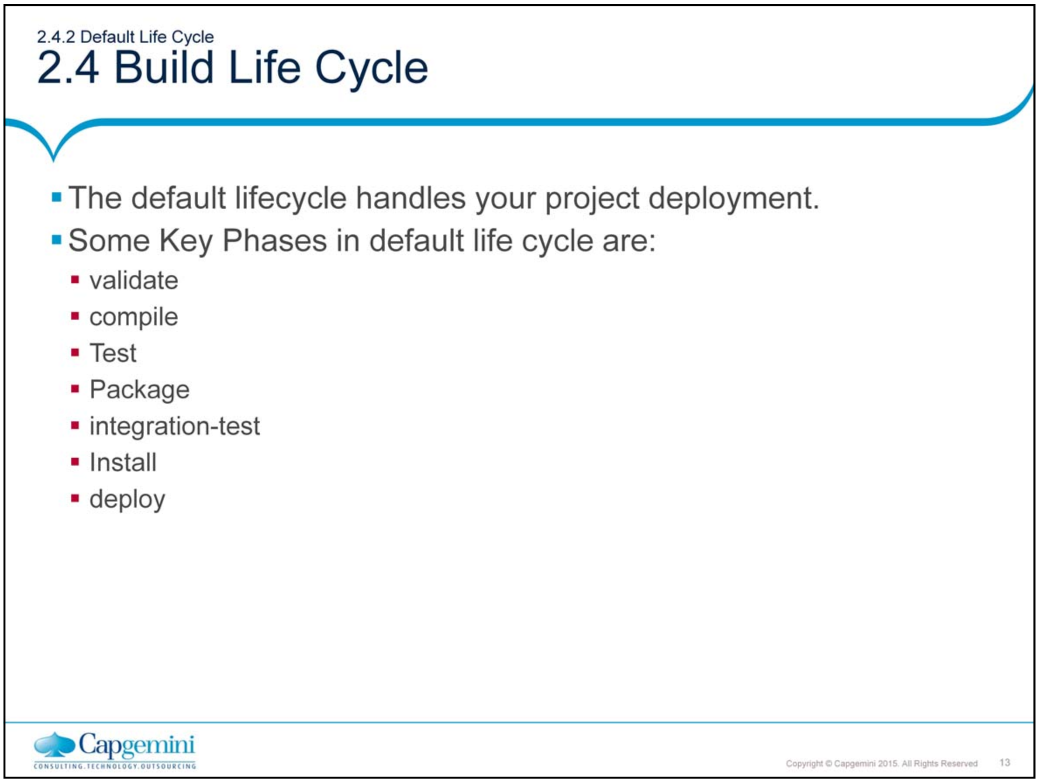
2.4.1 Clean Life Cycle

2.4 Build Life Cycle

- clean lifecycle handles the cleaning of all project files generated by a previous build.
- Running mvn clean invokes the clean lifecycle

pre-clean	**executes processes needed prior to the actual project cleaning**
clean	**remove all files generated by the previous build**
post-clean	**executes processes needed to finalize the project cleaning**

12



2.4.2 Default Life Cycle

2.4 Build Life Cycle

- The default lifecycle handles your project deployment.
- Some Key Phases in default life cycle are:
  - validate
  - compile
  - Test
  - Package
  - integration-test
  - Install
  - deploy

13

validate - validate the project is correct and all necessary information is available

compile - compile the source code of the project

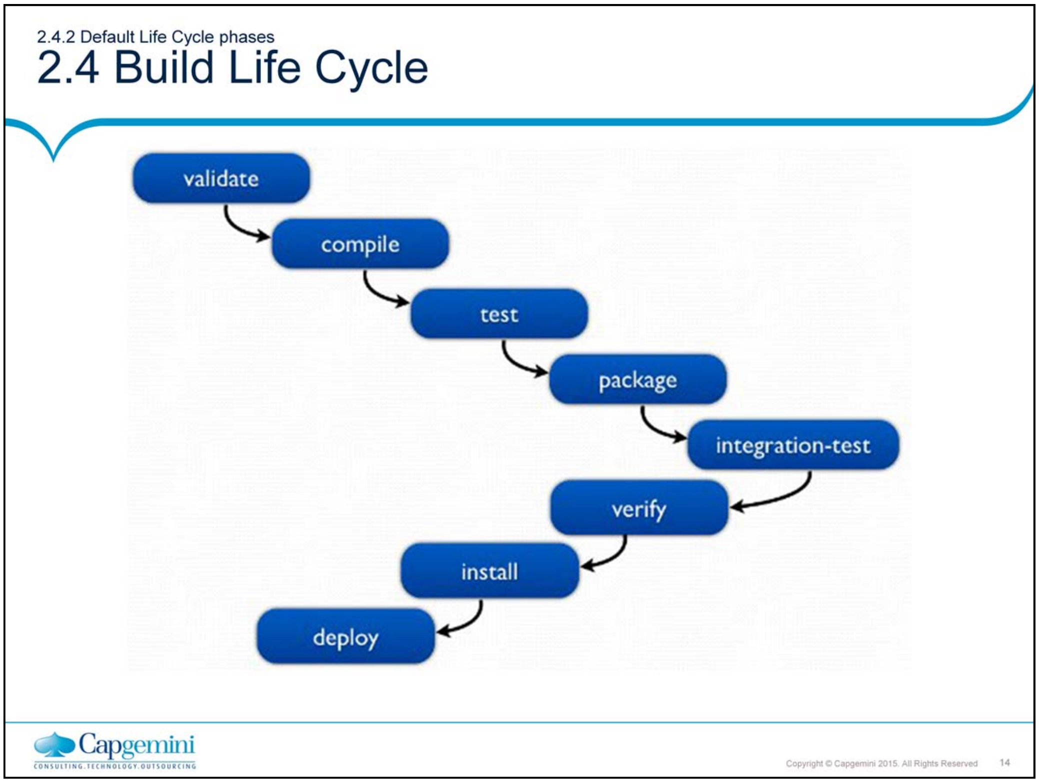
test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed

package - take the compiled code and package it in its distributable format, such as a JAR

integration-test - process and deploy the package if necessary into an environment where integration tests can be run

install - install the package into the local repository, for use as a dependency in other projects locally

deploy - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects



2.4.2 Default Life Cycle phases

2.4 Build Life Cycle

validate

compile

test

package

integration-test

verify

install

deploy

14

The default lifecycle of Maven first validates the project, like checking the consistency of pom.xml file, and then it tries to compile the code. If the compile is successful, it then tries to run the test against the compiled code, package the project in the specified format, run the integration tests against that package, verify the package by checking for the validity, install the verified package to the local repository, and finally deploy the package in the specified environment.

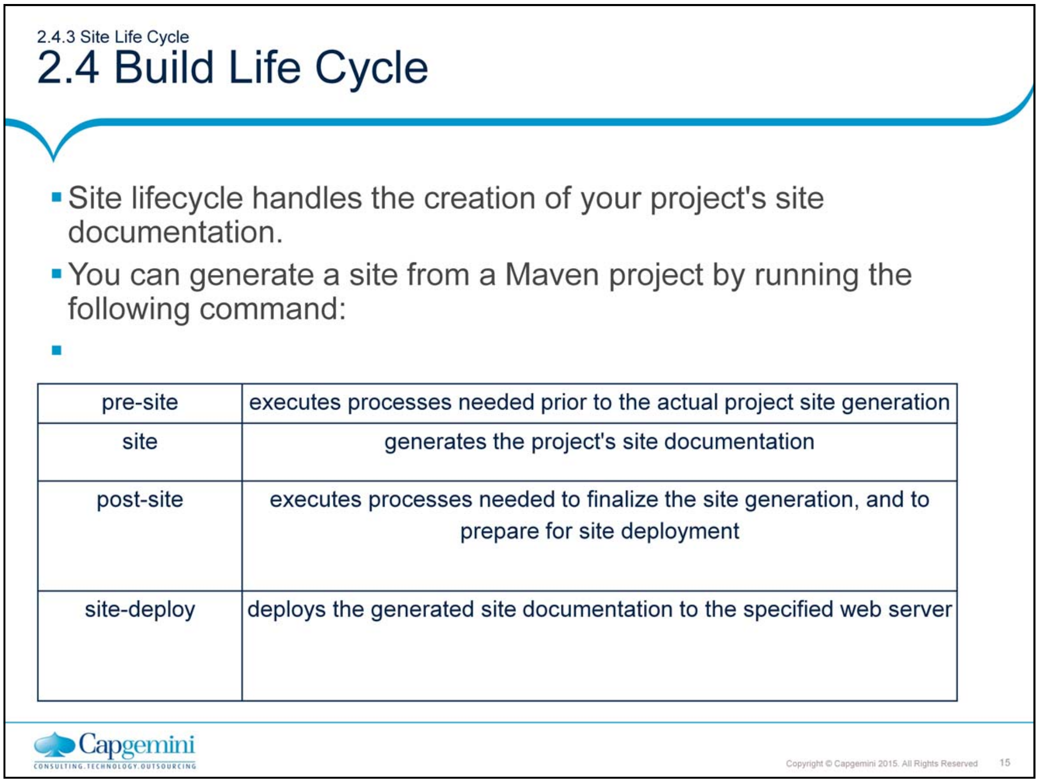
To execute phases/goals we can follow the notation below in the command line:

mvn *phase* mvn *phase:goal*

We can also invoke multiple phases/goals within one command line, like:

mvn *phase phase:goal* mvn *phase:goal phase:goal*

When we invoke the *mvn integration-test* command, Maven executes all the phases that are registered before that phase. So the validate, compile, and package phases get executed before the integration-test phase.



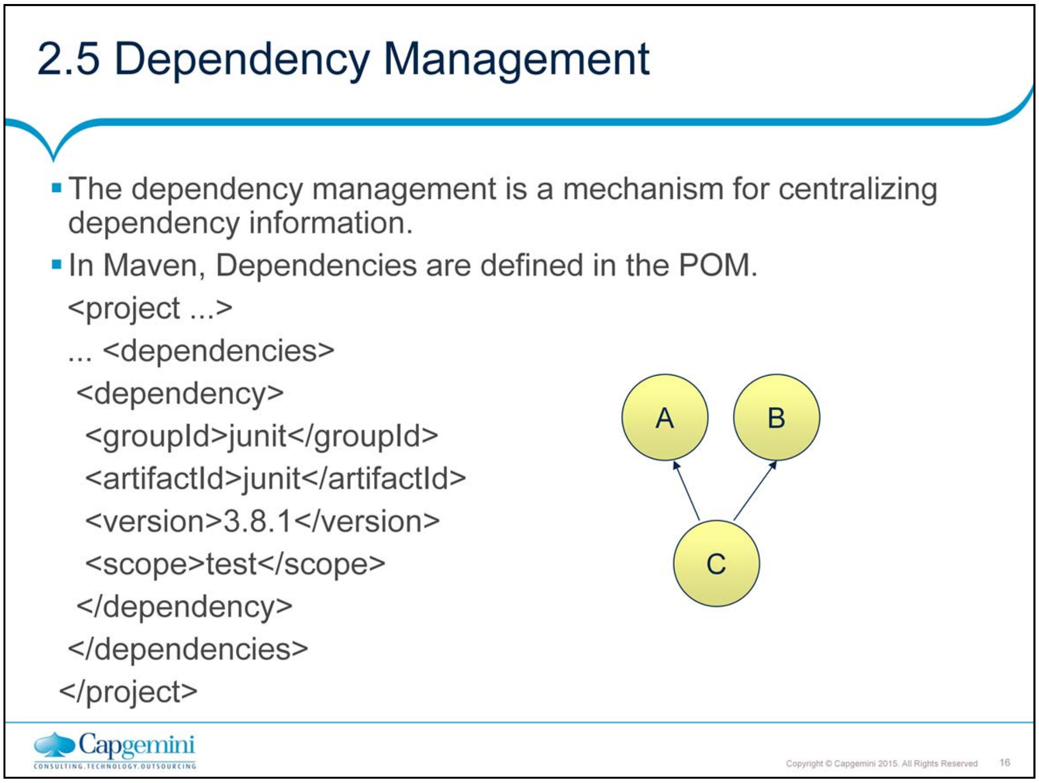
2.4.3 Site Life Cycle

2.4 Build Life Cycle

- Site lifecycle handles the creation of your project's site documentation.
- You can generate a site from a Maven project by running the following command:
-

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

15



2.5 Dependency Management

- The dependency management is a mechanism for centralizing dependency information.
- In Maven, Dependencies are defined in the POM.

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

16

- Dependency: a third-party or project-local software library (JAR or WAR file) of one project will be reused in another projects.
- In Maven, Dependencies are defined in the POM.

Dependency scope is used to limit the transitivity of a dependency, and also to affect the classpath used for various build tasks.

There are 6 scopes available:

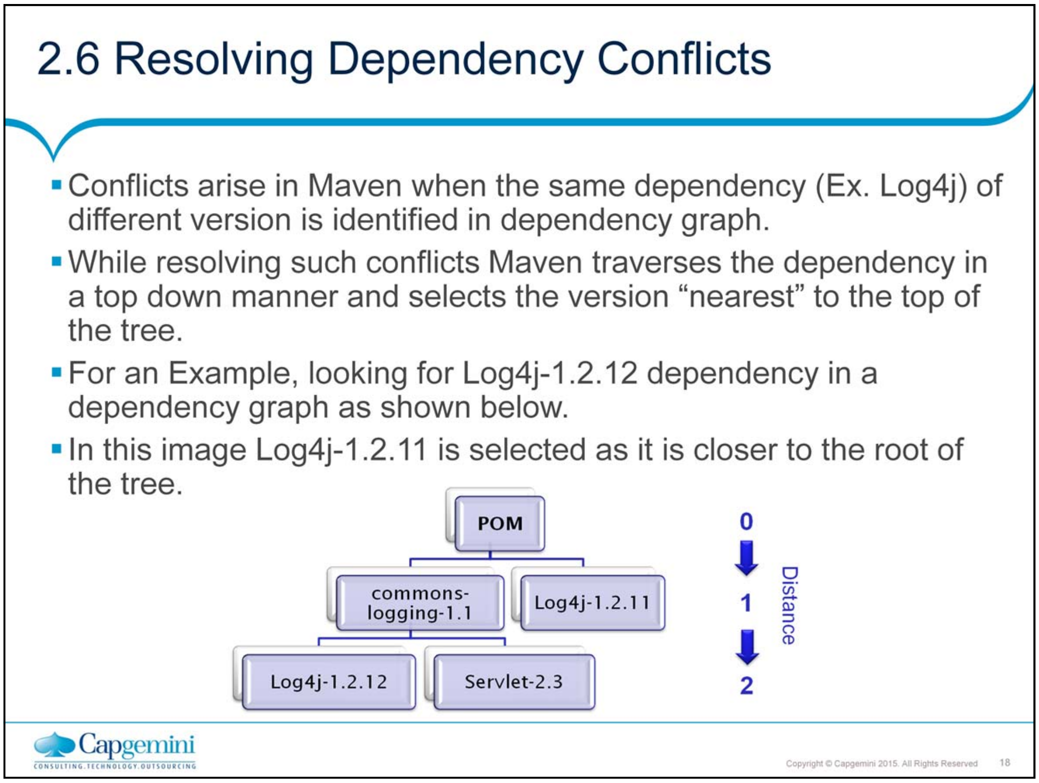
- **Compile** - This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.
- **Provided** - This is much like compile, but indicates you expect the JDK or a container to provide the dependency at runtime.

.For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope provided because the web container provides those classes. This scope is only available on the compilation and test classpath, and is not transitive.

- **Runtime -** This scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
- **Test -** This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.
- **System -** This scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.
- **import -** This scope is only used on a dependency of type pom in the <dependencyManagement> section. It indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

Transitive Dependencies:

- Resolves 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.
- Transitive dependencies allows you to avoid needing to discover and specify the libraries that your own dependencies require, and including them automatically.
- For example, if you define a dependency to commons-logging and commons-logging itself defines a dependency to log4j, the commons-logging.jar and log4j.jar will be added to your build process. More formally spoken, transitiveness means that if A->B and B->C then A->C



2.6 Resolving Dependency Conflicts

- Conflicts arise in Maven when the same dependency (Ex. 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.
- For an Example, looking for Log4j-1.2.12 dependency in a dependency graph as shown below.
- In this image Log4j-1.2.11 is selected as it is closer to the root of the tree.

As the tree grows, it is inevitable that two or more artifacts will require different versions of a particular dependency.

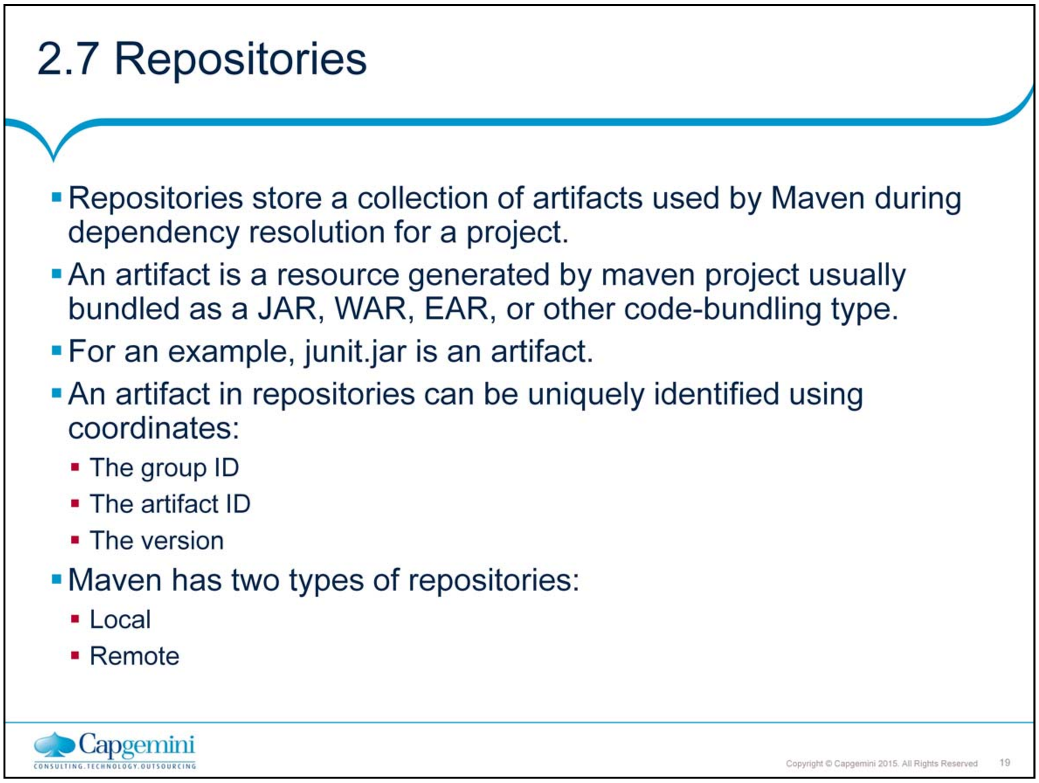
For an Example: The sample project defines two direct dependencies: One to commons-logging-1.1 and one to log4j-1.2.11. Now, because Maven transitively loads all dependencies that are defined for commons-logging-1.1, a second version of log4j (V1.2.12) pops up in the dependency tree.

Which dependency will be used by the build process?

Answer : log4j-1.2.11 is been loaded as it is the nearest to the top of the tree.

The dependency version to be used by the build process can be recommended as follows:

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
     <version>1.2.12</version>
</dependency>
```



2.7 Repositories

- Repositories store a collection of artifacts used by Maven during dependency resolution for a project.
- An artifact is a resource generated by maven project usually bundled as a JAR, WAR, EAR, or other code-bundling type.
- For an example, junit.jar is an artifact.
- An artifact in repositories can be uniquely identified using coordinates:
  - The group ID
  - The artifact ID
  - The version
- Maven has two types of repositories:
  - Local
  - Remote

19

The poor approach:

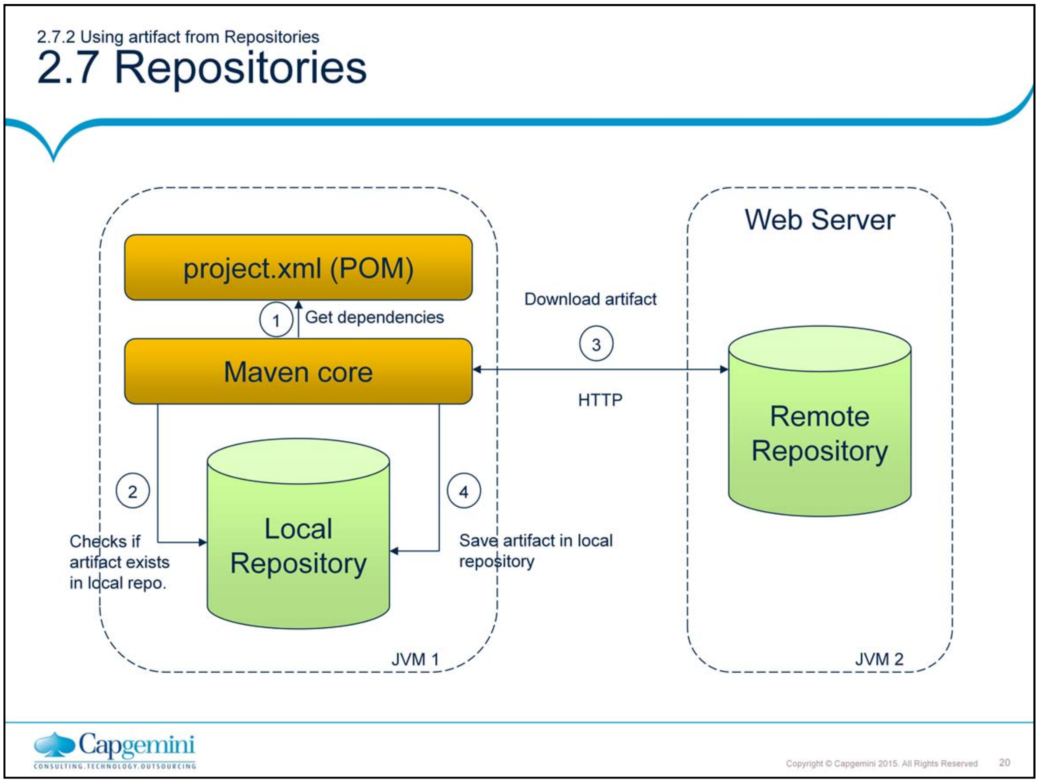
Replicating all dependencies for every project (put in /lib folder within the project)

– Dependencies are replicated and use more storage

– Checking out a project will be slow

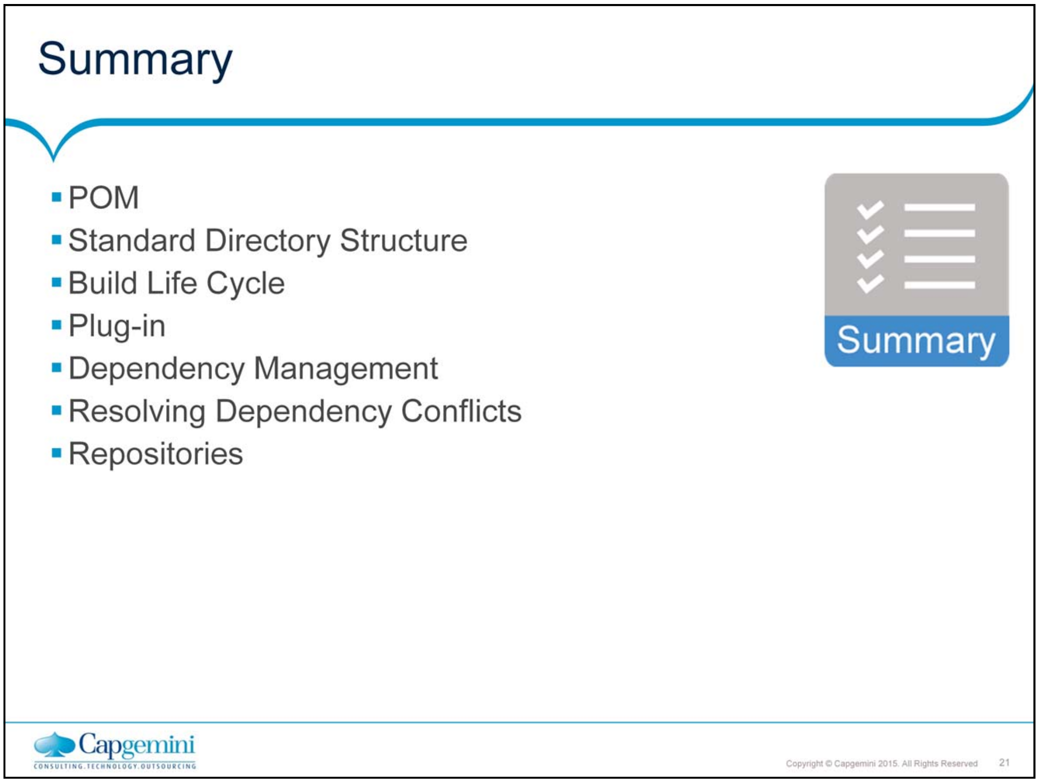
– Difficult to keep track of versions

• The preferred solution: Use a repository



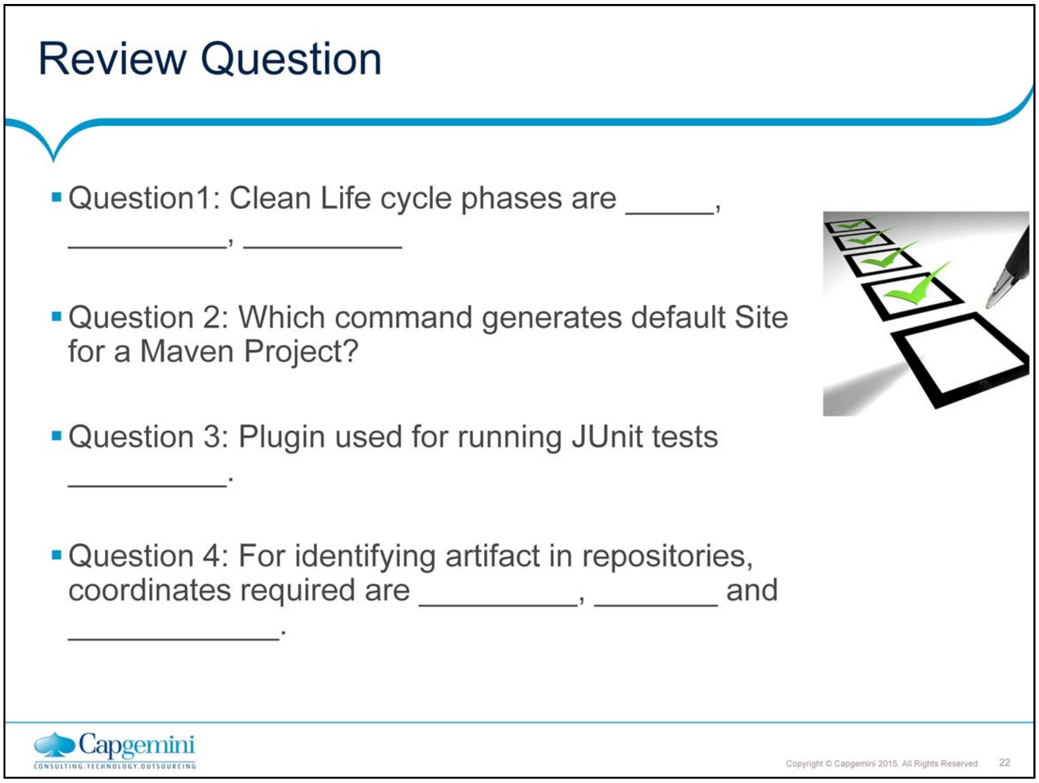
2.7.2 Using artifact from Repositories

2.7 Repositories



Summary

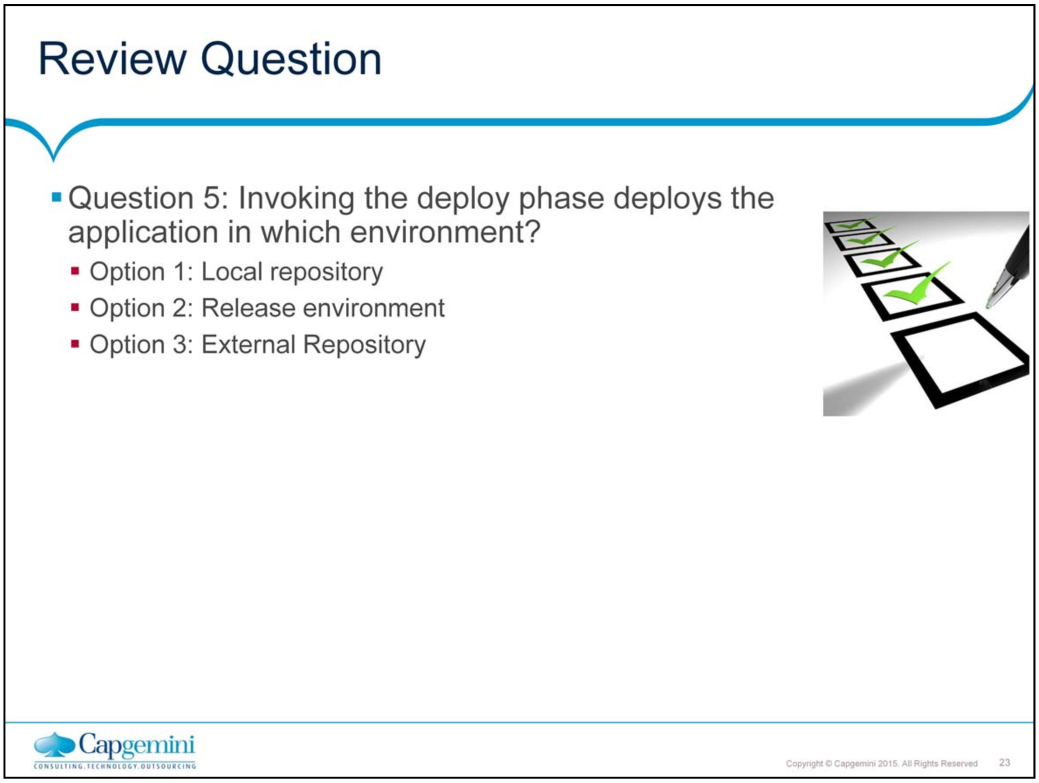
- POM
- Standard Directory Structure
- Build Life Cycle
- Plug-in
- Dependency Management
- Resolving Dependency Conflicts
- Repositories



Review Question

- Question1: Clean Life cycle phases are ______, __________, __________
- Question 2: Which command generates default Site for a Maven Project?
- Question 3: Plugin used for running JUnit tests __________.
- Question 4: For identifying artifact in repositories, coordinates required are __________, ________ and _____________.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



Review Question

- Question 5: Invoking the deploy phase deploys the application in which environment?
  - Option 1: Local repository
  - Option 2: Release environment
  - Option 3: External Repository