

# Maven Basics

Vijay Ramaswamy  
Test Architect & Consultant

# What you will learn...

- ▶ Introduction to Maven
- ▶ The POM.xml file
- ▶ The standard directory layout
- ▶ Maven archetypes
- ▶ The dependency mechanism:
  - ▶ Overview
  - ▶ Repositories
- ▶ The build lifecycle:
  - ▶ Overview
  - ▶ Goals and Plugins
- ▶ Maven Vs. Ant

# Pre-requisites

- ▶ You will need to be familiar with Java basics and Eclipse
- ▶ Familiarity with a build automation tool such as Ant would be useful

# Reference material

- ▶ <https://maven.apache.org/guides/>

Let's go!

# Introduction to Maven (~5 mins)

- ▶ A tool designed to build and manage any Java based project
- ▶ Core capabilities:
  - ▶ Build automation:
    - ▶ Automate processes such as compilation, packaging, running tests, etc.
  - ▶ Dependency management
  - ▶ Extensible through plugins
- ▶ Favors Convention over Configuration
- ▶ Widely accepted as a huge step forward from previous generation build automation tools such as Ant

# Introduction to Maven (~5 mins)

## ► Installing Maven:

- Eclipse comes pre-installed with Maven integration out of the box
- If you want to run Maven from the command line, you will need to install it separately outside of Eclipse:
  - Download link: <https://maven.apache.org/download.cgi>

## ► Running Maven:

- From the command line: Use the “mvn” command
  - From Eclipse: Use the “Run As->Maven build” menu option
  - From Jenkins: Using the Maven plugin for Jenkins
- Trivia: The word “Maven” means “accumulator of knowledge” in Yiddish

# The POM.xml file (~10 mins)

- ▶ POM = Project Object Model
- ▶ Basic unit of work in Maven
- ▶ XML file containing complete configuration details about the project
- ▶ Key elements within a POM.xml file:
  - ▶ groupId: Unique ID of the company/organization (E.g.: com.mycompany)
  - ▶ artifactId: Unique name of the build artifact to be generated (usually a JAR or WAR file)
  - ▶ version: Version number for the build artifact to be generated
  - ▶ packaging: The package type to be used while generating the build artifact (JAR/WAR/WAR, etc.)
- ▶ All these elements can be specified easily using the “Overview” tab for your POM file within Eclipse
- ▶ POM files can inherit from each other, so common configuration details can be collapsed into one place

mercurytours.pageObjectModel.dataNonIterative/pom.xml

### Overview

**Artifact**

Group Id: com.autopia4j.demo

Artifact Id: \*mercurytours.pageObjectModel.dataNonIterative

Version: 1.1.8

Packaging: jar

**Parent**

**Properties**

<project.build.sourceEncoding : UTF-8

Create... Remove

**Modules** New module element

**Project**

Name: object model and data non-iterative implementation demo

URL: https://bitbucket.org/autopiateam/autopia4j-webdriver-demos

Description: autopia4j implementation demo using the page object model and a non-iterative datatable

Inception:

**Organization**

Name: Autopia Team

URL: https://bitbucket.org/autopiateam

**SCM**

**Issue Management**

System: BitBucket

URL: ia4j-webdriver-demos/issues?status=new&status=open

**Continuous Integration**

Overview Dependencies Dependency Hierarchy Effective POM pom.xml



# The POM.xml file (contd.)

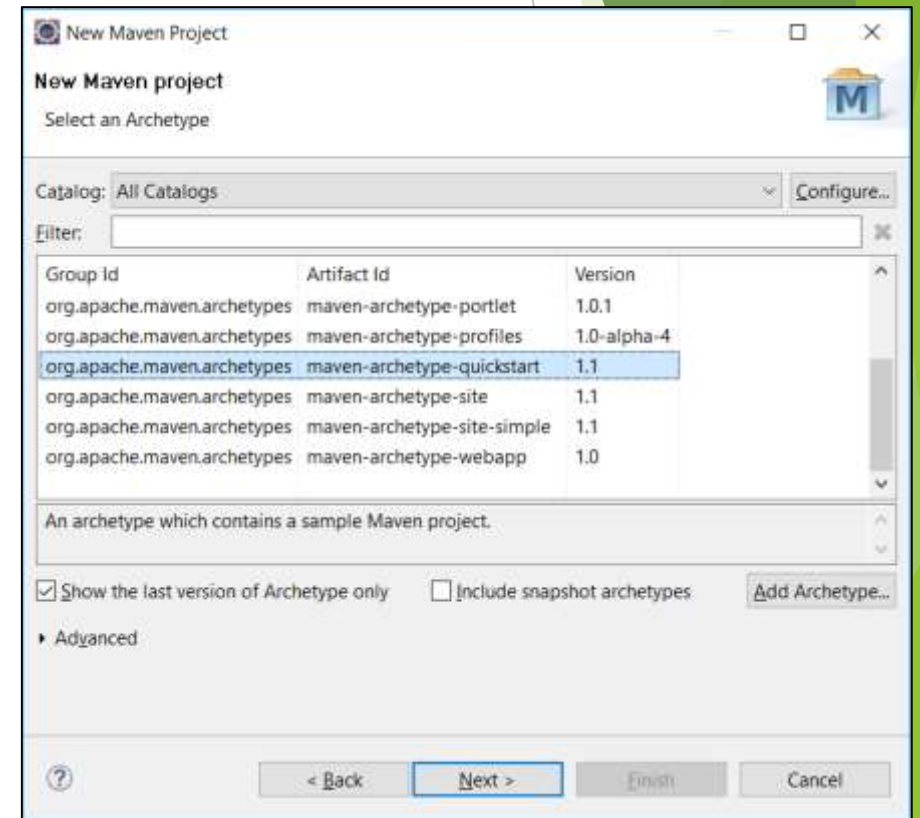
- ▶ Super POM:
  - ▶ Maven's default POM
  - ▶ All project specific POM's inherit from the Super POM by default:
    - ▶ The Super POM provides default configurations that are suitable for most projects
    - ▶ This helps to keep the project specific POM's really compact
    - ▶ Eclipse has an "Effective POM" tab, which combines the Super POM and the project specific POM into one view for easy reference
  - ▶ You can also create your own parent POM's if required

# The standard directory layout (~5 mins)

- ▶ One of the key goals of Maven is to standardize the structure of Java projects
- ▶ The standard directory layout is a common layout recommended by Maven to enable Java developers to move between Maven projects with little to no context switching
- ▶ Basic structure:
  - ▶ Root level files:
    - ▶ POM.xml
    - ▶ README files, LICENSE files, etc.
    - ▶ Metadata files such as .gitignore, .classpath, .project, etc.
  - ▶ Basic structure (contd.):
    - ▶ Root level directories:
      - ▶ “src” -> Directory to store source code and related artifacts
      - ▶ “target” -> Directory to store all output from a build
    - ▶ Sub-directories:
      - ▶ “src/main” -> Directory to store development source code and related artifacts
      - ▶ “src/test” -> Directory to store unit testing source code and related artifacts
      - ▶ “src/main/java” & “src/test/java” -> Directories to store Java source code
      - ▶ “src/main/resources” & “src/test/resources” -> Directories to store resources such as configuration files which are accessed by the source code

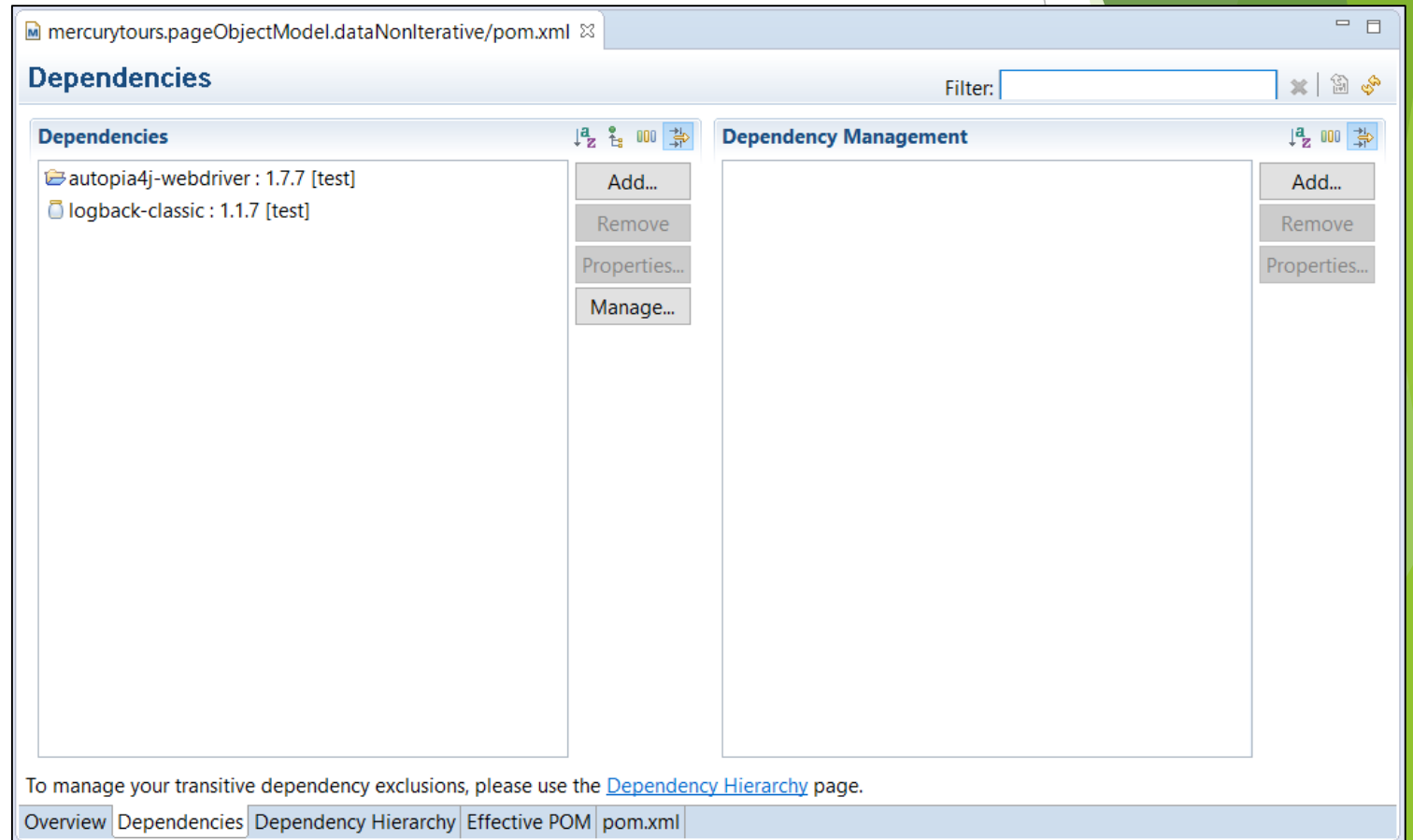
# Maven archetypes (~5 mins)

- ▶ A toolkit for Maven project templating:
  - ▶ Choose from a list of available templates while creating a new Maven project
- ▶ Helps users get up and running in the shortest time possible:
  - ▶ Automatically sets up the project as per the defined directory layout
  - ▶ Automatically generates the POM.xml file
  - ▶ Automatically sets up default dependencies for the project
- ▶ Maven comes with several in-built archetypes for common project types such as web applications, J2EE applications, etc.
  - ▶ Eclipse provides a list of available archetypes at the time of creating a new Maven project
  - ▶ Try using the “quickstart” archetype as an example
- ▶ It is also possible to create our own custom archetypes:
  - ▶ This is useful for standardizing best practices across an organization



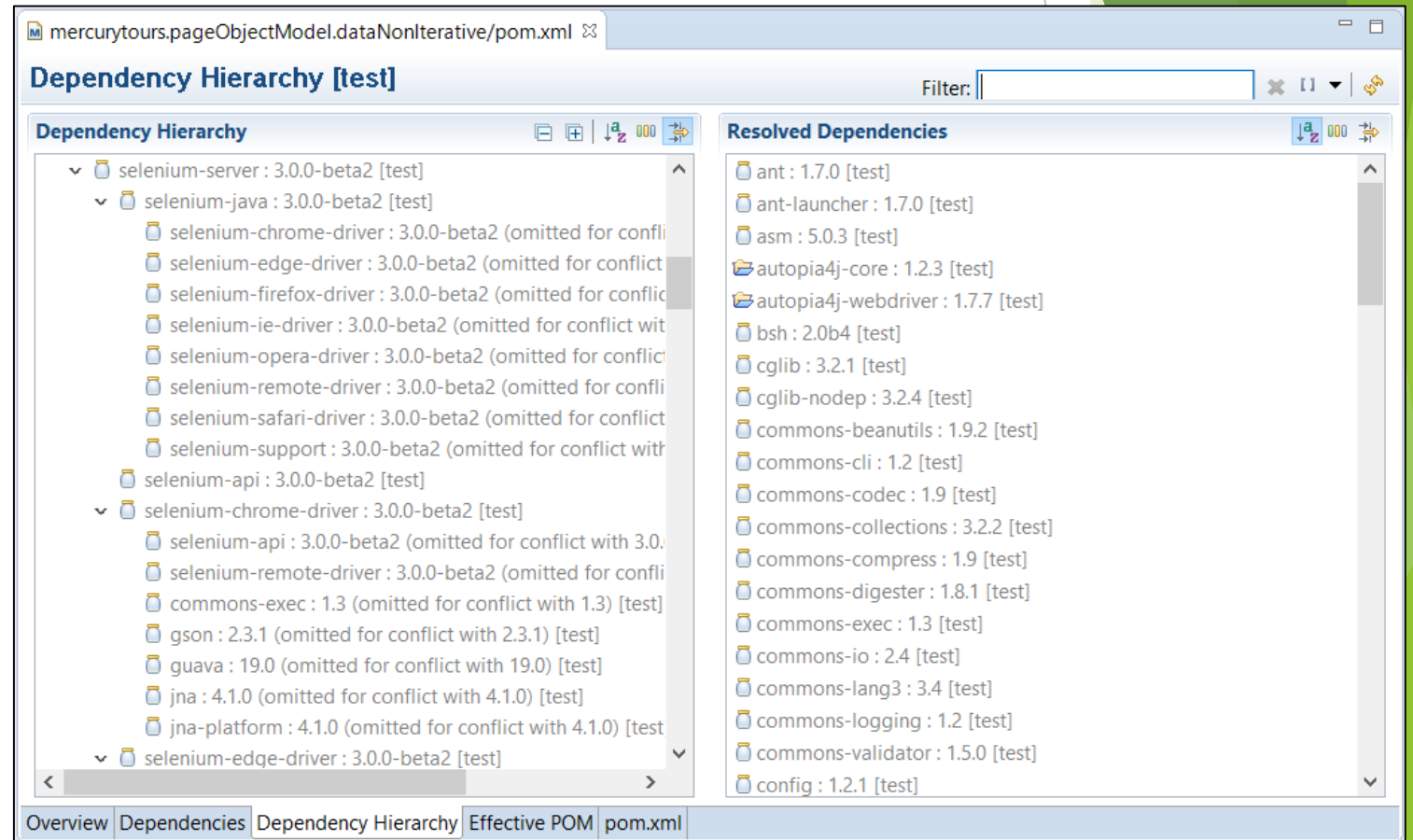
# The dependency mechanism: Overview (~5 mins)

- ▶ One of the key features of Maven
- ▶ Project dependencies are managed using the POM.xml file:
  - ▶ Eclipse provides the “Dependencies” tab to enable you to add dependencies to your POM.xml file easily
  - ▶ Each dependency should be assigned a “scope” -> Most common ones are “compile” and “test”



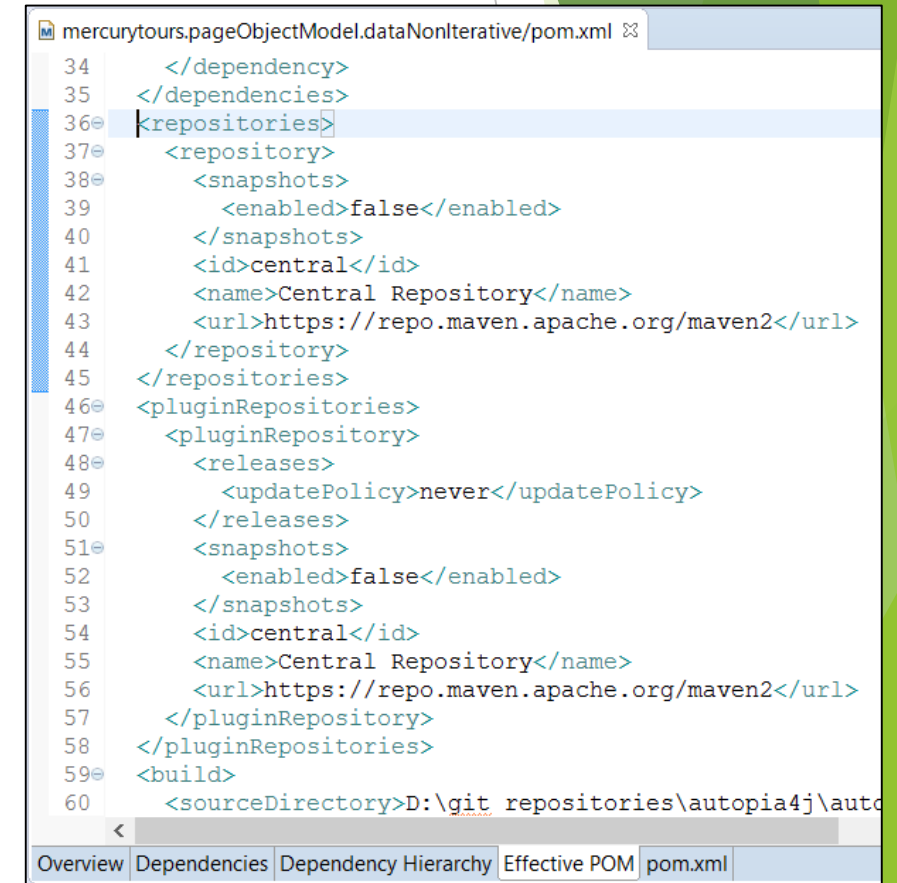
# The dependency mechanism: Overview (contd.)

- ▶ Maven builds a complete tree of dependencies for your project, and configures downstream dependencies for each of your primary dependencies:
  - ▶ Eclipse provides the “Dependency Hierarchy” tab to help manage the dependency tree for your POM.xml file
  - ▶ Note that the sequence of dependencies matters while building the dependency tree



# The dependency mechanism: Repositories (~10 mins)

- ▶ Maven stores all your dependencies in a centralized repository on the local machine, known as the m2 repository:
  - ▶ By default, this is available @ “%userprofile%\m2”
- ▶ When you add a dependency to your project, Maven automatically downloads the dependency and stores it in your local m2 repository:
  - ▶ This explains why it takes a long time when you import or build a project for the first time - that is Maven downloading everything behind the scenes
- ▶ By default, Maven downloads all dependencies from “[Maven Central](#)”, which is a common repository used by Maven users around the world:
  - ▶ This is specified within the Super POM
  - ▶ This can be overridden if required within the project specific POM
- ▶ In the same vein, it is also possible to upload/publish any artifacts that you create into a remote repository such as Maven Central, so that other developers can leverage your work as a dependency



```
mercury.tours.pageObjectModel.dataNonIterative/pom.xml
34 </dependency>
35 </dependencies>
36 <repositories>
37 <repository>
38 <snapshots>
39 <enabled>>false</enabled>
40 </snapshots>
41 <id>central</id>
42 <name>Central Repository</name>
43 <url>https://repo.maven.apache.org/maven2</url>
44 </repository>
45 </repositories>
46 <pluginRepositories>
47 <pluginRepository>
48 <releases>
49 <updatePolicy>never</updatePolicy>
50 </releases>
51 <snapshots>
52 <enabled>>false</enabled>
53 </snapshots>
54 <id>central</id>
55 <name>Central Repository</name>
56 <url>https://repo.maven.apache.org/maven2</url>
57 </pluginRepository>
58 </pluginRepositories>
59 <build>
60 <sourceDirectory>D:\git_repositories\autopia4j\auto
```

Overview Dependencies Dependency Hierarchy Effective POM pom.xml

# The build lifecycle: Overview (~5 mins)

- ▶ Maven is based around the central concept of a build lifecycle:
  - ▶ A build lifecycle defines the process for building and distributing a particular artifact
- ▶ Maven has 3 default lifecycles built-in:
  - ▶ clean -> Clean up the project
  - ▶ default -> Deploy the project
  - ▶ site -> Create the project's site documentation
- ▶ Each lifecycle consists of multiple build “phases”, which represent various stages within the lifecycle:
  - ▶ Build phases are executed sequentially in a defined order
- ▶ “clean” lifecycle phases:
  - ▶ pre-clean
  - ▶ clean (remove all files generated by previous build)
  - ▶ post-clean

# The build lifecycle: Overview (contd.)

- ▶ “default” lifecycle phases:
  - ▶ validate (check for errors)
  - ▶ compile (compile source code)
  - ▶ test (run unit tests)
  - ▶ package (generate JAR/WAR)
  - ▶ verify (run integration tests)
  - ▶ install (copy JAR/WAR to local repo)
  - ▶ deploy (upload JAR/WAR to remote repo)
- ▶ Use the “mvn <lifecycle-phase>” command to execute a build lifecycle up to the specified phase:
  - ▶ mvn install -> execute the default lifecycle until the “install” phase
  - ▶ mvn clean install -> execute the clean lifecycle followed by the default lifecycle



# The build lifecycle: Goals and Plugins (~10 mins)

- ▶ Each build phase within a lifecycle is made up of one or more “goals”:
  - ▶ If a build phase has zero goals, it is not executed
- ▶ Goals are fine grained tasks that are executed by so-called “plugins”:
  - ▶ Maven is - at its heart - a plugin execution framework; all work is done by plugins
  - ▶ Maven has a huge eco-system of plugins which handle various different aspects of the build process through specific goals
  - ▶ A plugin can have multiple goals if required
- ▶ Maven manages plugins the same way it manages dependencies:
  - ▶ Plugins are configured within the POM.xml file
  - ▶ Plugins are downloaded from a specified remote repository and stored in your local m2 repository
- ▶ Gotcha: Sometimes, a build phase, plugin and goal may all have the exact same name!

# The build lifecycle: Goals and Plugins (contd.)

## ► Commonly used plugins and goals:

Plugin	Goal	Goal Description	Build lifecycle phase
compiler	compile	Compile dev source code	Compile
	testCompile	Compile test source code	compile
surefire	test	Run unit tests	test
jar	jar	Package dev source code into JAR file	package
	testJar	Package test source code into JAR file	package
failsafe	integration-test	Run integration tests	verify
	verify	Verify integration test results	verify
javadoc	javadoc	Generate Javadoc for dev code	Not bound to a phase
	test-javadoc	Generate Javadoc for test code	Not bound to a phase
install	install	Install the project's main artifact (JAR/WAR, etc.), its POM, and any attached artifacts (such as Javadoc) into the local m2 repository	install
deploy	deploy	Install the project's main artifact (JAR/WAR, etc.), its POM, and any attached artifacts (such as Javadoc) into the specified remote repository	deploy

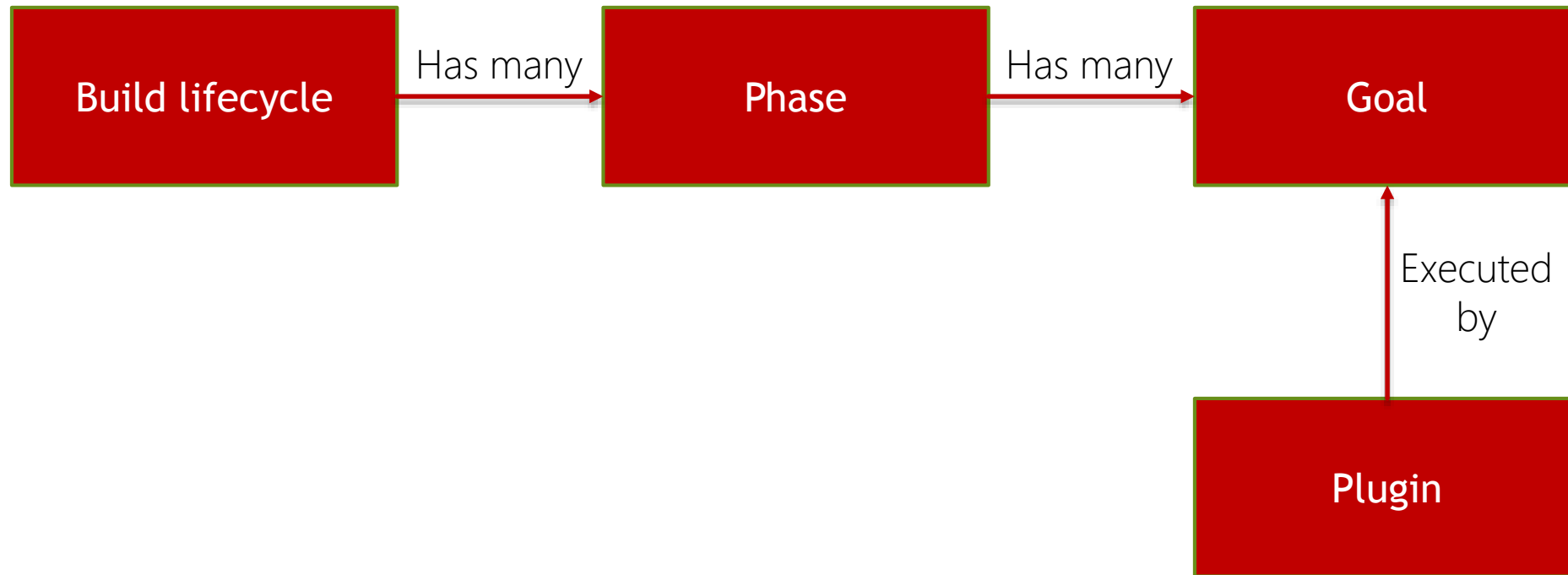
# The build lifecycle: Goals and Plugins (contd.)

- ▶ Use the “mvn <plugin>:<goal>” command to execute a specific plugin goal:
  - ▶ E.g.: mvn compiler:compile, mvn jar:jar, etc.
- ▶ A more recommended way to execute a plugin goal is to “bind” it to any of the pre-defined build lifecycle phases, and then execute the lifecycle until the desired phase using the “mvn <lifecycle-phase>” command
- ▶ Note that Maven automatically binds most of the basic goals to various phases of the build lifecycle:
  - ▶ These default bindings are configured within the Super POM file
  - ▶ Refer to the table on the previous slide to see some of the default bindings
- ▶ It is also possible to manually bind plugin goals to phases within your project specific POM file



```
api.test.framework/pom.xml 33
46     <plugin>
47       <groupId>com.lazerycode.jmeter</groupId>
48       <artifactId>jmeter-maven-plugin</artifactId>
49       <version>2.0.3</version>
50       <executions>
51         <execution>
52           <id>jmeter-tests</id>
53           <phase>verify</phase>
54           <goals>
55             <goal>jmeter</goal>
56           </goals>
57         </execution>
58       </executions>
59     </plugin>
60 </plugins>
61 </build>
62
```

# The build lifecycle: Summary



# Maven Vs. Ant (~5 mins)

- ▶ Similarities:
  - ▶ Command line execution
  - ▶ Eclipse integration
  - ▶ XML based configuration
- ▶ Maven benefits over Ant:
  - ▶ Maven POM.xml files are usually more compact than Ant build.xml files (because of the Super POM)
  - ▶ Better dependency management (network based)
  - ▶ Standardized directory layout
  - ▶ Archetypes for templating

# Maven Vs. Ant (contd.)

- ▶ Rough equivalents between Maven and Ant:
  - ▶ Maven build phase ~ Ant target
    - ▶ The main difference is that build phases are pre-defined in Maven, whereas Ant targets need to be manually defined
  - ▶ Maven goal ~ Ant task
  - ▶ Maven plugin ~ Ant plugin
- ▶ Migration path from Ant to Maven:
  - ▶ Adopt the standard directory layout
  - ▶ Identify your dependencies and map them out into your new POM.xml file
  - ▶ Identify any plugins required and map them out into your new POM.xml file
  - ▶ If required, use the Ant plugin for Maven to continue using key targets from your existing Ant build files

# Other topics of interest

- ▶ Maven modules
- ▶ Managed dependencies
- ▶ Optional dependencies and dependency exclusions
- ▶ Build profiles

# Recap

In this module, you learnt:

- ▶ Introduction to Maven
- ▶ The POM.xml file
- ▶ The standard directory layout
- ▶ Maven archetypes
- ▶ The dependency mechanism:
  - ▶ Overview
  - ▶ Repositories
- ▶ The build lifecycle:
  - ▶ Overview
  - ▶ Goals and Plugins
- ▶ Maven Vs. Ant



The background features abstract, overlapping green geometric shapes in various shades, creating a modern and dynamic visual effect. The shapes are primarily triangular and polygonal, with some areas appearing more translucent than others.

# THANK YOU!

Vijay Ramaswamy  
Test Architect & Consultant