Community Experience Distilled

Maven for Eclipse

A fast-paced guide that helps you create a continuous delivery solution by integrating Maven with an Eclipse environment

Sanjay Shah

[PACKT] open source
PUBLISHING
community experience distilled

Maven for Eclipse

A fast-paced guide that helps you create a continuous delivery solution by integrating Maven with an Eclipse environment

Sanjay Shah

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Maven for Eclipse

First published: August 2014

Production reference: 1190814

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-712-2

www.packtpub.com

Cover image by Asher Wishkerman (wishkerman@hotmail.com)

Credits

Author
Sanjay Shah

Reviewers
Patrick Forhan
Peter Johnson
Luca Masini
Maurizio Pillitu
Bhavani P Polimetla

Commissioning Editor
Amarabha Banerjee

Acquisition Editor
Vinay Argekar

Content Development Editor
Shali Sasidharan

Technical Editors
Shruti Rawool
Aman Preet Singh

Copy Editors
Mradula Hegde
Gladson Monteiro

Project Coordinator
Neha Bhatnagar

Proofreaders
Simran Bhogal
Maria Gould
Ameesha Green

Indexer
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Sanjay Shah has more than 9 years of experience of working in diverse areas of application development across mobile and web platforms. He is currently working as a software architect and has a number of enterprise applications to his name.

He is the co-author of the book *Android Development Tools for Eclipse*, *Packt Publishing*, also co-authored by Khirulnizam Abd Rahman.

Along with being a literature enthusiast, he is fond of philosophy and enjoys life in Nepal, the land of the highest peak in the world, Mt. Everest.

I would like to thank each and everyone who knows me and has supported me at different stages of my life. Special thanks to my parents without whom I wouldn't have been what I am today.

About the Reviewers

Patrick Forhan is a grizzled Java developer and an occasional accidental snake handler. He posts ideas and articles at `http://muddyhorse.com`.

Peter Johnson has over 34 years of experience in enterprise computing. He has been working with Java for 17 years and has been heavily involved in Java performance tuning for the past 12 years. He is a frequent speaker on Java performance topics at various conferences, including the Computer Measurement Group annual conference, JBoss World, and Linux World. He is a moderator for the IDE and WildFly/JBoss forums at JavaRanch. He is the co-author of the book *JBoss in Action*, *Manning Publications*, also authored by Javid Jamae, and has been a reviewer on numerous books on topics that range from Java to Windows PowerShell.

Luca Masini is a senior software engineer and an architect, born as a game developer for Commodore 64 (Football Manager) and Commodore Amiga (Ken il guerriero). He soon switched to object-oriented programming and after that, from its beginning in 1995, he was fascinated by the Java language.

He worked on this passion as a consultant for major Italian banks, developing and integrating the main software projects for which he usually was the technical lead. He adopted Java Enterprise in environments, where COBOL was the flagship platform, and converted them from mainframe-centric to distributed environments.

He then shifted his attention toward open source, starting from Linux and then to enterprise frameworks. With enterprise frameworks, he was able to introduce concepts such as IoC, ORM, and MVC with low impact. For that, he was an early adopter of Spring, Hibernate, Struts, and a whole host of other technologies, which in the long run, have given his customers a technological advantage and therefore, reduced development costs.

After introducing a new technology, he decided that it was time for simplification and standardization of development with Java EE, and for this, he's now working at the ICT of a large Italian company where he introduced build tools (Maven and Continuous Integration), archetypes of projects, and Agile Development with plain standards.

Finally, he focused his attention on mobilizing the enterprise, and now he is working on a whole set of standard and development processes to introduce mobile concepts and applications for sales force and management.

He has worked on the following books from Packt Publishing:

- *Securing WebLogic Server 12c,* co-authored by *Rinaldi Vincenzo*
- *Google Web Toolkit GWT Java AJAX Programming*, *Prabhakar Chaganti*
- *Spring Web Flow 2 Web Development, Sven Lüppken and Markus Stäuble*
- *Spring Persistence with Hibernate, Ahmad Reza Seddighi*
- *JavaFX 1.2 Application Development Cookbook, Vladimir Vivien*

Maurizio Pillitu has over 12 years of experience in the ICT industry, mostly related to open source technologies. In these 12 years, he has held different positions: Software Developer / Designer / Architect, Sales Engineer, Technical Trainer, and Project and Team Leader.

Through experience and education, he tried to push the Agile approach, thus providing a smooth path for change to the customer, incentivizing strong collaboration, and carefully managing the expectations of both parties.

He is passionate about application lifecycle management and frequently advises teams on how to structure software releases and deliveries in an automated and sustainable way.

He has wide knowledge of J2EE technologies and related open source frameworks, especially of Enterprise Content Management frameworks/products and large-scale web publishing platforms.

He is always keen on contributing code and ideas to the open source communities.

The following are his specialties:

- Team behavior and dynamics (Certified Scrum Master)
- Application Lifecycle Management and build tools (Maven and Puppet trainer)
- ECM/CMS open source solutions (ACA and ACE certifications)

You can contact him at `http://www.linkedin.com/in/mpillitu`.

Bhavani P Polimetla has been learning and working in the IT Industry since 1990. He graduated with a Bachelor's degree in Computer Science and a Master's degree in Computer Applications from Andhra University, India. He has worked on standalone Swing applications to grid computing and multi-tier architecture. He has worked with top clients of the world, including three from Fortune 50 companies. At present, he is working as a software architect in Mountain View, California, USA.

To demonstrate his skills, he has completed over 25 certifications in the subjects of spectrum of Java, Database, Project Management, and Architecture. He has also achieved lots of awards for many of his projects. He spends his free time indulging in social service activities. To learn more about him, you can visit his website at `www.polimetla.com`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Maven for Eclipse is an indispensable guide to help you understand and use Maven from within Eclipse IDE using the m2eclipse plugin. By no means is it an in-depth and comprehensive resource. Rather, it's a quick and handy guide toward Maven project's development. It starts with the basics of Apache Maven; covers core concepts; and shows you how to create, import, build, run, package, and customize to generate project artifacts of Maven projects using the m2eclipse plugin inside the Eclipse IDE.

What this book covers

Chapter 1, *Apache Maven – Introduction and Installation*, provides users with a quick introduction and installation reference to Apache Maven. By the end of this chapter, users will have a Maven project running on their systems.

Chapter 2, *Installing m2eclipse*, serves as a reference for users to install the m2eclipse plugin and also provides Maven integration for Eclipse. By the end of this chapter, users will have m2eclipse installed on their systems and ready to be used.

Chapter 3, *Creating and Importing Projects*, starts with the Maven project structure, introduces core aspects and concepts, and guides you toward creating and importing Maven projects using the m2eclipse plugin. By the end of this chapter, users will be familiar with the core concepts of the Maven project structure, and they'll be able to create and import Maven projects.

Chapter 4, *Building and Running a Project*, introduces users to different build lifecycles and teaches them how to view the m2eclipse console and build and run projects. By the end of this chapter, users will be familiar with the build lifecycle and will be competent at building and running projects using m2eclipse.

Chapter 5, *Spicing Up a Maven Project*, teaches users to create a simple web application, shows ways to customize it, and provides guides on how to write and run unit tests. By end of this chapter, users will learn to create web applications using m2eclipse and change the POM file to generate reports against unit tests.

Chapter 6, *Creating a Multimodule Project*, intends to introduce the concept of multimodule projects and teaches users to create, build, and run the project. At the end of this chapter, users will know how to create and run a multimodule Maven project using the m2eclipse plugin.

Chapter 7, *Peeking into m2eclipse*, dives into the m2eclipse plugin and introduces different features and aspects that makes life easier. By the end of this chapter, users will be familiar with every aspect of m2eclipse and will be able to use it efficiently and with ease.

What you need for this book

It is recommended that you have a laptop or a desktop with the following specifications for the best performance during development:

- 4 GB RAM
- Windows OS 7 / Ubuntu 12.04 / Mac OS Maverick
- Dual core / iSeries processor
- Internet connection

Who this book is for

This book is aimed at beginners and existing developers who want to learn how to use Maven for Java projects. It is assumed that you have experience in Java programming and that you have used an IDE for development.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Plugins and goals can be included declaratively in the `pom` file to customize the execution of a project."

A block of code is set as follows:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packt.mvneclipse</groupId>
    <artifactId>mvneclipse</artifactId>
    <version>1.2</version>
</project>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<!--General project Information -->
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.mvneclipse</groupId>
  <artifactId>hello-project</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>hello-project</name>
  <url>http://maven.apache.org</url>
  <properties>1
  <project.build.sourceEncoding>UTF8</project.build.sourceEncoding>
</properties>

<repositories>
  <repository>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Maven Repository Switchboard</name>
    <url>http://repo1.maven.org/maven2</url>
  </repository>
</repositories>
```

Any command-line input or output is written as follows:

```
set PATH =%PATH%;%M2_HOME%\bin
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "To make m2eclipse use the external Maven, navigate to **Window** | **Preference** in Eclipse, and the **Preference** window appears."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission

will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Apache Maven – Introduction and Installation

"A journey of a thousand miles starts with a single step", Lao Tzu. Rightly so, if you are reading this sentence here, you have taken a step towards a journey of Maven with Eclipse. As part of this journey, in the very first chapter, we will introduce you to Maven and its basic architecture and then guide you through the installation process through the following subtopics:

- Introduction to Maven
- Maven's origin
- Maven's principles
- Maven's component architecture
- Maven versus Ant
- Downloading Maven
- Installing Maven
- Installing Maven on Windows
- Installing Maven on Linux and Mac OS
- Verifying the installation of Maven

Introduction to Maven

Apache Maven's official site states that **Apache Maven**, which is also known as **Maven**, is a software project management and comprehension tool. Generally, software project management comprises planning, organizing, managing resource pools, and developing resource estimates; hence, it is a meaningless abstraction to justify Maven offerings. To put it in simple words, Maven is a comprehensive approach towards the process of applying patterns to a build infrastructure with primary goals as follows:

- Easing the build process
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practice development
- Allowing transparent migration to new features

In order to achieve the preceding goals, Maven provides a set of build standards, an artifact repository model, an engine that describes projects, and a standard lifecycle to build, test, and deploy project artifacts.

Maven's origin

Maven, a Yiddish word that means *accumulator of knowledge*, was initially started as an attempt to simplify the build processes in the Jakarta Turbine project. Prior to Maven, Ant was the build tool used across projects, and there were different Ant build files across different projects. Also, there were no standard or consistent Ant build files for projects, and JARs were also required to be checked in subversion. Hence, there was a growing necessity to standardize the project's build process and its structure, publish project information, and reuse JARs across projects, which resulted in the formation of a new tool, Maven. Maven has made the day-to-day work of developers easy, and it provides comprehension of any Java project.

Maven's principles

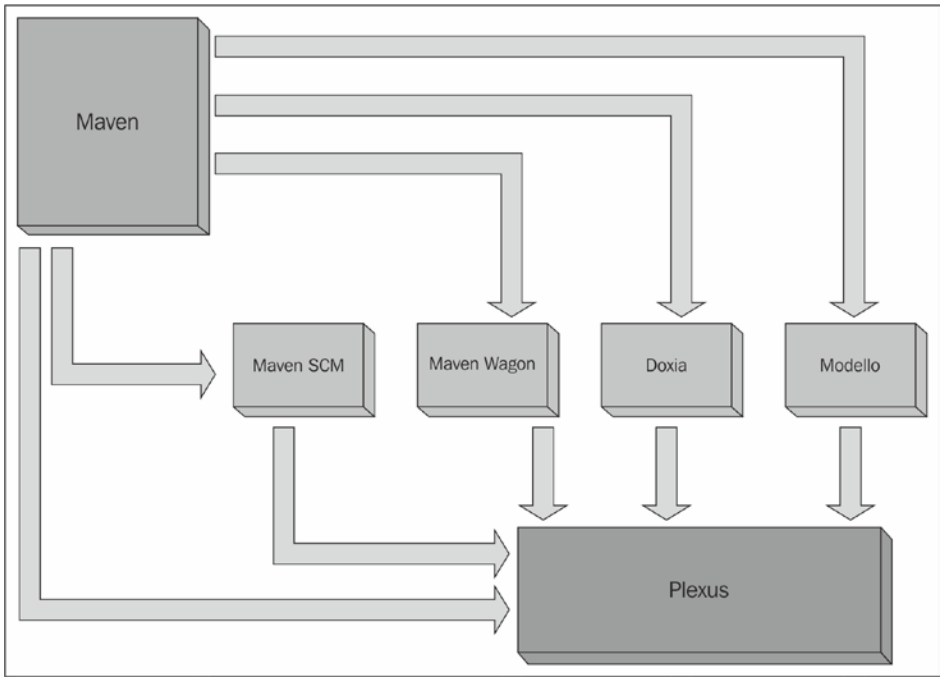
Maven's principles can be stated in the following points:

- **Convention over configuration**: Maven defines the default project structure and builds a life cycle that eases the burden during development. By specifying a publicly defined model, it makes the project more understandable.
- **Declarative execution**: Maven defines a build life cycle that comprises phases, which in turn are made up of plugin goals. Plugins and goals can be included declaratively in the `pom` file to customize the execution of a project.

- **Reusability**: Maven was built with reusability in mind. The build and execution declaration in one project can be used across different projects. Maven also makes it easier to create a component and integrate it into a multiproject build system. Also, with Maven Best Practices, development across the industry is encouraged.
- **Coherent organization of dependency**: Maven takes care of dependency management, thus reducing the burden on the part of developers. Different conflicts across dependencies are also handled beautifully.
- **Focus on writing applications**: With a standard project layout and build lifecycle, there is no need to develop the build; the focus should primarily be on building the application.

Maven's component architecture

Maven is built around different components as shown in the following diagram:



Maven component architecture (Reference Apache Team Presentation)

The Plexus container

Plexus is an IOC container that provides component-oriented programming to build modular, reusable components that can be easily assembled and reused. Some of the features supported are as follows:

- Component lifecycles
- Component instantiation strategies
- Nested containers
- Component configuration
- Auto-wiring
- Component dependencies
- Various dependency injection techniques, including constructor injection, setter injection, and private field injection

More information on this can be found at `http://plexus.codehaus.org/`.

Wagon

Maven Wagon is a transport abstraction used in the Maven artifact and repository-handling code. Wagon defines a unified API, and it currently has the following providers:

- File
- HTTP
- HTTP lightweight
- FTP
- SSH/SCP
- WebDAV

More information on this can be found at `https://maven.apache.org/wagon/`.

Maven Doxia

Doxia is a content generation framework that provides users with powerful techniques to generate static and dynamic content. Doxia is also used in a web-based publishing context to generate static sites, in addition to being incorporated into dynamic content generation systems such as blogs, wikis, and content management systems.

For more information on Maven Doxia, refer to `https://maven.apache.org/doxia/`.

Modello

The Modello component in Maven can be used to generate the following types of artifacts at build time with reference to the data model:

- Java POJOs of the data model
- Java POJOs to XML
- XML to Java POJOs
- Xdoc documentation of the data model
- XML schema to validate that XML content matches the data model

For more information, refer to `http://maven.apache.org/maven-1.x/plugins/modello/`.

Maven SCM

This component provides a common API to perform **Source Code Management** (**SCM**) operations. The following type of SCMs are supported:

- Bazaar
- CVS
- Git
- Jazz
- Mercurial
- Perforce
- StarTeam
- Subversion
- CM energy

More information is available at http://maven.apache.org/scm/.

Maven versus Ant

Before the emergence of Maven, Ant was the most widely used build tool across Java projects. Ant emerged from the concept of creating files in C/C++ programming to a platform-independent build tool. Ant used XML files to define the build process and its corresponding dependencies.

Another Neat Tool (**Ant**) was conceived by James Duncan Davidson while preparing Sun's reference JSP/Servlet engine, Apache Tomcat. The following is a simple sample of an Ant build file (`http://ant.apache.org/manual/using.html`):

```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

<!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file
-->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
basedir="${build}"/>
  </target>
```

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

Downloading the sample code

You can download the sample code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

This example shows how to build a simple JAR file. Note how all the details corresponding to source files, class files, and JAR files have to be specified. Even the sequence of steps must be specified. This results in a complex build file and often a lot of duplicated XML.

Let's look at the simplest Maven build file, the `pom` file, which will be discussed in more detail in *Chapter 3*, *Creating and Importing Projects*.

A simple `pom` file will look as shown in the following code snippet:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packt.mvneclipse</groupId>
    <artifactId>mvneclipse</artifactId>
    <version>1.2</version>
</project>
```

This is all we need to build and package as a JAR from a Java project. Some of the differences between Ant and Maven in the preceding examples are as follows:

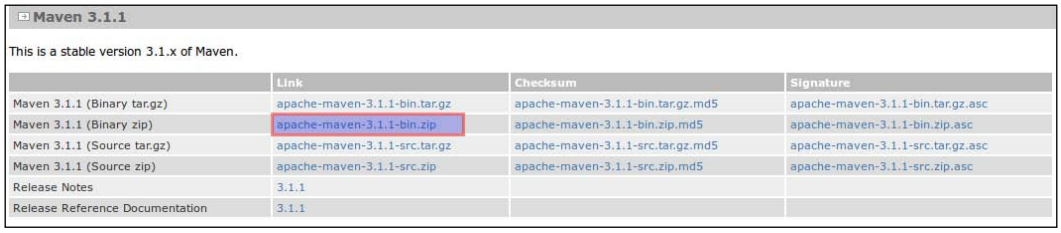
- **Convention over configuration**: Ant requires a developer to configure everything right from the source code's location to the storage of a JAR file. Maven, on the other hand, follows conventions, has a well-defined project structure, and knows where to reference source, resource files, and place the output.
- **Lifecycle**: Ant does not have a lifecycle and requires defining goals and their dependencies. Also, in Ant, the sequence of tasks needs to be specified. Maven has defined a lifecycle that consists of build phases and goals; hence, no configuration is required.

Apart from the preceding differences that can be cited from the preceding simple example, Maven is superior to Ant in the following aspects:

- **Higher level of reusability**: The build logic can be reused with Maven across different projects in Maven.
- **Less maintenance**: With a standardized structure and the reusability option, it requires less effort towards maintenance.
- **Dependency management**: One of the most superior aspects of Maven over Ant is its ability to manage the corresponding dependencies. Though, lately, Ant in combination with Apache Ivy does ease dependency management; however, Maven has other aspects that outdo this combo offering.
- **Automatic downloads**: Maven downloads the dependencies automatically; however, Ant lacks this. While Ant can use Ivy to replicate this behavior, it requires additional behavior.
- **Repository management**: Maven repositories are arbitrary and accessible locations that are designed to store the artifacts that Maven builds. They manage repositories as local versus remote (will be discussed in detail in the *Repository* section of *Chapter 3*, *Creating and Importing Projects*). Ant does not have this aspect built.

Downloading Maven

To download Maven, please visit `http://maven.apache.org/download.cgi`. Click on the latest version, `apache-maven-x.x.x-bin.zip`; at the time of writing this, the current version is `apache-maven-3.2.1-bin.zip`. Download the latest version as shown in the following screenshot:

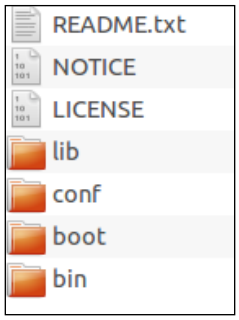


Maven 3.1.1

This is a stable version 3.1.x of Maven.

	Link	Checksum	Signature
Maven 3.1.1 (Binary tar.gz)	apache-maven-3.1.1-bin.tar.gz	apache-maven-3.1.1-bin.tar.gz.md5	apache-maven-3.1.1-bin.tar.gz.asc
Maven 3.1.1 (Binary zip)	apache-maven-3.1.1-bin.zip	apache-maven-3.1.1-bin.zip.md5	apache-maven-3.1.1-bin.zip.asc
Maven 3.1.1 (Source tar.gz)	apache-maven-3.1.1-src.tar.gz	apache-maven-3.1.1-src.tar.gz.md5	apache-maven-3.1.1-src.tar.gz.asc
Maven 3.1.1 (Source zip)	apache-maven-3.1.1-src.zip	apache-maven-3.1.1-src.zip.md5	apache-maven-3.1.1-src.zip.asc
Release Notes	3.1.1		
Release Reference Documentation	3.1.1		

Once the ZIP file is downloaded, extract the files to, let's say, `maven3`. After extraction, the contents of the `maven3` folder will have another folder named `apache-maven-3.2.1` and the contents of that folder will be as shown in the following screenshot:



Installing Maven

Before we install Maven, we need to have JDK installed. Check out the Java installation with the following command:

```
>javac -version
```

For Windows, open the command prompt, and for Linux/Mac OS, open the terminal and use the preceding command to see the version of the JDK that is installed.

If JDK is not installed, please refer to following link and install it:

`http://www.oracle.com/technetwork/java/javase/index-137561.html`

Once Java is in place, let's move towards Maven's installation.

Maven's installation is a simple two-step process:

- Setting up Maven home, that is, the `M2_HOME` variable
- Adding Maven home to the `PATH` variable

Installing Maven on Windows

The installation of Maven is just setting up Maven home in the extracted Maven folder. For ease, let's assume the `maven3` folder resides in `C:\Program Files`. Now, set Maven home with the following command in the command prompt:

```
set M2_HOME="c:\Program Files\maven3\apache-maven-3.2.1"
```

Update the `PATH` variable as follows:

```
set PATH =%PATH%;%M2_HOME%\bin
```

Alternatively, the variables can be set permanently by navigating to **Desktop | My Computer | Properties**. Visit `http://www.computerhope.com/issues/ch000549.htm` for more information.

Installing Maven on Linux and Mac OS

Let's assume the `maven3` folder resides in the `/opt` folder. As Mac OS does not have the `/opt` folder, let's create a folder `opt` in root, that is, `/opt`. Now, let's assume we have `maven3`, the extracted folder in it. Then, set the Maven home by issuing the following command via the terminal:

```
export  M2_HOME=/opt/maven3/apache-maven-3.2.1
```

Add Maven to the `PATH` variable as follows:

```
export PATH=${M2_HOME}/bin:${PATH}
```

To add it permanently, add it to the `bash` file as follows:

```
cd $HOME
vi .bash_profile
```

Add the preceding variable, that is, two lines to the file, save it, and execute the following command:

```
source  .bash_profile
```

Verifying the installation of Maven

After performing the previous steps, its time to verify the installation of Maven. To verify the installation, perform the following:

- For Windows, open the command prompt and type the following:
  ```
  mvn -version
  ```
- For Linux and Mac OS, open the terminal and type the following:
  ```
  mvn -version
  ```

It should show the corresponding version of Maven installed, as shown in the following screenshot:

```
Last login: Wed Jul  2 06:25:20 on console
macs-MacBook-Air:~ mac$ mvn -version
Apache Maven 3.2.1 (ea8b2b07643dbb1b84b6d16e1f08391b666bc1e9; 2014-02-14T23:22:52+05:45)
Maven home: /opt/maven3/apache-maven-3.2.1
Java version: 1.6.0_65, vendor: Apple Inc.
Java home: /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x", version: "10.9.1", arch: "x86_64", family: "mac"
```

Summary

Congratulations! By the end of this chapter, you have got yourselves acquainted with Maven and have installed Maven in your system. Now you are ready to take a sprint towards the journey. In the next chapter, you will learn about installing and setting up the m2eclipse plugin for Eclipse.

2
Installing m2eclipse

We set out on our journey by taking the first step in the previous chapter; here, we will take another step forward. In this chapter, we will start with the installation of an IDE, that is, Eclipse, and then get into the details of installation of Maven integration into the Eclipse plugin, that is, m2eclipse. The topics covered in this chapter are:

- Introduction to m2eclipse
- Downloading Eclipse
- Installing and launching Eclipse
- Methods to install m2eclipse
- Setting up Maven for use

Introduction to m2eclipse

m2eclipse is a plugin that provides Maven integration with Eclipse. It intends to bridge the gap between Eclipse and Maven, help to create projects using simple intuitive interfaces from Maven Archetypes, and launch and manage the projects build using a simple editor. It makes the use of Maven right from the IDE so much easier. Some of the features provided by m2eclipse are as follows:

- Creating and importing Maven projects
- Launching the Maven build from within Eclipse
- Dependency management for the Eclipse build path
- Automatic dependency downloads and updates
- Materializing a project
- Browsing and searching remote Maven repositories
- Providing support for multimodule Maven projects

Apart from the preceding features, in conjunction with different m2e connectors and the Mylyn plugin, it provides the ability to communicate with code versioning repositories and task-based interfaces.

m2eclipse has been around since 2006 and is credited to Eugene Kuleshov. It was developed under the Codehaus community for 2 years before it was moved to the Eclipse Foundation in 2008.

Downloading Eclipse

If you have Eclipse installed, you can skip this and the next section and right away proceed to the *Installing m2eclipse* section.

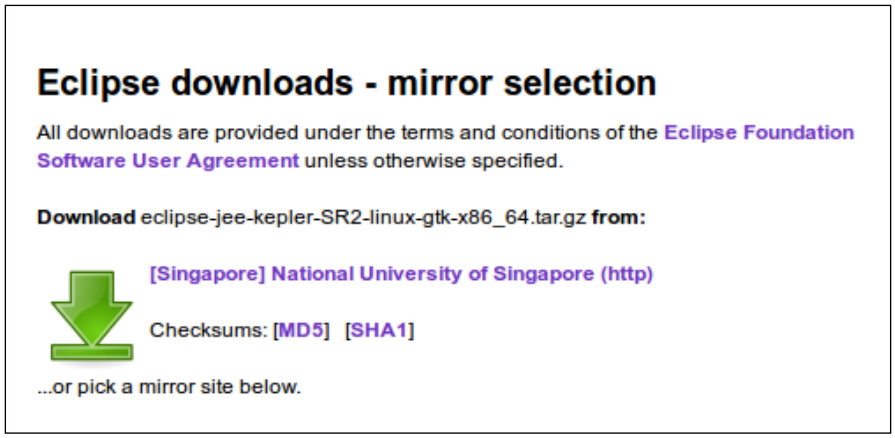
To download Eclipse, please visit the following URL:

```
https://www.eclipse.org/downloads/
```

The next screenshot can be visualized. At the time of writing this book, the latest version of Eclipse is Eclipse Kepler 4.3.2 SR2, and we will be downloading this and using it for the rest of the book.

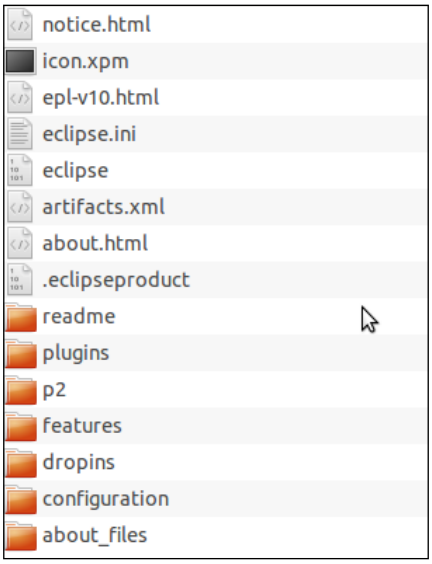
Choose an appropriate OS from the dropdown and download the **Eclipse IDE for Java Developers** package for the corresponding architecture, that is, 32 or 64 bit (which is shown in the following screenshot). Choose **32 bit** for 32-bit Java or **64 bit** for 64-bit Java installed in the system.

In the next step, choose the appropriate mirror close to your location and the download will begin. The mirror screen may look like the following screenshot:

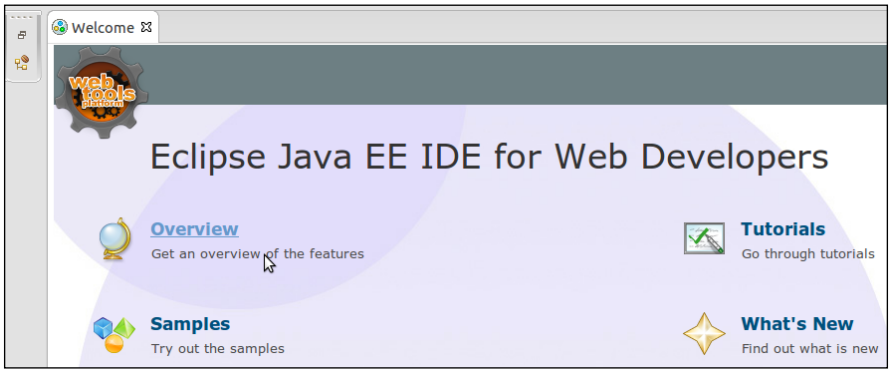


Installing and launching Eclipse

Go to the location of the downloaded file, as shown in the preceding screenshot, and extract it to a desired location of your choice. The extraction will result in a folder named `eclipse`. The contents of the `eclipse` folder are shown in the following screenshot:



We can see there is an application or executable file named `eclipse`, which on double-clicking, launches the Eclipse IDE. When Eclipse is launched, it will prompt you for a workspace location. Provide an appropriate location where the projects are to be stored and click on **OK**. Now, we are right in the Eclipse space and ready for action. You see something similar to the following screenshot:



Methods to install m2eclipse

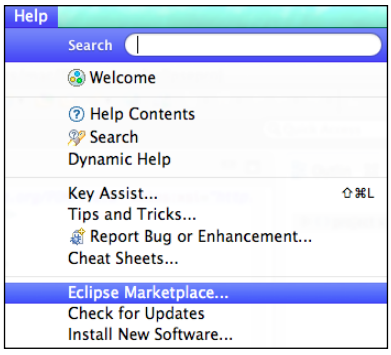
Installing m2eclipse is fairly simple. Primarily, there are two ways to install the m2eclipse plugin in Eclipse:

- **Using Eclipse Marketplace**: Use Eclipse Marketplace to find and install the plugin
- **Using update site**: Add the m2eclipse update site and install

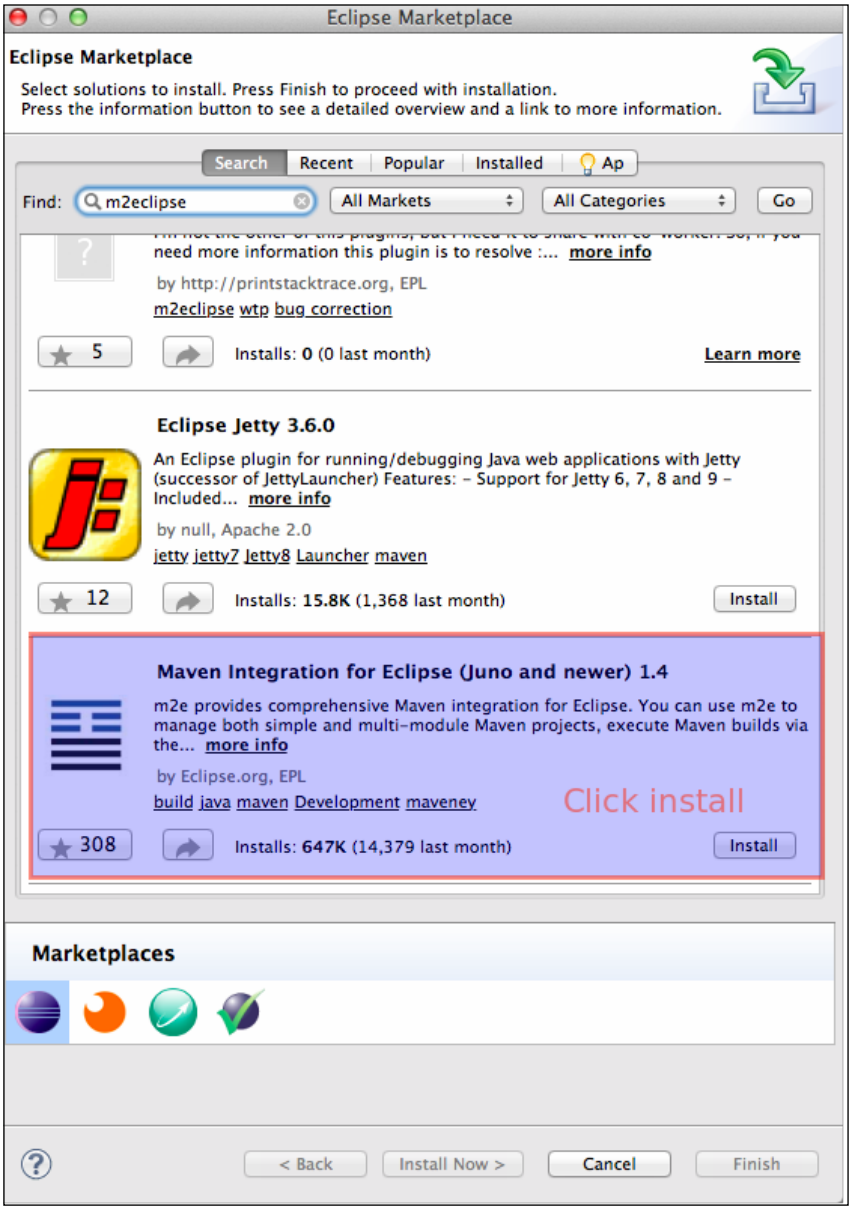
Using Eclipse Marketplace

The installation of m2eclipse using Eclipse Marketplace involves the following steps:

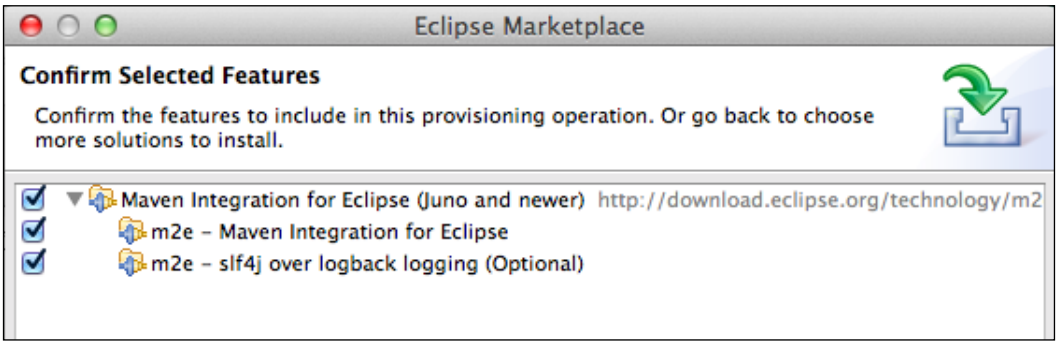
1. Navigate to **Help** | **Eclipse Marketplace** as shown in the following screenshot:



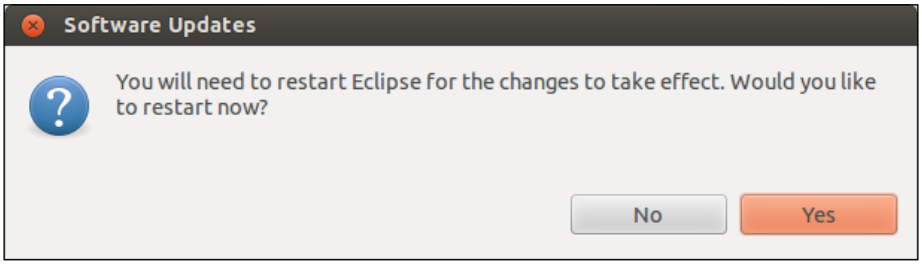
2. Then, search for `m2eclipse` in the search box, and click on the **Install** button for Maven integration for the Eclipse package, as shown in the following screenshot:



3. On the next window, confirm the package to be installed as follows:



4. Accept the terms and conditions and click on **Finish**. After the installation is done, the following prompt appears:

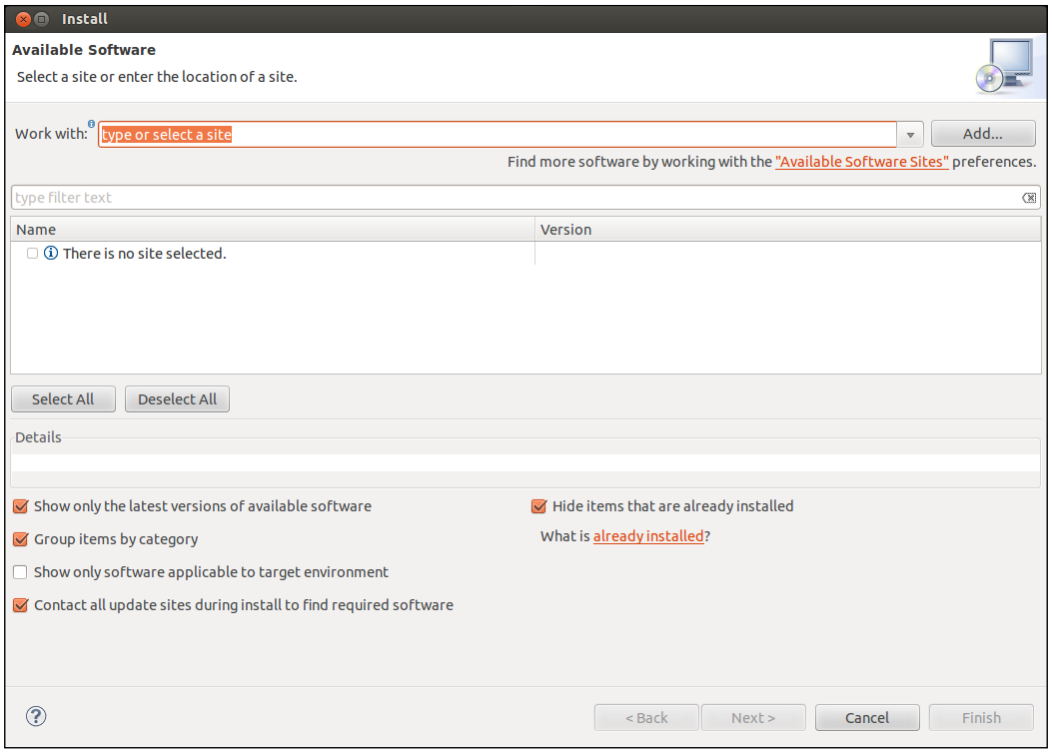


5. Click on **Yes** to restart Eclipse and to have the changes reflected.
6. For Mac users, choose the **Restart Now** option, and for other OSes, choose **Yes**.

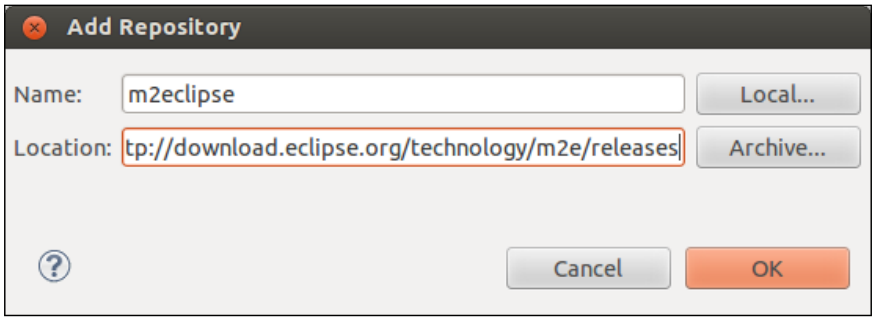
Using Update Site

The installation of m2eclipse using update site involves the following steps:

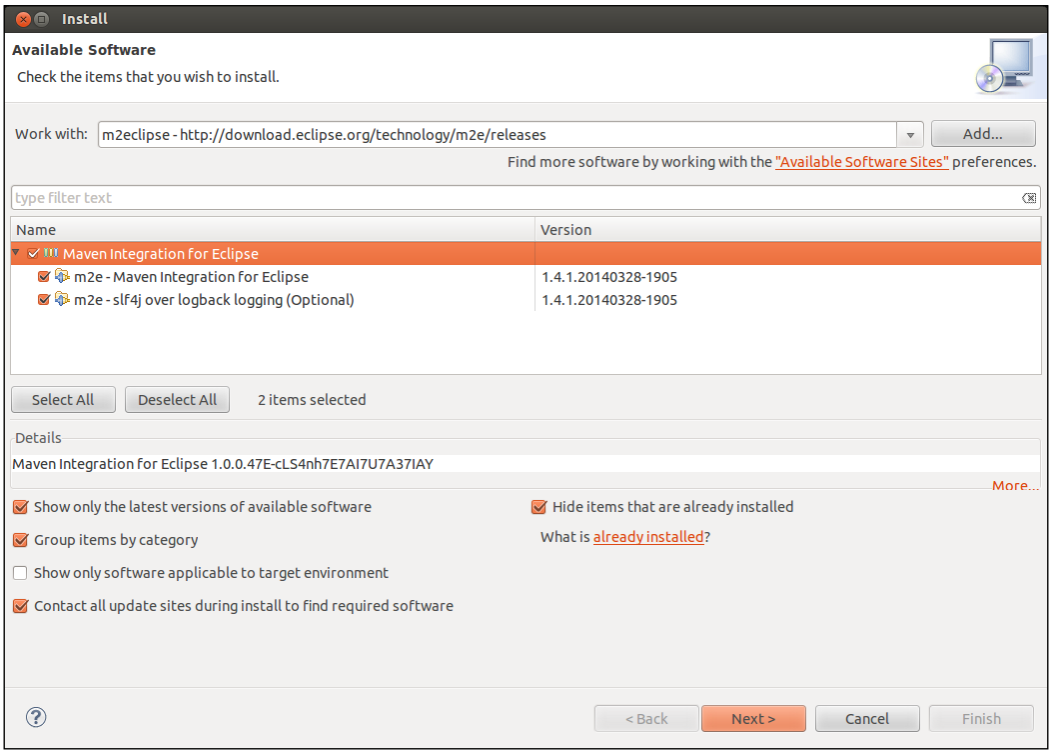
1. Navigate to **Help | Install New Software** and your screen will look similar to the following screenshot:



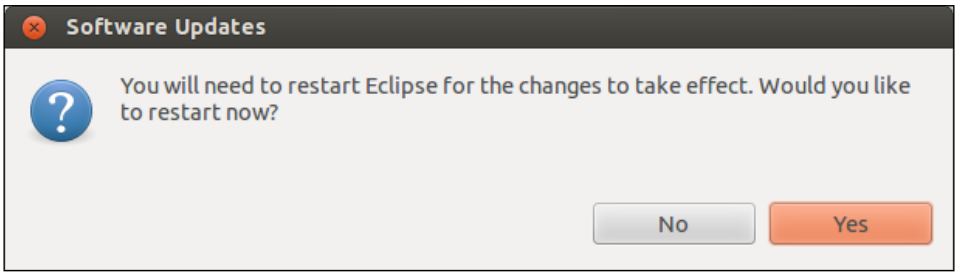
2. Click on the **Add...** button. Add the `http://download.eclipse.org/technology/m2e/releases` site as the m2eclipse update site, as shown in the following screenshot and click on **OK**:



3. Choose the packages as shown in the following screenshot:



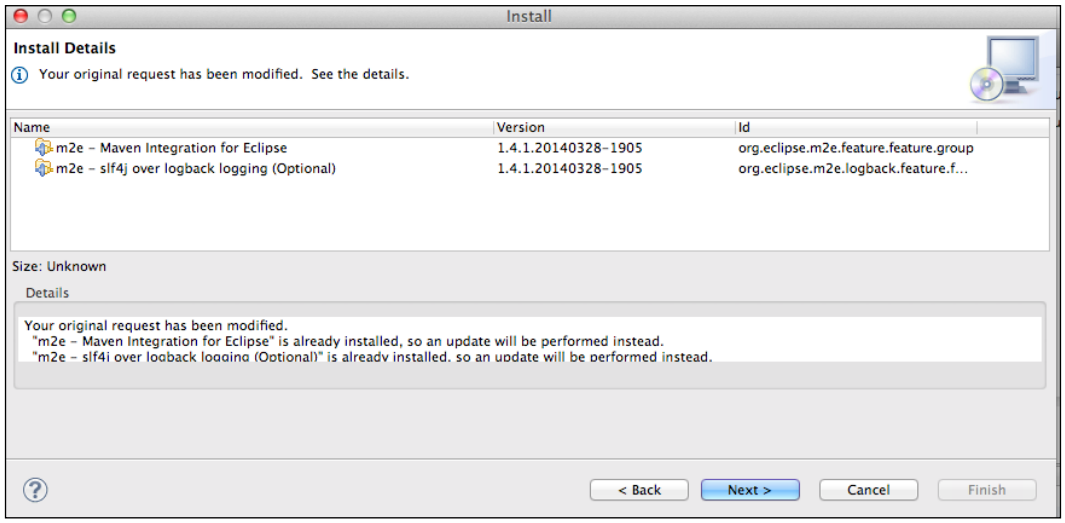
4. Click on **Next**, agree to the terms, and finally click on **Finish** to start installing. Once the installation is done, the following prompt appears:



5. Click on **Yes** to restart Eclipse and to have the changes reflected. For Mac users, choose the **Restart Now** option and for users with other OSes, choose **Yes**.

Eclipse Kepler 4.3.2 SR2 has m2eclipse installed and hence the preceding step of installation would update the plugin to the latest one. Regardless of any of the preceding methods of installation, m2eclipse that comes packaged with Eclipse Kepler is still going to be updated.

So, midway, you will see something similar to the following screen:

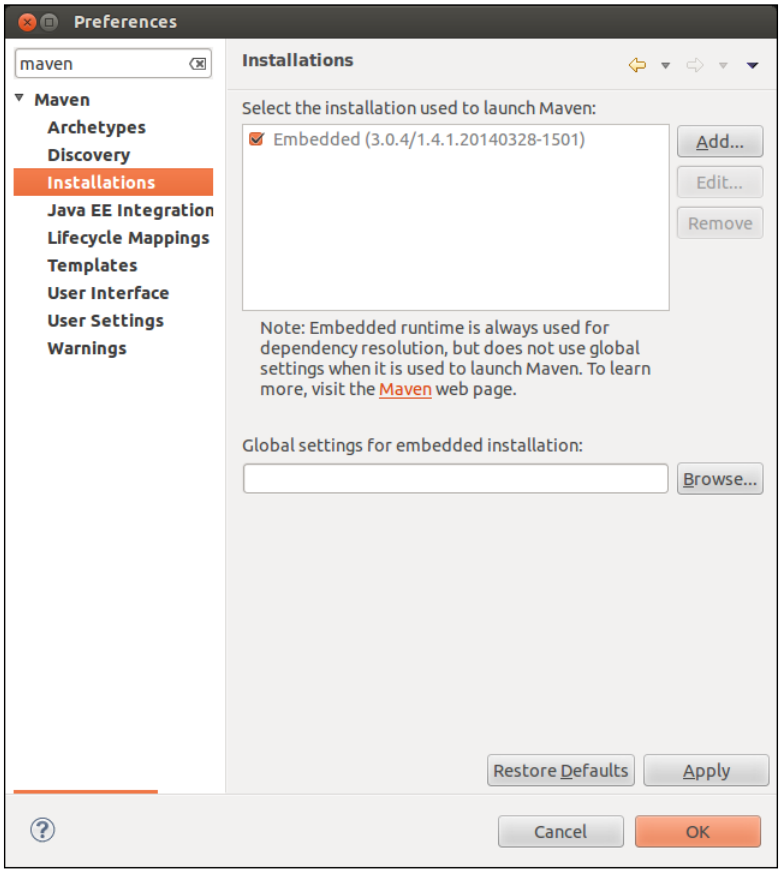


6. Click on **Next** and accept the terms, click on **Finish** to start the installation, and you will have to restart to have the changes reflected.

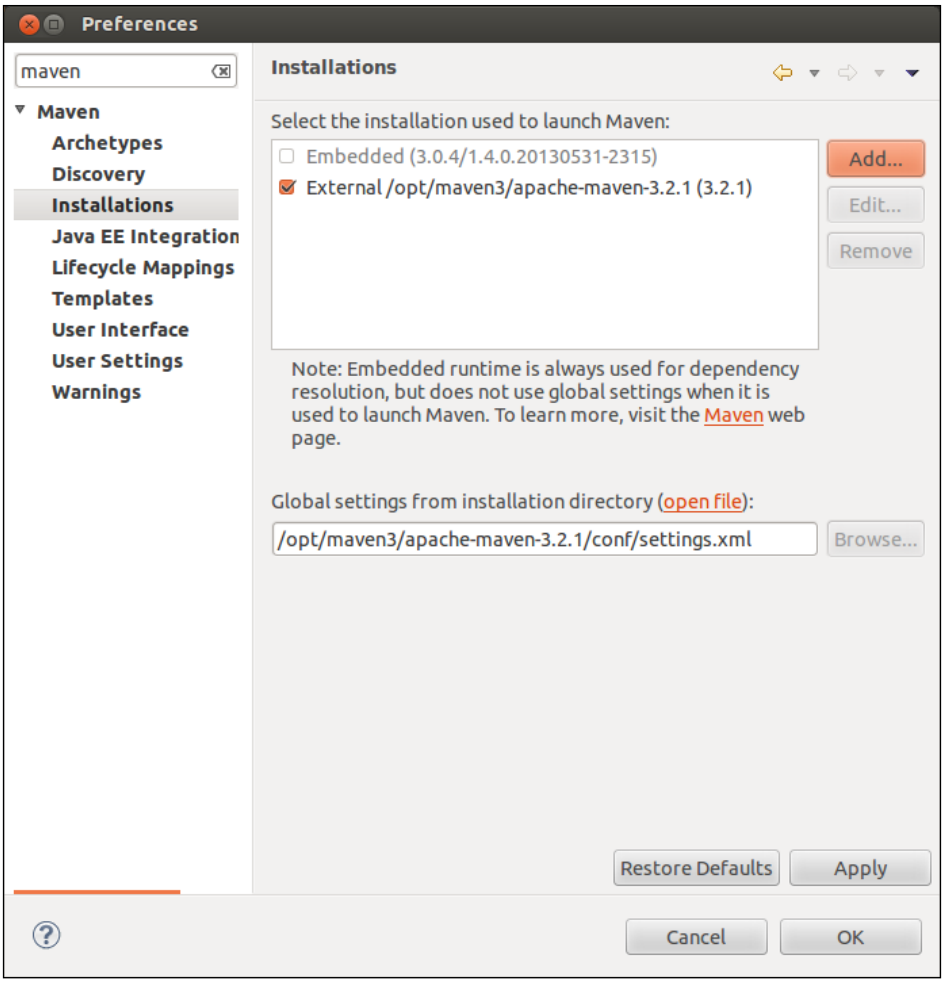
Setting up Maven for use

m2eclipse comes with an embedded Maven component in it, so the external Maven installation discussed in *Chapter 1*, *Apache Maven – Introduction and Installation*, is optional. However, to use the latest version of Maven, we are required to install Maven externally, as discussed in the previous chapter. We also need to make sure our m2eclipse plugin uses it. Also, the use of continuous integration servers nowadays requires us to have a common Maven version across servers, thus making use of the externally installed Maven.

To make m2eclipse use the external Maven version, navigate to **Window** | **Preference** in Eclipse and the **Preference** window appears. Search for `maven` in the search box in the left pane and click on **Installations** as shown in the following screenshot:



Click on the **Add...** button and select the location of the Maven directory. From the previous chapter, our location was `/opt/maven3/apache-maven-3.2.1`. Check the corresponding external Maven checkbox, as shown in the following screenshot, and click on **Apply** followed by **OK**:



Now, m2eclipse will make use of this Maven.

Summary

By the end of this chapter, you have learned about installing Eclipse and m2eclipse as well as setting up m2eclipse to use the externally installed Maven. In the next chapter, we will look at the important concepts of Maven and you will learn to create and import Maven projects and familiarize yourself with the structure of Maven projects.

3
Creating and Importing Projects

Let's proceed on our journey. In this chapter, we will start with the Maven project structure followed by the build architecture, then we will cover some essential concepts, and finally learn how to create simple Maven projects. The chapter is divided into the following sections:

- The Maven project structure
- **POM (Project Object Model)**
    - Maven coordinates
    - POM relationships
    - Simple POM
    - Super POM
- The Maven project build architecture
- Other essential concepts
    - Repository
    - Project dependencies
    - Plugins and goals
    - Site generation and reporting
- Creating a Maven project
    - Using an archetype
    - Using no archetypes
    - Checking out a Maven project
- Importing Maven projects

The Maven project structure

Maven, as stated in earlier chapters, follows convention over configuration. This makes us believe that there is a standard layout of the Maven project structure. Before we get into creating and playing with Maven projects, let's first understand the basic common layout of Maven projects, as follows:

Folder/Files	Description
`src/main/java`	This contains an application's Java source files
`src/main/resources`	This contains files of an application's resources such as images, sounds, templates, and so on
`src/main/filters`	This contains the resource's filter files
`src/main/config`	This contains the configuration files of the application
`src/main/scripts`	This has files of application-specific scripts
`src/main/webapp`	This has sources files for web applications
`src/test/java`	This contains unit test files of Java
`src/test/resources`	This has unit testing-specific resources used in an application
`src/filters`	This has files of the test-specific filter for resources
`src/it`	This has integration tests files (primarily for plugins)
`src/assembly`	This contains files of the assembly descriptors
`src/site`	This contains site artifacts
`LICENSE.txt`	This denotes the projects license
`NOTICE.txt`	This includes the notice and attributions that the project depends on
`README.txt`	This denotes the project's readme information
`target`	This houses all the output of the build
`pom.xml`	This is the project's `pom` file (which will be discussed in detail in the forthcoming sections)

Though the previously mentioned layout is the standard recommended convention, this can always be overridden in the project descriptor file (`pom` file).

POM (Project Object Model)

POM stands for **Project Object Model**. It is primarily an XML representation of a project in a file named `pom.xml`. POM is the identity of a Maven project and without it, the project has no existence. It is analogous to a **Make** file or a `build.xml` file of **Ant**.

A project in a broad sense should contain more than just mere code files and should act as a one-stop shop for all the things concerning it. Maven fulfills this need using the `pom` file. POM tends to answer questions such as: Where is the source code? Where are the resources? How is the packaging done? Where are the unit tests? Where are the artifacts? What is the build environment like? Who are the actors of the project? and so on.

In a nutshell, the contents of POM fall under the following four categories:

- **Project information**: This provides general information of the project such as the project name, URL, organization, list of developers and contributors, license, and so on.
- **POM relationships**: In rare cases, a project can be a single entity and does not depend on other projects. This section provides information about its dependency, inheritance from the parent project, its sub modules, and so on.
- **Build settings**: These settings provide information about the build configuration of Maven. Usually, behavior customization such as the location of the source, tests, report generation, build plugins, and so on is done.
- **Build environment**: This specifies and activates the build settings for different environments. It also uses profiles to differentiate between development, testing, and production environments.

A POM file with all the categories discussed is shown as follows:

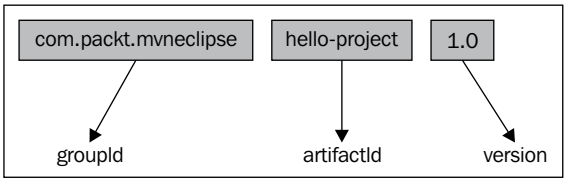
```
<project>
  <!-- The Basics Project Information-->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>

  <!-- Build Settings -->
  <build>...</build>
  <reporting>...</reporting>
  <properties>...</properties>
  <packaging>...</packaging>

  <!-- More Project Information -->
  <name>...</name>
```

```
    <description>...</description>
    <url>...</url>
    <inceptionYear>...</inceptionYear>
    <licenses>...</licenses>
    <organization>...</organization>
    <developers>...</developers>
    <contributors>...</contributors>

     <!-- POM Relationships -->
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
    <parent>...</parent>

    <dependencyManagement>...</dependencyManagement>
    <dependencies>...</dependencies>

    <modules>...</modules>

  <!-- Environment Settings -->
    <issueManagement>...</issueManagement>
    <ciManagement>...</ciManagement>
    <mailingLists>...</mailingLists>
    <scm>...</scm>
    <prerequisites>...</prerequisites>
    <repositories>...</repositories>
    <pluginRepositories>...</pluginRepositories>
    <distributionManagement>...</distributionManagement>
    <profiles>...</profiles>
  </project>
```

Maven coordinates

Maven coordinates define a set of identifiers that can be used to uniquely identify a project, a dependency, or a plugin in a Maven POM. Analogous to algebra where a point is identified by its coordinate in space, the Maven coordinates mark a specific place in a repository, acting like a coordinate system for Maven projects. The Maven coordinates' constituents are as follows:

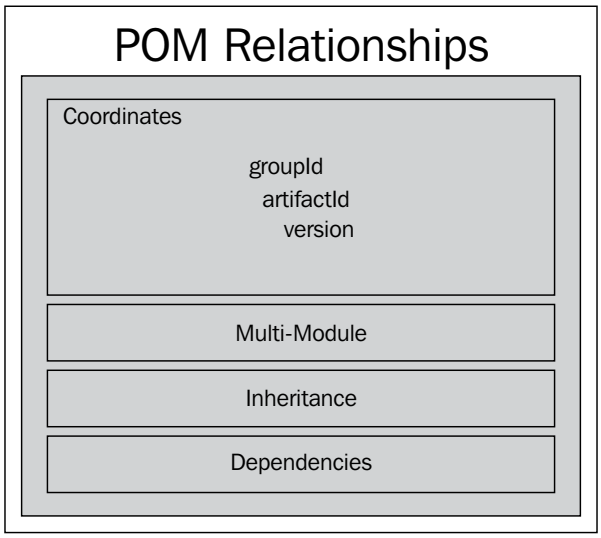
- **groupId**: This represents a group, company, team, organization, or project. A general convention for a group ID is it begins with a reverse domain name of the organization that creates the project. However, it may not necessarily use the dot notation as it does in the `junit` project. The group forms the basis for storage in the repository and acts much like a Java packaging structure does in OS. The corresponding dots are replaced with OS-specific directory separators (such as `/` in Unix), which forms the relative directory structure from the base repository. For example, if `groupId` is `com.packt.mvneclipse`, it lives in the `$M2_REPO/com/packt/mvneclipse` directory.
- **artifactId**: This is a unique identifier under `groupId` that represents a single project/the project known by. Along with the `groupId` coordinate, the `artifactId` coordinate fully defines the artifact's living quarters within the repository. For example, continuing with the preceding example, the artifact ID with `hello-project` resides at the `$M2_REPO/com/packt/mvneclipse/hello-project` path.
- **project version**: This denotes a specific release of a project. It is also used within an artifact's repository to separate versions from each other. For example, `hello-project` with version 1.0 resides in the `$M2_REPO/com/packt/mvneclipse/hello-project/1.0/` directory.
- **packaging**: This describes the packaged output produced by a project. If no packaging is declared, Maven assumes the artifact is the default `jar` file. The core packaging values available in Maven are: `pom`, `jar`, `maven-plugin`, `ejb`, `war`, `ear`, `rar`, and `par`. The following figure illustrates an example of Maven coordinates:



As the local repository, `$M2_REPO` signifies the `%USER_HOME% /.m2` directory in the user's machine.

POM relationships

POM relationships identify the relationship they possess with respect to other modules, projects, and other POMs. This relationship could be in the form of dependencies, multimodule projects, parent-child also known as inheritance, and aggregation. The elements of POM relationships are represented graphically as shown in the following figure:



Similarly, the elements of POM relationships in the XML file can be specified as shown in the following code:

```
  <!-- POM Relationships -->
<groupId>...</groupId>
<artifactId>...</artifactId>
<version>...</version>
<parent>...</parent>
<dependencyManagement>...</dependencyManagement>
<dependencies>...</dependencies>
<modules>...</modules>
```

A simple POM

The most basic POM consists of just the Maven coordinates and is sufficient to build and generate a `jar` file for the project. A simple POM file may look like the following code:

```
<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.packt.mvneclipse</groupId>
  <artifactId>hello-project</artifactId>
  <version>1.0</version>

</project>
```

The following points will explain these elements:

- The `modelVersion` value is `4.0.0`. Maven supports this version of POM model.
- There is a single POM file for every project.
- All POM files require the `project` element and three mandatory fields: `groupId`, `artifactId`, and `version`.
- The root element of `pom.xml` is `project`, and it has three major subnodes.

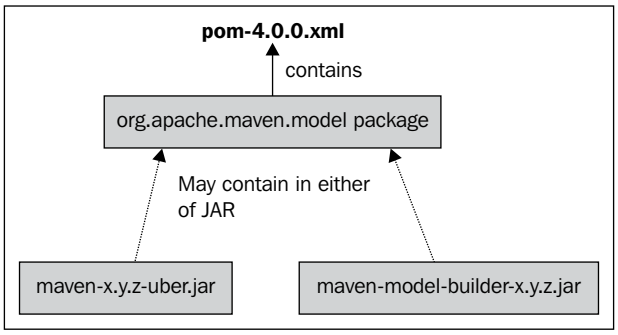
A simple POM (as shown in the previous code snippet) is hardly enough in real-world projects.

A super POM

Like Java, where every object inherits from `java.lang.Object`, every POM inherits from a base POM known as **Super POM**. Implicitly, every POM inherits the default value from the base POM. It eases the developer's effort toward minimal configuration in his/her `pom.xml` file. However, default values can be overridden easily when they are specified in the projects' corresponding `pom` file. The default configuration of the super POM can be made available by issuing the following command inside the respective project:

```
mvn help:effective-pom
```

The super POM is a part of the Maven installation and can be found in the `maven-x.y.z-uber.jar` or `maven-model-builder-x.y.z.jar` file at `$M2_HOME/lib`, where `x.y.z` denotes the version. In the corresponding JAR file, there is a file named `pom-4.0.0.xml` under the `org.apache.maven.model` package.



The default configuration of the super POM inherited in a sample project is given as follows; for the sake of brevity, only some important aspects are shown:

```
<!--General project Information -->
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.mvneclipse</groupId>
  <artifactId>hello-project</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>hello-project</name>
  <url>http://maven.apache.org</url>
  <properties>1
  <project.build.sourceEncoding>UTF8</project.build.sourceEncoding>
</properties>

<repositories>
  <repository>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Maven Repository Switchboard</name>
    <url>http://repo1.maven.org/maven2</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <releases>
      <updatePolicy>never</updatePolicy>
```

```
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
    </pluginRepository>
  </pluginRepositories>

  <!-- Build source directory and details>
    <build>
  …
    <sourceDirectory> ...</sourceDirectory>
    <scriptSourceDirectory>..</scriptSourceDirectory>
    <testOutputDirectory>..</testOutputDirectory>
    <outputDirectory>...<outputDirectory>
  …


  <finalName>hello-project-0.0.1-SNAPSHOT</finalName>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-antrun-plugin</artifactId>
          <version>1.3</version>
        </plugin>
        <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <version>2.2-beta-5</version>
        </plugin>
          <plugin>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>2.1</version>
          </plugin>
          <plugin>
            <artifactId>maven-release-plugin</artifactId>
            <version>2.0</version>
          </plugin>
        </plugins>
      </pluginManagement>
      <plugins>
```

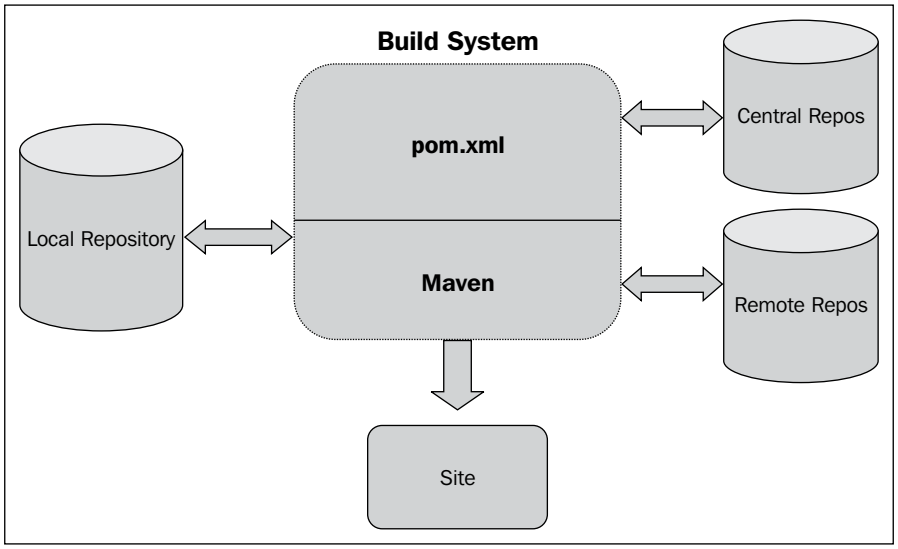
```
<!-- Plugins, phases and goals -->
    <plugin>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.4.1</version>
      <executions>
        <execution>
          <id>default-clean</id>
          <phase>clean</phase>
          <goals>
            <goal>clean</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.7.2</version>
      <executions>
        <execution>
          <id>default-test</id>
          <phase>test</phase>
          <goals>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <executions>
        <execution>
          <id>default-testCompile</id>
          <phase>test-compile</phase>
          <goals>
            <goal>testCompile</goal>
          </goals>
        </execution>
        <execution>
          <id>default-compile</id>
          <phase>compile</phase>
          <goals>
            <goal>compile</goal>
          </goals>
```

```
    </execution>
  </executions>
</plugin>
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.3.1</version>
  <executions>
    <execution>
      <id>default-jar</id>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
<plugin>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>2.5</version>
  <executions>
    <execution>
      <id>default-deploy</id>
      <phase>deploy</phase>
      <goals>
        <goal>deploy</goal>
      </goals>
    </execution>
  </executions>
</plugin>
<plugin>
  <artifactId>maven-site-plugin</artifactId>
  <version>2.0.1</version>
  <executions>
    <execution>
      <id>default-site</id>
      <phase>site</phase>
      <goals>
        <goal>site</goal>
      </goals>
      <configuration>
</project>
```

The Maven project build architecture

The following figure shows the common build architecture for Maven projects. Essentially, every Maven project contains a POM file that defines every aspect of the project essentials. Maven uses the POM details to decide upon different actions and artifact generation. The dependencies specified are first searched for in the local repository and then in the central repository. There is also a notion that the remote repository is searched if it is specified in the POM. We will talk about repositories in the next section. In addition, POM defines details to be included during site generation.

Have a look at the following diagram:



Other essential concepts

The other essential concepts of Maven are discussed in the following sections.

Repository

Maven repositories are accessible locations designed to store the artifacts that Maven builds produce. To be more precise, a repository is a location to store a project's artifacts that is designed to match the Maven coordinates.

A Maven repository can be one of the following types:

- Local
- Central
- Remote

The local repository

A local repository is one that resides in the same machine where a Maven build runs. It is a `.m2` folder located in the `$USER_HOME` directory of the user's machine. It is created when the `mvn` command is run for the very first time. However, to override the default location, open the `settings.xml` file if it exists; else, create one in the `$M2_HOME\conf` (for windows: `%M2_HOME%\conf`) folder and respective location as in the following code:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>/opt/m2repos</localRepository>
</settings>
```

When we run the Maven command, Maven will download dependencies to a custom path.

The central repository

The central repository is the repository provided by the Maven community. It contains a large repository of commonly used libraries. This repository comes into play when Maven does not find libraries in the local repository. The central repository can be found at: `http://search.maven.org/#browse`.

The remote repository

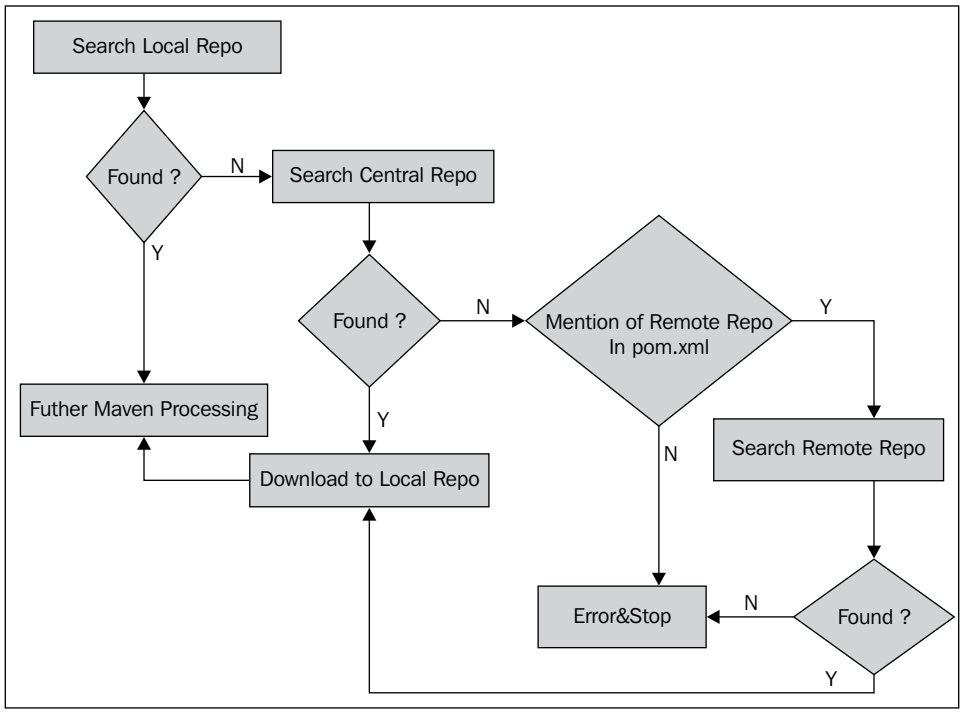
Enterprises usually maintain their own repositories for the libraries that are being used for the project. These differ from the local repository; a repository is maintained on a separate server, different from the developer's machine and is accessible within the organization. Also, sometimes, there are cases where the availability of the libraries in central repositories is not certain, thus giving rise to the need for a remote repository.

For example, the following POM file mentions the remote repositories, where the dependency is not available in the central repository:

```
<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.packt.mvneclipse</groupId>
  <artifactId>hello-project</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>com.packt.commons</groupId>
      <artifactId>utility-lib</artifactId>
      <version>1.0.0</version>
    </dependency>
  <dependencies>
  <repositories>
    <repository>
       <id>packt.ser1</id>
       <url>http://download.packt.net/maven2/1</url>
    </repository>
    <repository>
      <id>packt.ser2</id>
      <url>http://download.packt.net/maven2/2</url>
    </repository>
  </repositories>
</project>
```

Search sequence in repositories

The following figure illustrates the sequence in which the search operation is carried out in the repositories on execution of the Maven build:



Maven follows the ensuing sequence to search dependent libraries in repositories, and the sequence is explained as follows:

1. In step 1, Maven searches for dependencies in the local repository; if found, it proceeds further, else it goes to the central repository.
2. In step 2, the search continues in the central repository; if found, it proceeds to download the dependent libraries to the local repository and continues the processing. If the search fails in the central repository and if there is a mention of a remote repository in the POM file, it continues with step 3 or else throws an error and stops.
3. In step 3, the search continues in the remote repositories. If found, it proceeds to download the dependent libraries to the local repository and continues processing. If search encounters a failure, it throws an error and stops at that juncture.

Project dependencies

The powerful feature of Maven is its dependency management for any project. Dependencies may be external libraries or internal (in-house) libraries/project. Dependencies in POM can be stated under the following tags with the following attributes as shown:

```
<dependencies>
  <dependency>
    <groupId>org.testng </groupId>
    <artifactId>testng</artifactId>
    <version>6.1.1</version>
    <type>jar</type>
    <scope>test</scope>
    <optional>true</optional>
  </dependency>
...
  </dependencies>
```

The attributes used in the preceding code snippet are as follows:

- `groupId`, `artifactId`, and `version`: These are the Maven coordinates for dependency.
- `type`: This is a dependency packaging type. The default type is JAR. We have already discussed this in an earlier section.
- `scope`: This provides a mechanism of control over the inclusion of dependencies in the class path and with an application. We will talk about this scope in the next section.
- `optional`: This indicates the dependency as optional when the project is a dependency. To put this in simple terms, consider that project A has the `optional` dependency, which means it needs this library at build time. Now, project B has this project A that is dependency defined, so this implies B may not need A's dependency for its build and is a part of transitive dependencies.

Dependency scopes

Dependency scopes control the availability of dependencies in a classpath and are packaged along with an application. There are six dependency scopes, which are described in detail as follows:

- `Compile`: This is the default scope if not specified. Dependencies with this scope are available in all classpaths and are packaged.
- `Provided`: Similar to the `compile` scope, however, this indicates JDK or the container to provide them. It is available only in compilation and test classpaths and is not transitive.

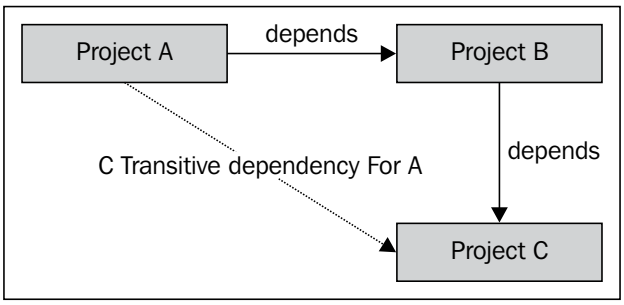
- `Runtime`: This scope indicates that the dependency is not required for compilation but is available for execution. For example, a JDBC driver is required only at runtime, however the JDBC API is required during compile time.
- `Test`: This scope indicates that the dependency is not required for normal use of the application, and it is only available for the test compilation and execution phases.
- `System`: This is similar to the `provided` scope but the explicit path to JARs on the local filesystem is mentioned. The path must be absolute such as `$JAVA_HOME/lib`. Maven will not check the repositories; instead it will check the existence of the file.

Transitive dependencies

Project A depends on project B and project B depends on C—now C is a transitive dependency for A. Maven's strength lies in the fact that it can handle transitive dependencies and hide the chain of dependencies under the hood from a developer's knowledge. As a developer, the direct dependency of the project is defined, and all other dependencies' chain nuisance is dealt by Maven with effective version conflict management. Scope limits the transitivity of a dependency as discussed in the preceding section by allowing the inclusion of dependencies appropriate for the current stage of the build.

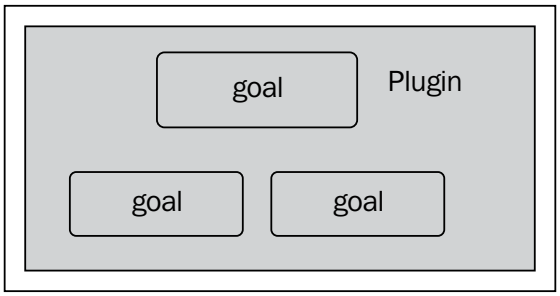
For more information, please visit `http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html`.

Transitive dependency is illustrated in the following figure:



Plugins and goals

Maven, essentially, is a plugin framework where every action is the result of some plugin. Each plugin consists of goals (also called Mojos) that define the action to be taken. To put it in simple words, a *goal* is a unit of work. For example, a `compiler` plugin has `compile` as the goal that compiles the source of the project. An illustration is as follows:



A plugin with set of goals can be executed using the following command:

```
mvn [pluginID:goalID]
```

Typically, the following are the types of plugins:

Type	Description
Build plugins	These are executed during the build and are specified in the `<build> <build/>` element in the `pom` file.
Reporting Plugins	These are executed during site generation and are configured in the `<reporting> <reporting/>` element in the `pom` file.
Core plugins	These plugins correspond to the default core phases.
Packaging types/tools	These relate to the respective artifact types for packaging.

The following table consists of some of the common plugins:

Plugin	Description
`compiler`	This is used to compile the source code.
`jar`	This builds the `jar` file from the project.
`war`	This builds the `war` file from the project.
`install`	This installs the build artifact into the local repository.

Plugin	Description
`site`	This generates the site for the current project.
`surefire`	This runs unit tests and generates reports.
`clean`	This cleans up the target after the build.
`javadoc`	This generates a Javadoc for the project.
`pdf`	This generates the PDF version of the project documentation.

For more plugins, navigate to `http://maven.apache.org/plugins/`.

Site generation and reporting

Seldom are projects a single developer's asset. A project contains stakeholders, and collaboration among them is essential. Often, a lack of effective documentation has paralyzed the project, its maintenance, and its usage. Maven with its `site` plugin has eased this process of having effective project documentation by generating a site and reports related to project. A site can be generated using the following command:

```
mvn site
```

The site is generated at the `target/site` directory. Maven uses the Doxia component (discussed in the *Maven Component Architecture* section of *Chapter 1*, *Apache Maven – Introduction and Installation*) to generate documentation. The site also contains all the configured reports such as the unit test coverage, PMD report, and others. We will cover site and report generation in more detail in the *Generating site documentation* section of *Chapter 5*, *Spicing Up a Maven Project*.

Creating a Maven project

m2eclipse makes the creation of Maven projects simple. Maven projects can be created in the following two ways:

- Using an archetype
- Without using an archetype

Now, we will discuss how to go about creating projects using these methods.

Using an archetype

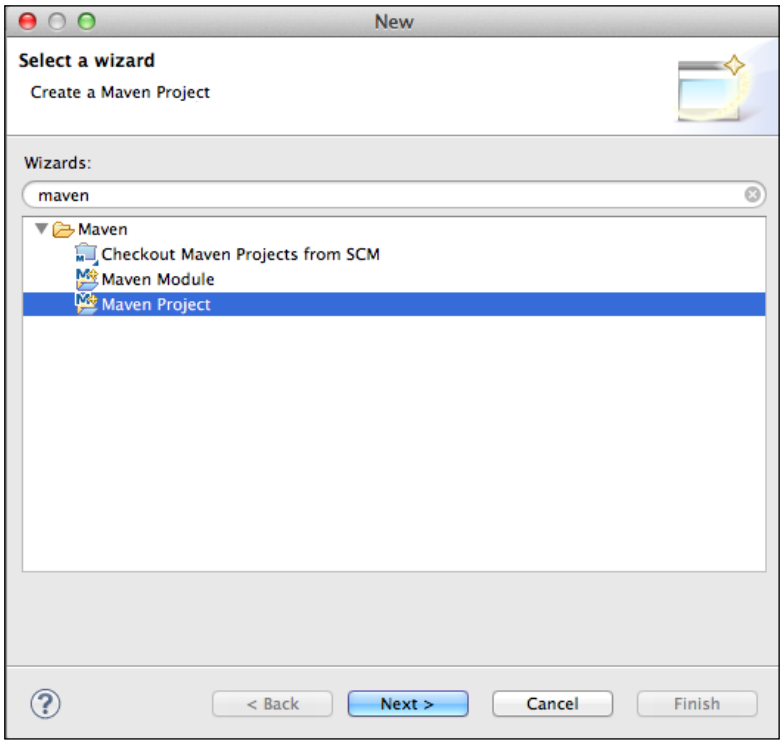
An archetype is a plugin that allows a user to create Maven projects using a defined template known as archetype. There are different archetypes for different types of projects.

Archetypes are primarily available to create the following:

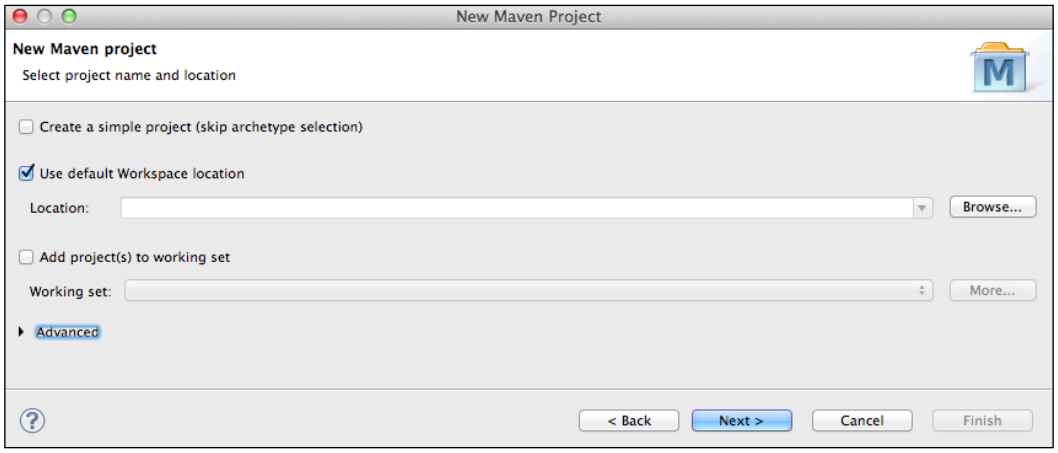
- Maven plugins
- Simple web applications
- Simple projects

We will now see how to create a simple Hello World! project using an archetype:

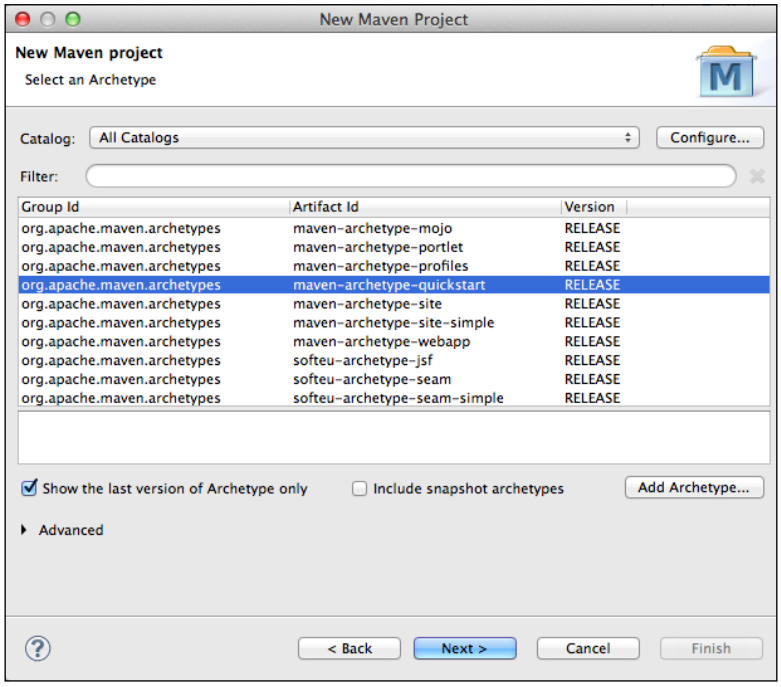
1. Navigate to **File | New** and click on **Other**. The project wizard appears and expands the `Maven` folder. Select **Maven Project** as shown in the following screenshot and click on **Next**:



2. The **New Maven Project** wizard appears. Use the default workspace and click on **Next** as shown in the following screenshot:

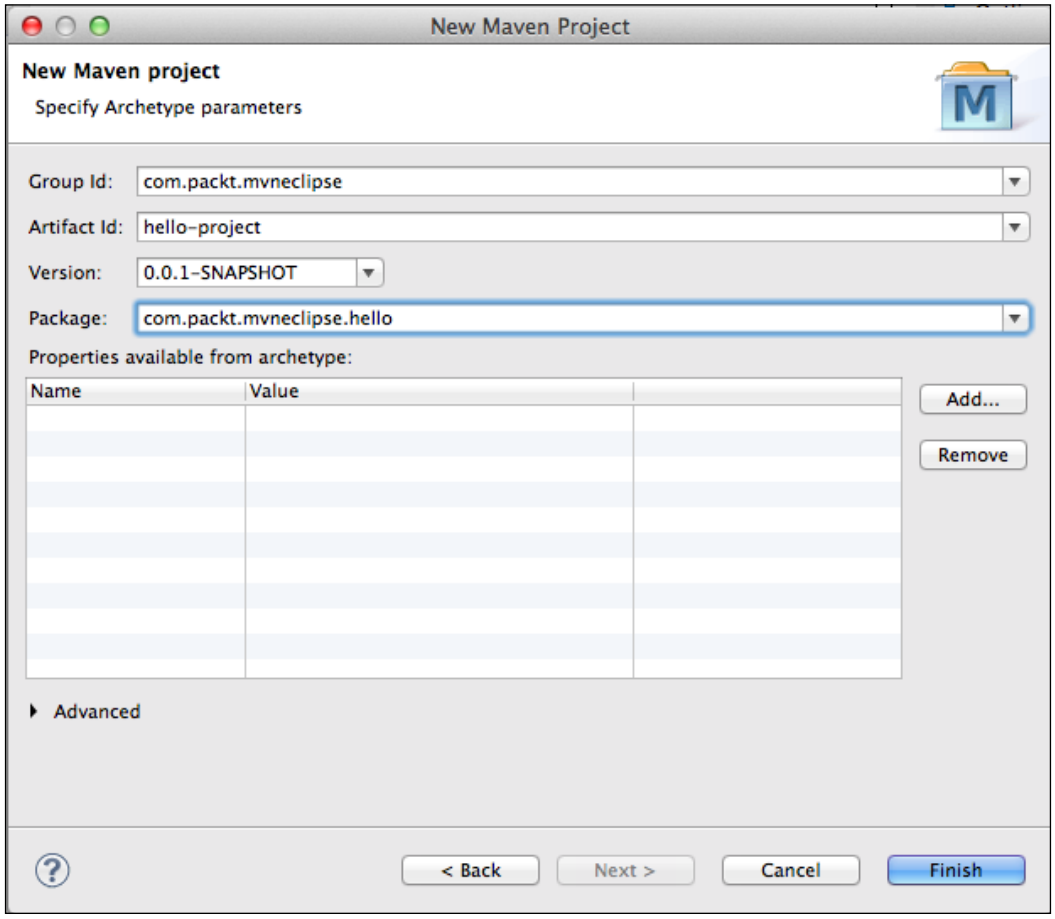


3. The **Select an Archetype** wizard appears. This shows a list of archetypes available in the Maven repository. New archetypes can be added using the **Add Archetypes** button. For our case here, let's choose **maven-archetype-quickstart** as shown in the following screenshot and click on **Next**:

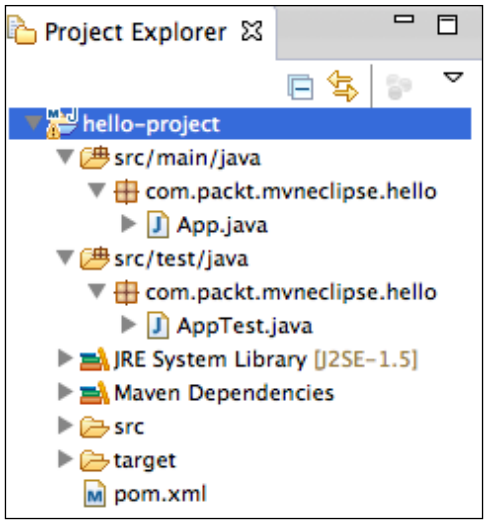


4. A wizard to specify the Maven coordinates appears. Please fill in the details given in the following table in the screenshot that follows the table and click on **Finish**:

Field	Value
Group Id	`com.packt.mvneclipse`
Artifact Id	`hello-project`
Version	Default – **0.0.1-SNAPSHOT**
Package	`com.packt.mvneclipse.hello`



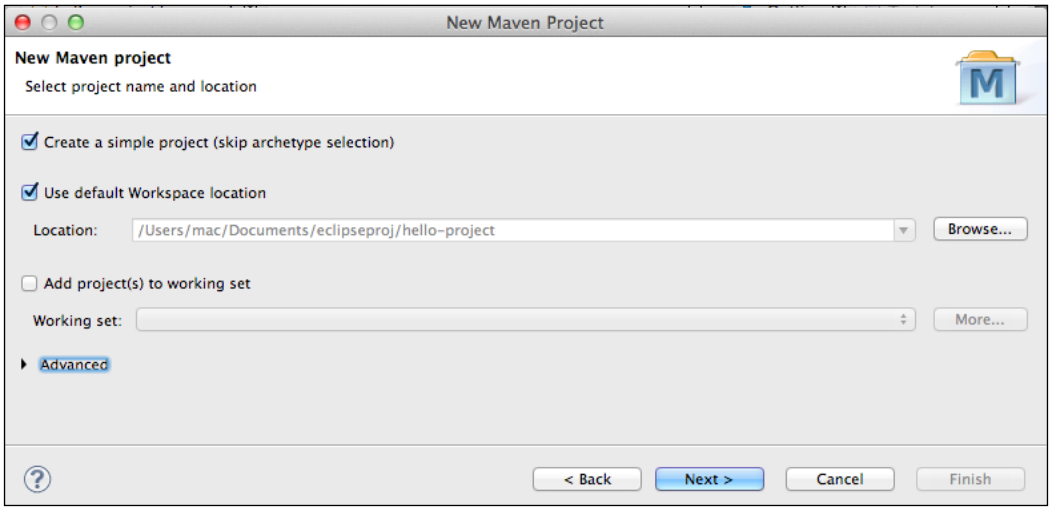
A sample Maven project has now been created, and it contains a Java file that prints **Hello World!**. The project has the following structure:



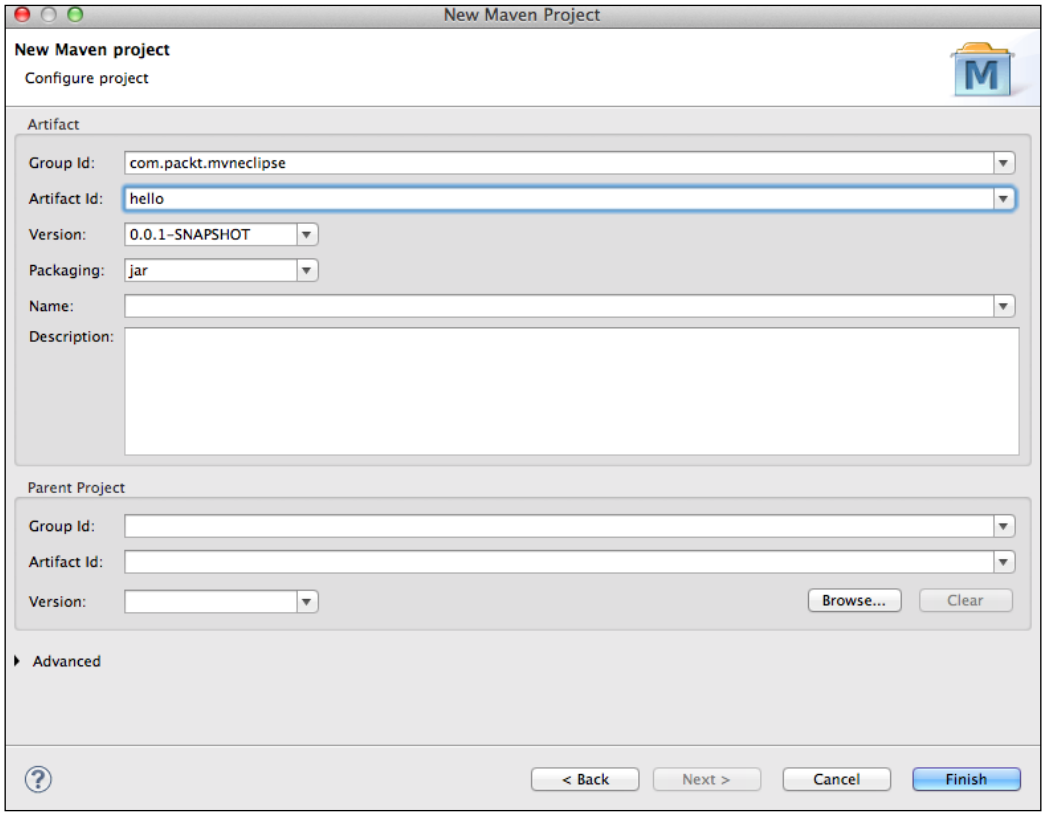
Using no archetypes

You can create a Maven project without archetypes using the following steps:

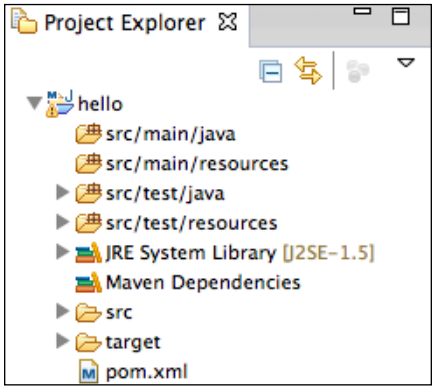
1. Navigate to **File** | **New** and click on **Other**. The project wizard appears and expands the `Maven` folder. Select **Maven Project** and click on **Next**.
2. The **New Maven Project** wizard appears. Use the default workspace and check the **Skip archetype** checkbox as shown in the following screenshot and click on **Next**:



3. The wizard to specify the Maven coordinates appears. Please fill in the details, as shown in the following screenshot, and click on **Finish**:



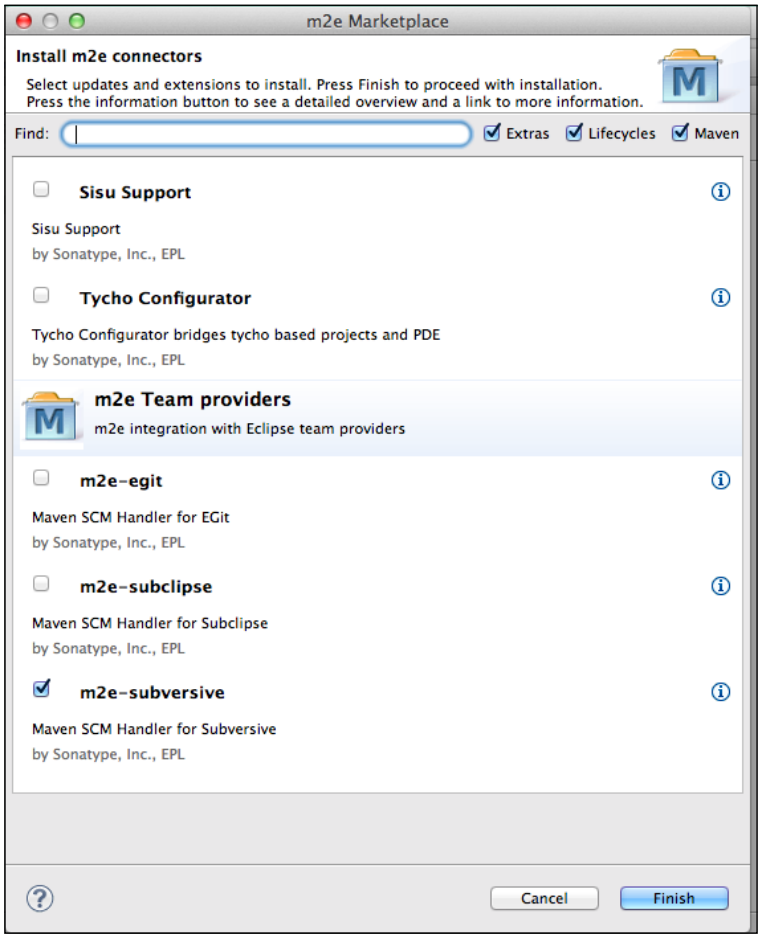
4. A skeleton structure, as shown in the following screenshot, will be created, and we have customized it according to the type of application we are building:



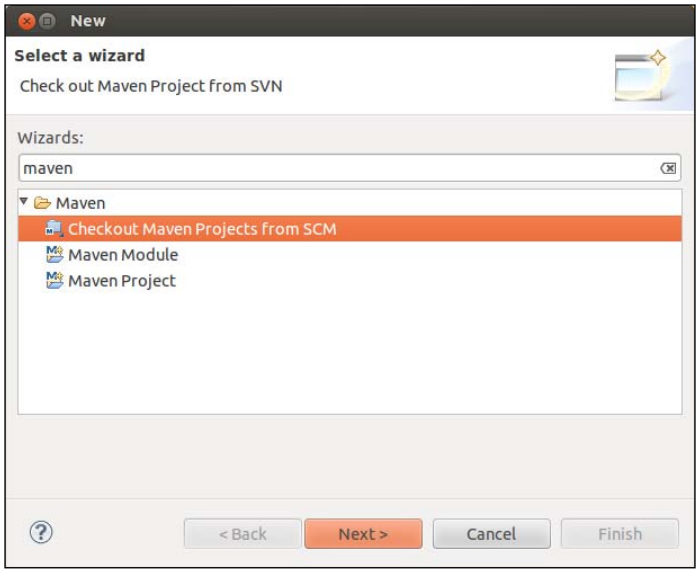
Checking out a Maven project

Checking out a Maven project means checking out from the source code versioning system. Before we process this, we need to make sure we have the Maven connector installed for the corresponding SCM we plan to use. Use the following steps to check out a Maven project:

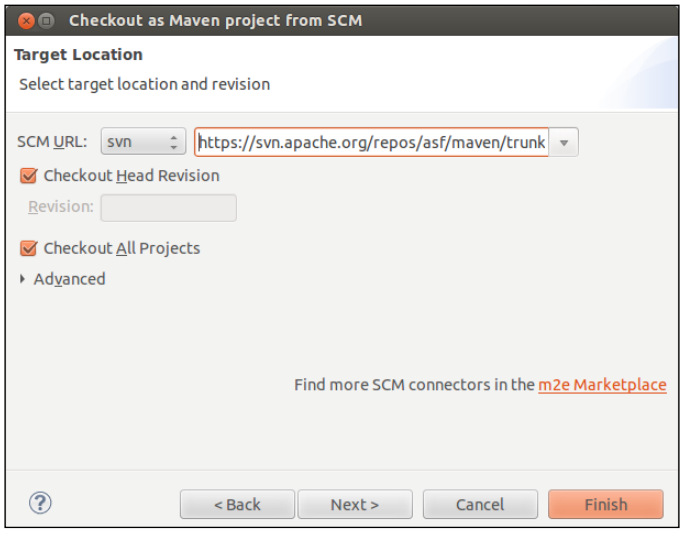
1. Navigate to **Eclipse** | **Preferences** in Mac, else **Windows** | **Preference search** in other OS, and search for `Maven`, expand it, and click on **Discovery**.
2. Then, click on **Open Catalog**. This lists all the m2connectors available in the marketplace. In our case, we are going to use SVN, so choose **m2-Subversive**, as shown in the following screenshot, and click on **Finish**. In the screens to follow, click on **Next**, accept the license, and finally click on **Finish** to install it. Similarly, we can choose any connector we intend to use for SCM.



3. Like how you create projects, navigate to **File | New** and click on **Other**. The project wizard appears. Expand the Maven folder. Click on **Checkout Maven Projects from SCM** and click on **Next**.



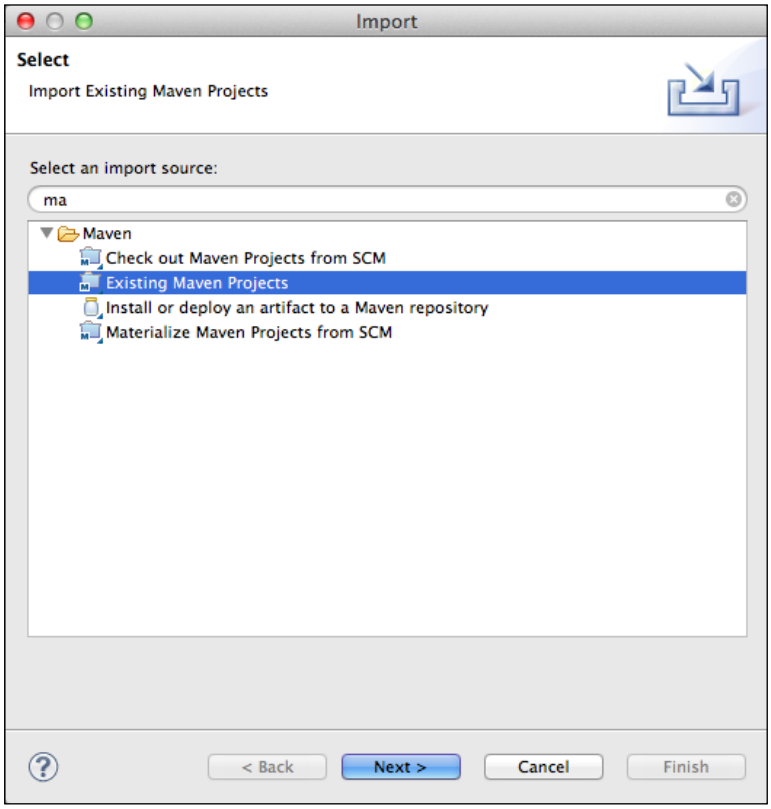
4. In the next screen, choose the SCM connector **SVN** and provide the corresponding SVN URL, as shown in the following screenshot, and click on **Finish**. If you click on **Next**, it will show the repository structure.



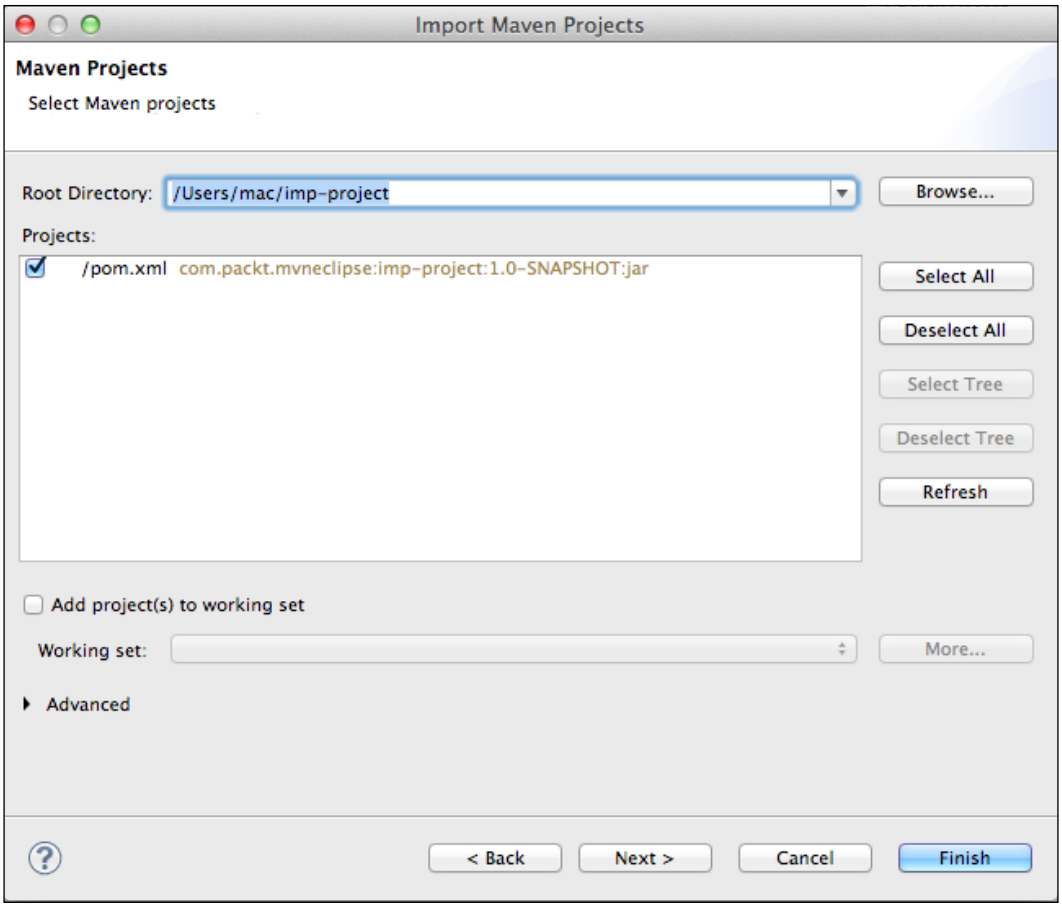
Importing a Maven project

Importing a Maven project is like importing any other Java project. The steps to import a Maven project are as follows:

1. From the **File** menu, click on **Import**. Choose `Import`, a source window appears, expand `Maven` and click on **Existing Maven Projects** as shown in the following screenshot:



2. In the next wizard, we have to choose the Maven project's location. Navigate to the corresponding location using the **Browse...** button, and click on **Finish** to finish the import as shown in the following screenshot; the project will be imported in the workspace:



Summary

Congratulations! In this chapter, you got acquainted with the Maven project structure, the POM file, other essential concepts of the Maven realm, and finally you ended up learning how to create and import Maven projects. For more information, you can refer to *Maven: The Complete Reference* by Tim O'Brien, published by Sonatype, Inc., and the Apache Maven site. In the next chapter, we will look at the build cycle and you will learn how to run Maven projects.

4
Building and Running a Project

Congratulations! You are halfway through the book. As discussed in earlier chapters, Maven follows convention over configuration; this implies there is a default build mechanism in place. The build mechanism, often termed as the **build lifecycle**, forms a sequence of steps grouped together in phases (also known as **stages)**. Each **phase** is accompanied with a set of goals that define the unit of task. In this chapter, we will look at three standard lifecycles—clean, default, and site—and get acquainted with other common lifecycles. You will also get to know about building and running the `hello-project`, which was created in *Chapter 3*, *Creating and Importing Projects*.
This chapter covers the following sections:

- Build lifecycle
    - Default lifecycle
    - Clean lifecycle
    - Site lifecycle
- Package-specific lifecycle
- The Maven console
- Building and packaging projects
- Running hello-project

The build lifecycle

Building a Maven project results in the execution of set goals grouped in phases. Though there is a default build cycle of Maven, it can be customized to suit our needs; that's the beauty Maven inherits. To ascertain, it is essential to have knowledge of the build's lifecycle. Essentially, the following are the three standard lifecycles in Maven:

- Default
- Clean
- Site

The default lifecycle

The default lifecycle handles the build of the project and its deployment. It is the primary lifecycle of Maven and is also known as the build lifecycle. In general, it provides the build process model for Java applications. There are 23 phases for the default lifecycle that starts with validation and ends with deploy. For details on all 23 phases, please refer to `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference`.

However, here we will see some of the phases and the default associated goals that need attention for common application development, which are as follows:

Lifecycle phases	Description	Plugin:goals
validate	This validates that the project is correct and contains all the necessary information to perform the build operation	-
compile	This compiles the source code	`compiler:compile`
test-compile	This compiles the test source code in the test destination directory	`compiler:testCompile`
test	This runs the test using suitable unit testing framework as configured in the `pom` file	`surefire:test`
package	This packages the compiled source code in the corresponding distributable format such as JAR, WAR, EAR, and so on	`jar:jar` (for JAR packaging)

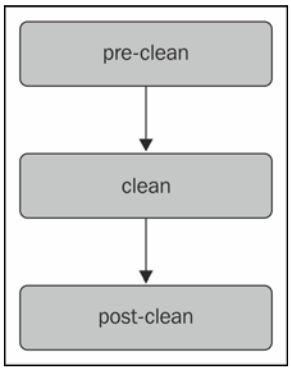
Lifecycle phases	Description	Plugin:goals
install	This installs the package in the local repository, which can act as a dependency for other projects	`install:install`
deploy	This copies the final package to a remote repository to share with other developers and projects	`deploy:deploy`

The clean lifecycle

The clean lifecycle is the simplest lifecycle in Maven, and it consists of the following phases:

- **pre-clean**: This phase executes the process needed before a project's clean up
- **clean**: This phase removes all files built by an earlier `build` (the `target` directory)
- **post-clean**: This phase executes the process required after a project's cleanup

Out of these phases, the one that gathers our interest is the **clean** phase. The Maven "`clean:clean`" goal is bound to the clean phase. It cleans the project's `build` (usually `target`) directory. Executing any one phase result in execution of all phases up to it and the phase itself, for example, a call of a clean phase would execute the first pre-clean phase and then the clean phase; similarly, a call of post-clean results in the calling of pre-clean, clean, and post-clean phases. The following diagram illustrates the execution of the clean lifecycle phases (reference: the Apache Maven site):

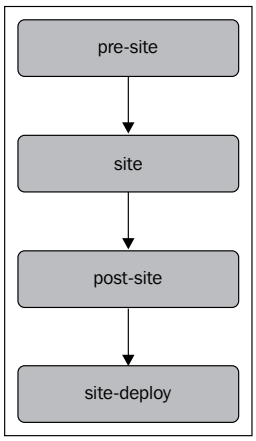


We can bind other goals to the phases of the clean lifecycle. Suppose we want to echo some message on the pre-clean phase; we can achieve this by binding the `maven-antrun-plugin:run` goal to this phase, which can be done as follows:

```
</project>
  ..........
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-antrun-plugin</artifactId>
      <executions>
        <execution>
          <id>precleanid</id>
          <phase>pre-clean</phase>
          <goals>
            <goal>run</goal>
          </goals>
          <configuration>
            <tasks>
              <echo>Hello am in pre-clean phase</echo>
                </tasks>
          </configuration>
        </execution>
```

The site lifecycle

The site lifecycle handles the creation of the project site documentation. The phases of a site lifecycle are shown in the following diagram:



The following table describes the site lifecycle phases in the order of execution. (reference: Apache Maven website)

Phases	Description
pre-site	This phase executes processes needed before the generation of a project site.
site	This phase generates documentation of a project site
post-site	This phase executes a process required after a site's generation and to prepare for site deployment
site-deploy	This phase deploys the generated site documentation to the specified web server

Executing any one phase results in the execution of all phases up to it and the phase itself. For example, calling post-site results in the execution of pre-site, site, and post-site phases. Similar to the clean lifecycle, we can bind other goals to the site's lifecycle.

The package-specific lifecycle

Each type of packaging has its own set of default goals. The default goals for JAR packaging is different from WAR packaging. Maven provides a lifecycle for the following built-in packaging types:

- JAR
- WAR
- EAR
- POM
- EJB
- Maven plugins

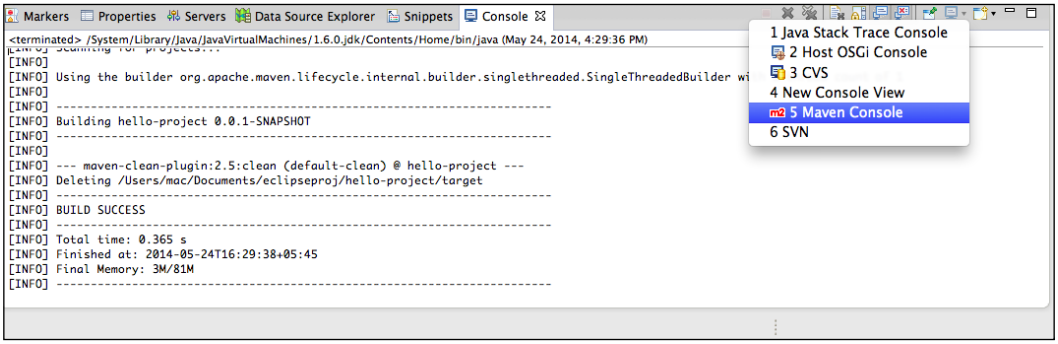
The lifecycle and goal binding for WAR packaging is described here. For other packaging lifecycle and goal binding, please refer to `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Built-in_Lifecycle_Bindings`.

Phases	plugin:goals
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`

The Maven console

Before we get our hands dirty with building and executing Maven projects, we need to enable the Maven console. The Maven console can be enabled with the following steps:

1. Navigate to **Window** | **Show View** | **Console**. This shows the console view.
2. Next, click on the little arrow of **Open Console** as shown in the following screen and click on **Maven Console**:

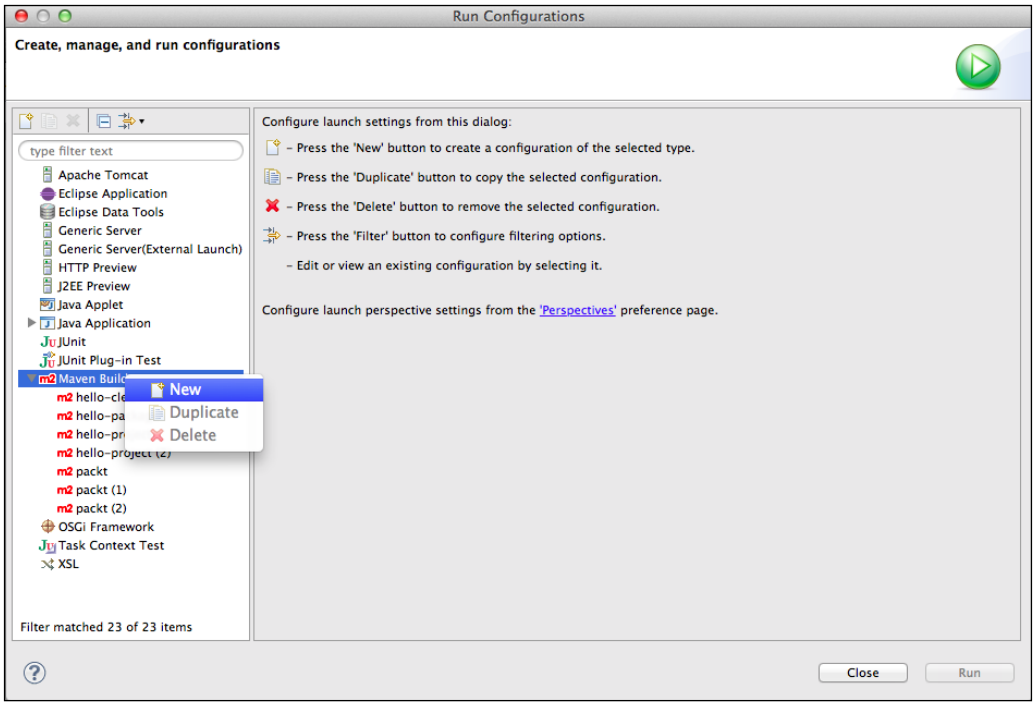


The Maven console shows all the output of the Maven build process. It shows all the details that Maven processes and is really helpful in getting to know what is happening underneath and you can also see the debug messages.

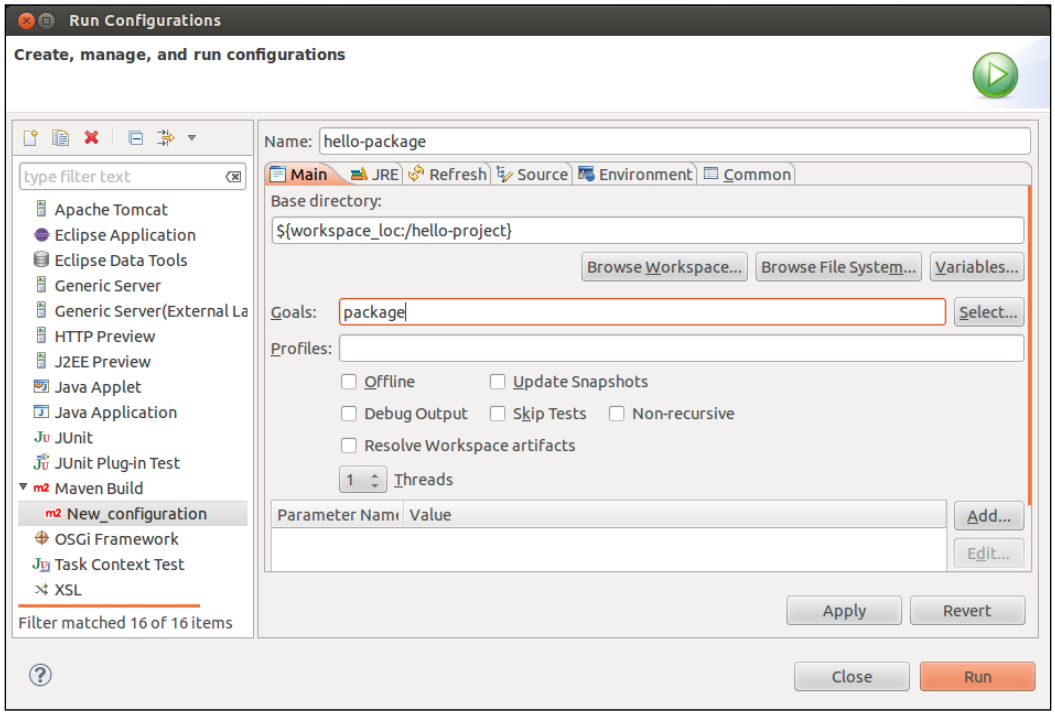
Building and packaging projects

Building and packaging Maven projects needs execution of required phases, which we discussed in the preceding sections. Let's build and package `hello-project` from *Chapter 3*, *Creating and Importing Projects*, which we generated using archetypes. In the *Default Lifecycle* section, the phase package executes the following phases in order: compile, test, and package phases. Now, we will see how to invoke the package phase from m2eclipse. The following steps will ascertain this:

1. Right-click on `hello-project` and select **Run As**. Click on **Run Configurations** and the **Run Configurations** window will appear.
2. Right-click on **Maven Build** and choose **New** as shown in the following screenshot:

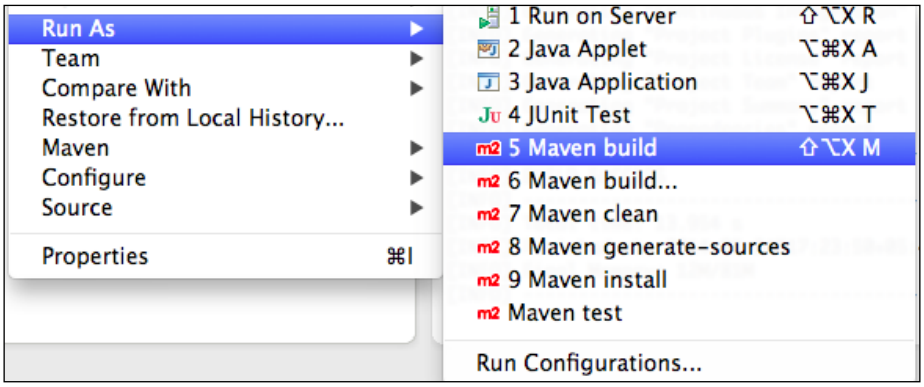


3. Once the launch configurations window appears, fill in the details as shown in the following screenshot. For **Base Directory**, click on **Browse Workspace...** and choose `hello-project` from the pop-up list:

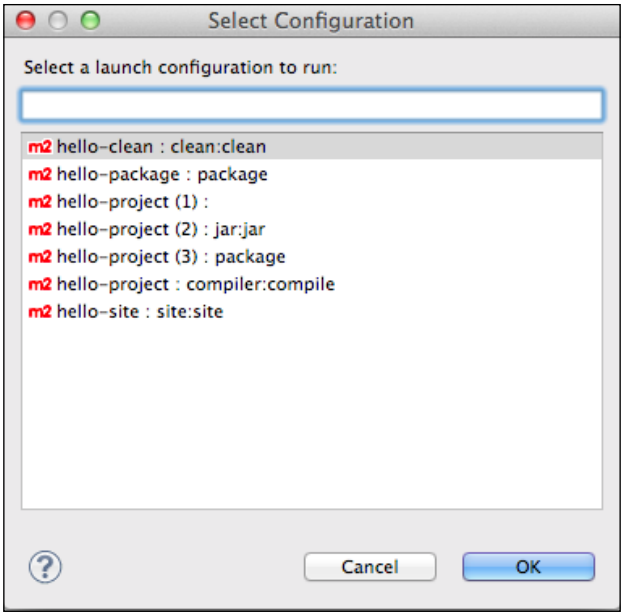


4. Next, click on **Apply** and close it using the **Close** button.

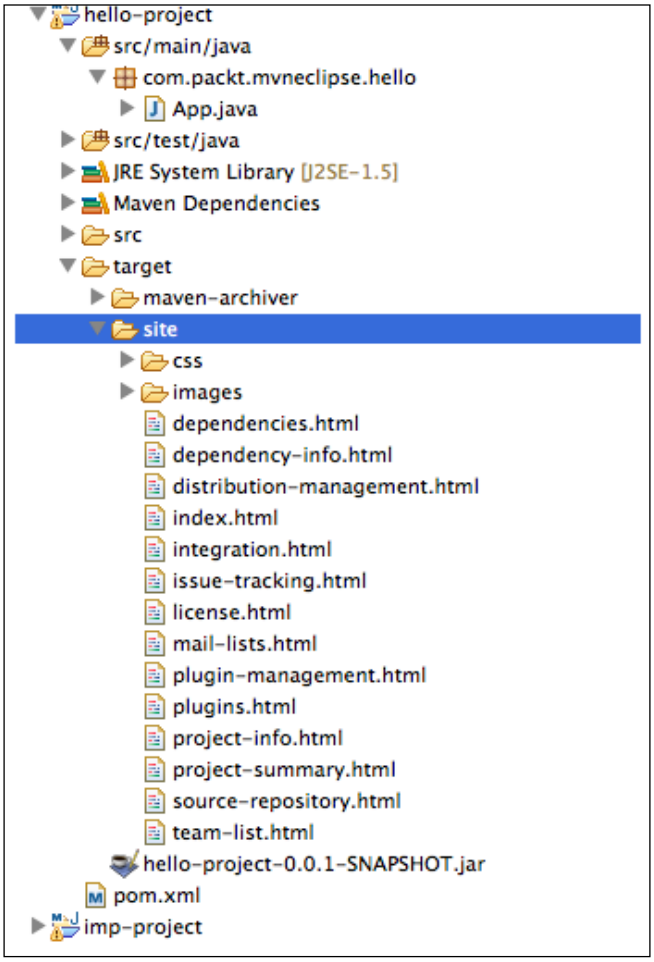
5. Again right-click on the project and select **Run As**, and click on **Maven build** as shown in the following screenshot:



6. A window, as shown in the following screenshot, will appear with all the run configurations available:



7. Choose the **hello-package** launch configuration and click on **OK**. It should compile, run tests, generate site documentation, and package in the `target` directory, as shown in following screenshot:



Running hello-project

Since `hello-project` from the previous chapter is a Java application, running it is similar to any other Java application. Right-click on the project, select **Run As**, and click on **Java Application**, select the main JAVA class, and click on **OK**. It will print `Hello World!` in the console.

Running a web application requires some extra steps, which we will discuss in *Chapter 5*, *Spicing Up a Maven Project*.

Summary

In this chapter, you learned about the clean, site, and default build lifecycles of the Maven project, and later used this knowledge to get the application to package and run.

In the next chapter, we will build a web application and you will learn to customize the `pom` file to suit our needs.

5
Spicing Up a Maven Project

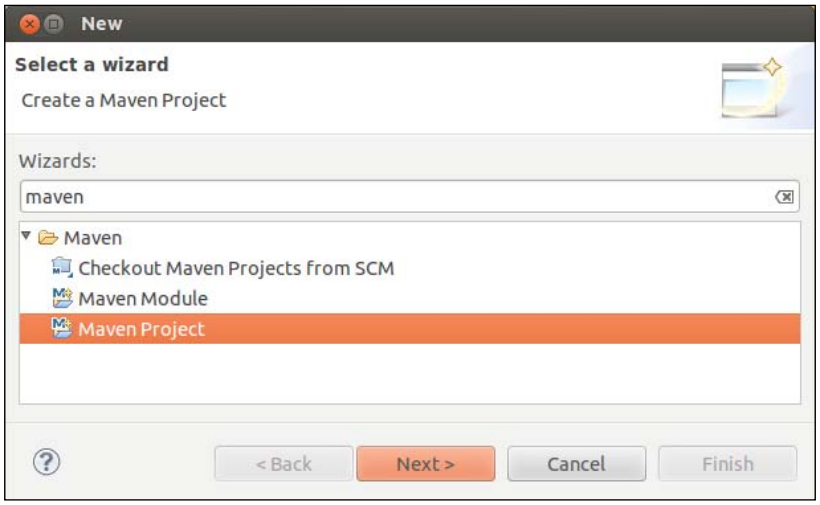
So far we have built the base, and now we are finally ready to launch the rocket. Rocket! Exciting, isn't it? Let's put our knowledge from previous chapters to practice; we will use Maven to create a simple web application, MyDistance, which lets the user convert distance between different units. In the process of building this application, we will also learn to customize the project's information and generate different artifacts. The topics that will be covered in this chapter are categorized as follows:

- Creating the MyDistance project
- Changing the project information
- Adding dependencies
- Adding resources
- The application code
    - Adding a form to obtain an input
    - Adding the servlet
    - Adding a utility class
- Running an application
- Writing unit tests
- Running unit tests
- Generating site documentation
- Generating unit tests—HTML reports
- Generating javadocs

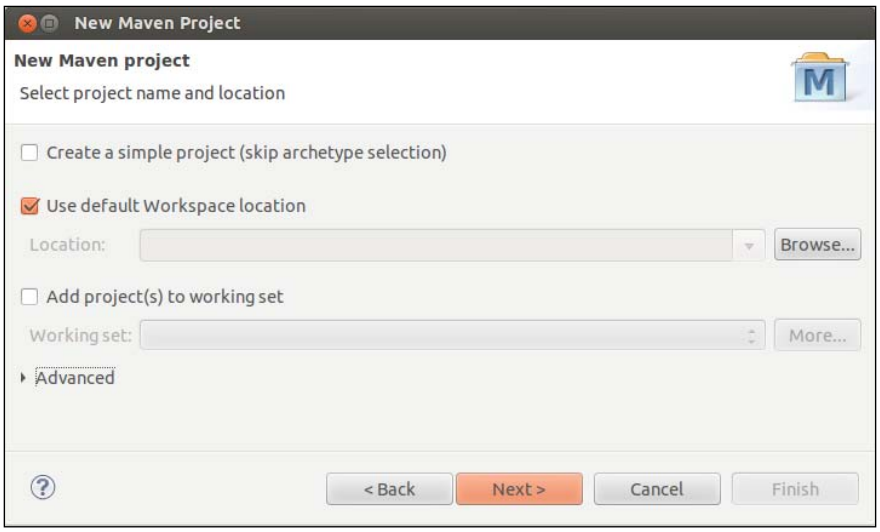
Creating the MyDistance project

To create the `MyDistance` application, we need to perform the following steps:

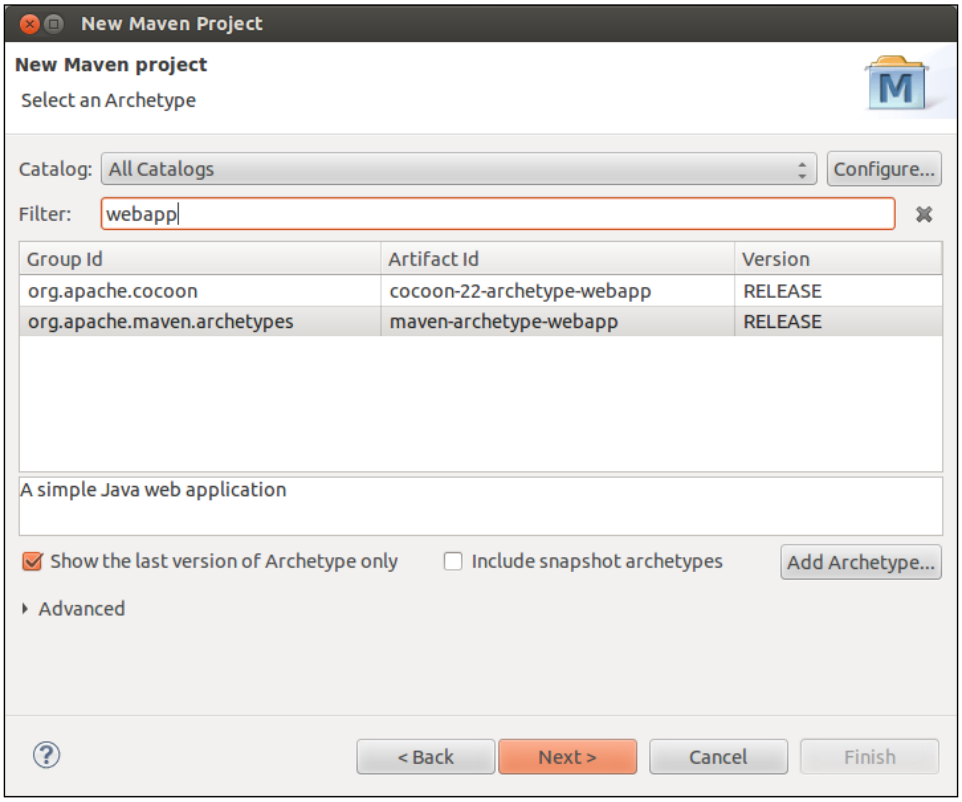
1. From the menu, navigate to **File** | **New** | **Other...**. A new project wizard window appears. Search for `maven` in the textbox, select **Maven Project**, and click on the **Next** button, as shown in the following screenshot:



2. A **New Maven Project** wizard appears; select the **Use default Workspace location** checkbox, and ensure the **Create a simple project (skip archetype selection)** checkbox is unchecked, as shown in the following screenshot:



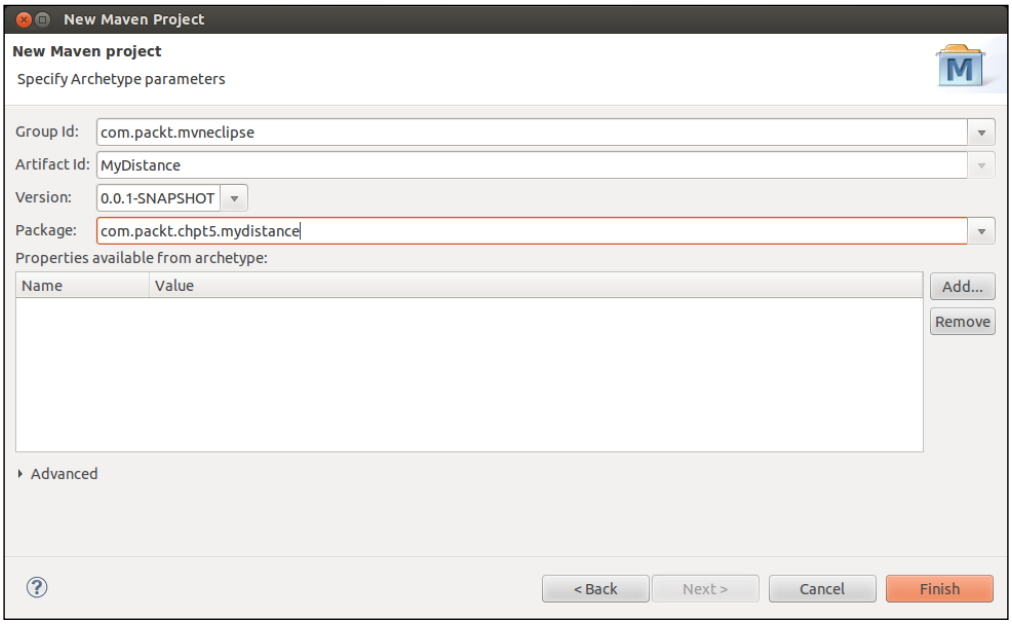
3. Next, choose an archetype from the new archetype wizard. Since we are building a web application, in **Filter**, search for `webapp`, choose **maven-archetype-webapp**, and click on **Next**, as shown in the following screenshot:



4. Specify the Maven coordinates, also termed as **Group-Artifact-Version** (**GAV**) in technical parlance, with the following values, and click on **Finish**:

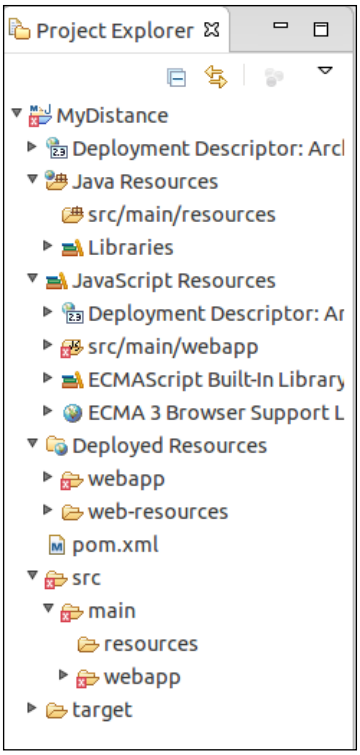
Field	Value
Group Id	`com.packt.mvneclipse`
Artifact Id	`MyDistance`
Version	`0.0.1-SNAPSHOT`
Package	`com.packt.chpt5.mydistance`

Your screen will look like the following screenshot once you perform the previous step:



A snapshot in Maven indicates the current development copy, that is, the current snapshot of the code. Maven checks for a new SNAPSHOT version in a remote repository at a configured interval, for a default time of 24 hours. For more information on Maven versions, refer to `http://docs.oracle.com/middleware/1212/core/MAVEN/maven_version.htm`.

5. The web application skeleton gets created and the structure would look like the following screenshot:



Don't worry if you see a red cross that indicates an error in the project; we will learn more about it in the upcoming section, *Application code*.

Changing the project information

Before we venture into further details of the code, let's customize the project information. Let's add information about the organization, license, and developers associated with it. To do this, let's open the `pom` file and add the following code:

```
<project >
…….
  <!-- Organization information -->
    <organization>
      <name>Packt Publishing</name>
      <url>www.packtpub.com</url>
    </organization>
```

```
    <!-- License information  -->
    <licenses>
      <license>
         <name>Apache 2</name>
        <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
        <distribution>manual</distribution>
          <comments>A Friendly license</comments>
      </license>
    </licenses>
    <!-- Developers Information -->
    <developers>
      <developer>
        <id>foo</id>
          <name>Foo foo</name>
          <email>foo@foo.com</email>
          <url>http://www.foofoo.net</url>
          <organization>Packt</organization>
          <organizationUrl>http://packtpub.com</organizationUrl>
              <roles>
                <role>developer</role>
              </roles>
              <timezone>-8</timezone>
      </developer>
    </developers>
.......
</project>
```

For detailed information on the Maven model, visit `http://maven.apache.org/ref/3.2.1/maven-model/maven.html`.

Adding dependencies

Our project is a simple web application, and to begin, it will need JUnit as a dependency for testing and `log4j` for logging purposes. As we progress further, we will add more dependencies progressively; the idea of this section is to show how to add dependencies in the `pom` file. If we see our `pom` file, we can see that JUnit is already present as a dependency; so, let's add `log4j` as a dependency by adding the following code snippet:

```
<project>
….....
<dependencies>
…
```

```
<!-- For logging purpose -->
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
</dependencies>
….....
</project>
```

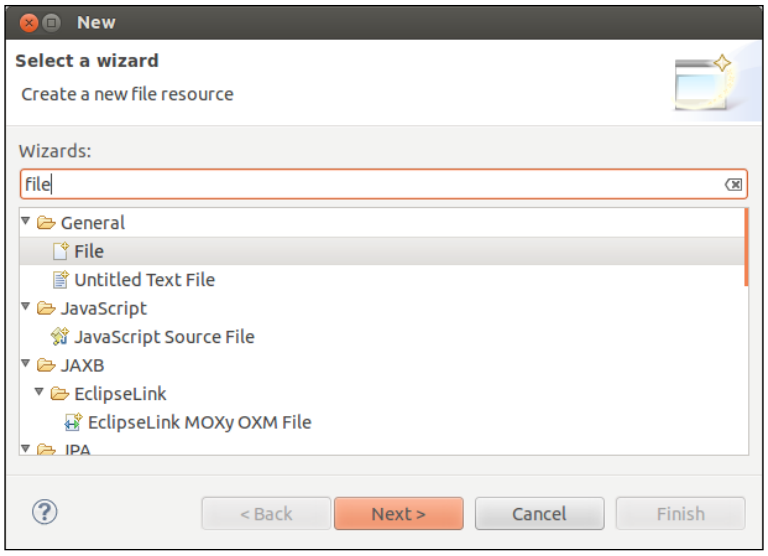
The complete resultant `pom` file would look like the following:

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.packt.mvneclipse</groupId>
  <artifactId>MyDistance</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>MyDistance Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <!-- Organization information -->
  <organization>
    <name>Packt Publishing</name>
    <url>www.packtpub.com</url>
  </organization>

  <!-- License information  -->
  <licenses>
    <license>
       <name>Apache 2</name>
       <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
       <distribution>manual</distribution>
       <comments>A Friendly license</comments>
    </license>
  </licenses>
  <!-- Developers Information -->
  <developers>
    <developer>
    <id>foo</id>
      <name>Foo foo</name>
      <email>foo@foo.com</email>
```

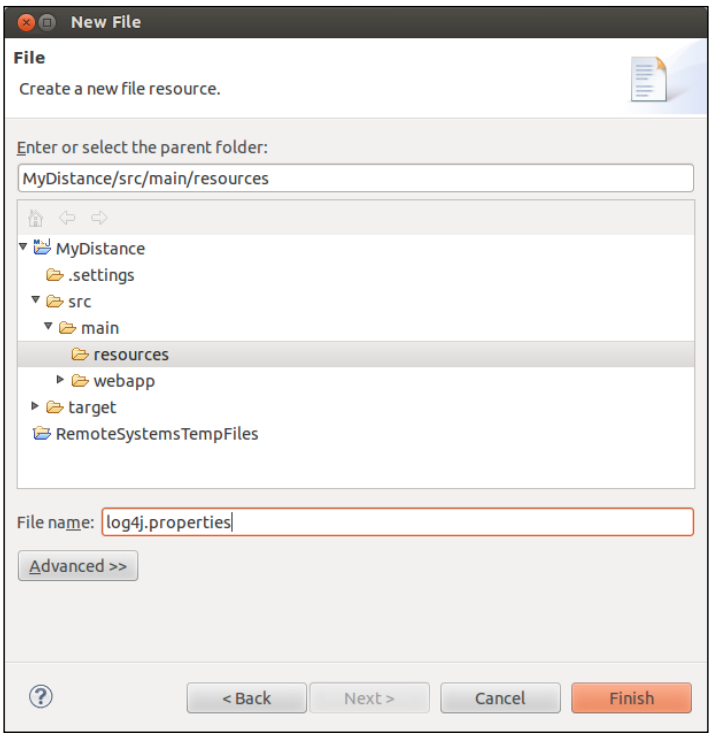
```
        <url>http://www.foofoo.net</url>
        <organization>Packt</organization>
        <organizationUrl>http://packtpub.com</organizationUrl>
        <roles>
          <role>developer</role>
            </roles>
              <timezone>-8</timezone>
      </developer>
      </developers>
    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
      </dependency>
      <!-- For logging purpose -->
      <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
      </dependency>
    </dependencies>
    <build>
      <finalName>MyDistance</finalName>
    </build>
  </project>
```

Adding resources

We are going to use `log4j` to log in to the file or console. Log4j is configured via the `log4j` properties file. Now let's create the properties file. To do so, navigate to `src/main/resources`, right-click on resources and select **New** | **Other...**; a new wizard appears. Search for `file` in the **Filter** section, select **File**, and click on **Next**, as shown in the following screenshot:



Next, a **File** resource window appears; enter the filename as `log4j.properties` and make sure the parent folder is `MyDistance/src/main/resources` and click on **Finish**, as shown in the following screenshot:



Resources are placed in the `src/main/resources` folder.

Once the file is created, add the following piece of code to set the different properties of `log4j`. It attaches the pattern layout to split out information on the console, writes a log to the `Mydistance.log` file, and is set to the `DEBUG` level, as shown in the following code:

```
#log4j Properties
  log4j.rootLogger=DEBUG, consoleAppender, fileAppender

  log4j.appender.consoleAppender=org.apache.log4j.ConsoleAppender
  log4j.appender.consoleAppender.layout=org.apache.log4j.PatternLayout
  log4j.appender.consoleAppender.layout.ConversionPattern=[%t] %-5p
%c{1} %x - %m%n

  log4j.appender.fileAppender=org.apache.log4j.RollingFileAppender
  log4j.appender.fileAppender.layout=org.apache.log4j.PatternLayout
  log4j.appender.fileAppender.layout.ConversionPattern=[%t] %-5p
%c{1} %x - %m%n
  log4j.appender.fileAppender.File=Mydistance.log
```

For more information on `log4j`, refer to `http://logging.apache.org/log4j/1.2/`.

The application code

The application is deliberately created in JSP or servlets to keep it simple and to avoid having familiarity with other frameworks to understand the example. Before we get into the basics of the application code, let's solve the error that Eclipse complains of in step 5 of the *Creating a MyDistance Project* section. Add the following dependency in the `pom` file and the error should vanish:

```
<!-- Include servlet  API -->
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>
```

The preceding dependency will also be required for writing servlets later in the *Adding a Servlet* section.

The scope is provided, which means that the container will provide this dependency, and Maven will not include it in this project's output or war file. We discussed scopes in more detail in *Chapter 4, Building and Running a Project*.

The application will require the following additional files:

- `index.jsp`: This is a JSP file with a form that allows users to enter a distance, its unit, and the desired conversion unit
- `DistanceServlet`: This is a servlet that processes the inputs from the form
- `ConversionUtil`: This is a utility class that has a method to perform conversion between different units

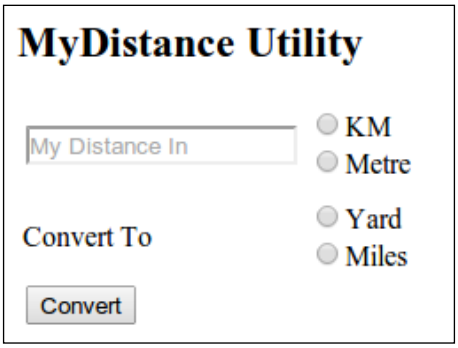
Adding a form to get an input

Under `src/main/webapp`, open the `index.jsp` file, and add the following code to get the distance, its unit, and conversion unit as input. The form consists of an input box, two radio buttons to choose units, and a button to initiate the conversion, as shown in the following code:

```
<body>
<h2>MyDistance Utility</h2>
  <form>
    <table>
      <tr>
        <td><input type="text" id="mydistance" name='distance'
          placeholder="My Distance In"></td>

        <td> <input type="radio" name="distin" id="distin"
          value="km">KM<br>
        <input type="radio" name="distin" id="distin"
          value="m">Metre</td>
      </tr>
      <tr></tr>
      <tr></tr>
      <tr></tr>
      <tr>
```

```
          <td> <label for="convert">Convert To</label></td>
          <td> <input type="radio" name="convertto" id="convertto"
            value="yd">Yard<br>
          <input type="radio" name="convertto" id="convertto"
            value="mi">Miles</td>
        </tr>
        <tr>
          <td><input type="button" id="submit" value='Convert'></td>
        </tr>

      </table>
      <div id="convertvalue"> </div>
    </form>
  </body>
```

If you like, you can add CSS styles to make the UI more pleasing. The preceding bare bones file results in something like this:



We want to calculate the value and show the corresponding result beneath it using Ajax (jQuery Ajax). To achieve this, add the following piece of code:

```
<head>
  <script src="http://code.jquery.com/jquery-latest.js">
  </script>
  <script>
    $(document).ready(function() {
    $('#submit').click(function(event) {
      var mydistance=$('#mydistance').val();

      var mydistanceIn=$('[name=distin]:checked').val();
      var convertTo=$('[name=convertto]:checked').val();
      if(mydistanceIn==convertTo){
```

```
          alert("Cannot have same unit");
          return false;
        }
        console.log(mydistance+mydistanceIn+convertTo);
        $.get('mydistance',{distance:mydistance,distIn:
          mydistanceIn,convert:convertTo},function(responseText) {
            $('#convertvalue').text(responseText);
          });
        });
      });
    </script>
  </head>
```

Adding a servlet

Before we add any Java files, create a folder, `java`, under `src/main` as Maven looks for Java files in this directory (all Java files should reside under it). Add the `DistanceServlet` servlet in the `com.packt.chpt5.mydistance` package. The servlet gets the request parameters, extracts it, and calls the corresponding conversion method in the utility class. The servlet would look like the following:

```
public class DistanceServlet extends HttpServlet {

  private static final long serialVersionUID = 1L;
  static Logger log=Logger.getLogger(DistanceServlet.class);
  public void doGet(HttpServletRequest req, HttpServletResponse
    resp)
        throws ServletException, IOException {
        double convertVal = 0;
        double distanceProvided
          =Double.parseDouble(req.getParameter("distance"));
    String distanceIn=req.getParameter("distIn");
    String convertTo=req.getParameter("convert");
    log.debug("Request Parameters ==>"+"Distance-
      "+distanceProvided+distanceIn+" Conversion Unit- "+convertTo
      );
    ConversionUtil conversion= new ConversionUtil();
    if(distanceIn.equals("km") && convertTo.equals("yd")){
      convertVal=conversion.convertkmToYard(distanceProvided);
    }

    if(distanceIn.equals("m") && convertTo.equals("yd")){
```

```
            convertVal=conversion.convertMtoYard(distanceProvided);
        }

        if(distanceIn.equals("km") && convertTo.equals("mi")){
            convertVal=conversion.convertKMToMile(distanceProvided);
        }

        if(distanceIn.equals("m") && convertTo.equals("mi")){
          convertVal=conversion.convertMToMile(distanceProvided);
        }

        resp.setContentType("text/html");
          PrintWriter out = resp.getWriter();
          out.print("The converted value is "+convertVal);
          out.flush();
          out.close();

        }


    }
```

Add the following lines in the `web.xml` file under `src/main/webapp/WEB-INF`:

```
<web-app>
  <display-name>MyDistance Calculator</display-name>
  <servlet>
        <servlet-name>mydistance</servlet-name>
        <servlet-class>com.packt.chpt5.mydistance.DistanceServlet</
servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>mydistance</servlet-name>
        <url-pattern>/mydistance</url-pattern>
    </servlet-mapping>
</web-app>
```

Adding a utility class

Add a utility class `ConversionUtil` in the `com.packt.chpt5.mydistance.util` package. A utility class contains methods to perform conversion across different distance units. Add the following code to the utility class:

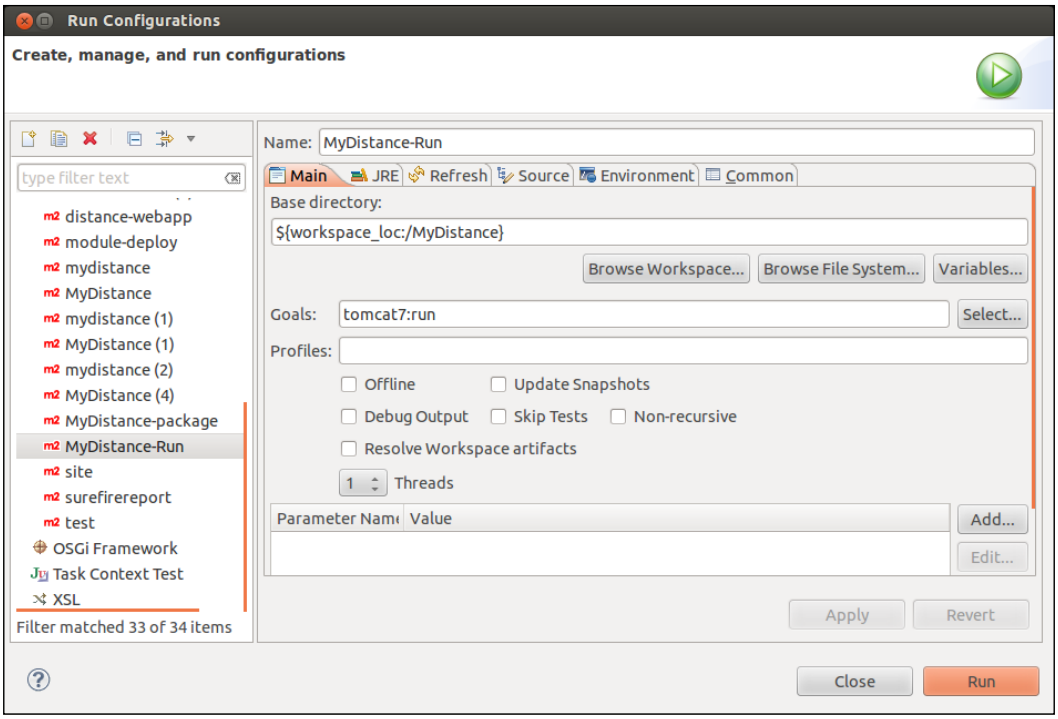
```
public double convertKMToMile(double distance){
  return (distance*0.62137);
}
public double convertkmToYard(double distance){
  return distance*1093.6;
}


public double convertMToMile(double distance){
  return (distance/1000)*0.62137 ;
}
public double convertMtoYard(double distance){
  return (distance/1000)*1093.6;
}
```

Running an application

Now we are done with coding. Let's run the coding and see how it works. The project is a web application, so it requires a servlet container to run. We will use the Tomcat container here. Add the following build plugin for the Tomcat that can actually host a Tomcat instance right from Maven and no installation is required:

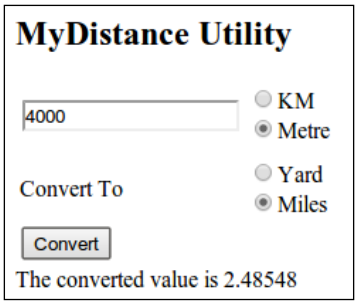
```
    <build>
….
  <plugins>
  <plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.1</version>
  </plugin>
  </plugins>
    </build>
```

This will download all the dependencies from the repository. After the download is complete, right-click on the project, choose **Run As | Run Configurations...**, create the configuration in the configuration window specifying **Goals** as `tomcat7:run`, and click on **Run**, as shown in the following screenshot:



The Tomcat plugin has the `run` goal, which compiles and runs the application.

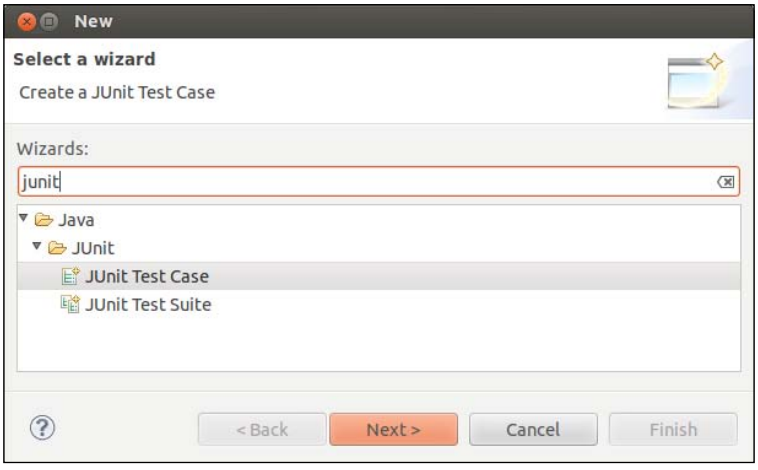
Similarly, we can add any other container and run the application in it. The running application will be available at `http://localhost:8080/MyDistance/` and would look like the following screenshot (shown with a sample conversion):



Writing unit tests

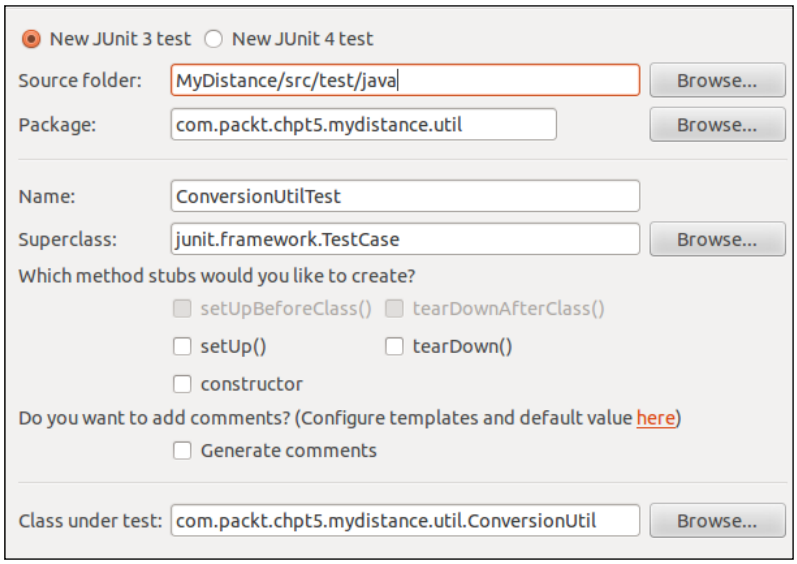
Writing unit tests is a part of good practice in software development. Maven's `test` phase executes unit tests and generates the corresponding report. In this section, we will learn about writing a simple unit test for our utility class `ConversionUtil`, and in the next section, we will see how to execute it and generate reports.

All the unit test classes should go under `src/test/java`. Create the corresponding folder in the `MyDistance` project. Once the folder is in place, right-click on it and navigate to **New** | **Other...**. Once the wizard window appears, type in `junit` in the **Filter** section, select **JUnit Test Case**, and click on **Next**, as shown in the following screenshot:

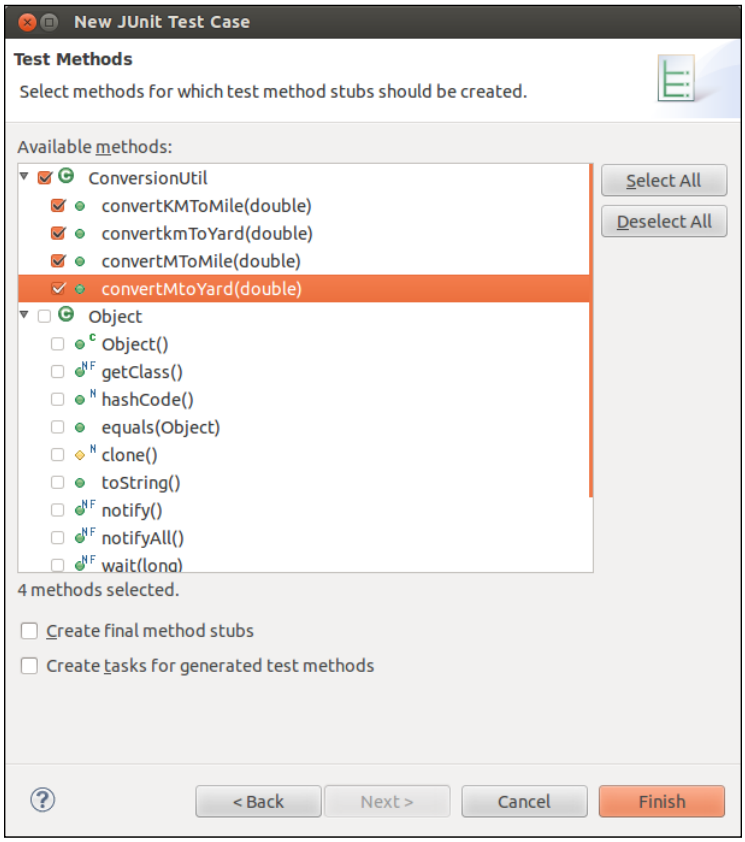


In the window to follow, define the unit test class by filling in the following details and click on **Next**, as shown in the preceding screenshot:

Fields	Values
Source folder	`MyDistance/src/test/java`
Package	`com.packt.chpt5.mydistance.util`
Name	`ConvertionUtilTest`
Class under test	`com.packt.chpt5.mydistance.util.ConversionUtil`



A window to choose test methods will be shown, for which stubs will be generated as shown in the following screenshot. Make sure that all methods of the `ConversionUtil` class are checked and click on **Finish** as follows:



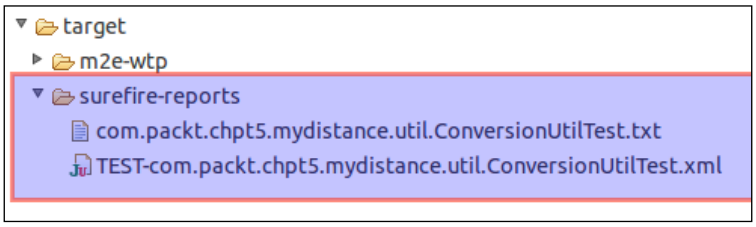
The `ConversionUtilTest` test class with the test method stubs will be generated. Edit the code of the class as follows:

```
private ConversionUtil conversion;

  @Override
  protected void setUp() throws Exception {

    super.setUp();
    conversion= new ConversionUtil();
  }

  public void testConvertKmToMile() {
    double actual=conversion.convertKMToMile(4);
    assertEquals(2.48548,actual,0.001);
  }

  public void testConvertkmToYard() {
    double actual=conversion.convertkmToYard(4);
    assertEquals(4374.45,actual,0.10);
  }

  public void testConvertMToMile() {
    double actual=conversion.convertMToMile(4000);
    assertEquals(2.48548,actual,0.001);
  }

  public void testConvertMtoYard() {
    double actual=conversion.convertMtoYard(4000);
    assertEquals(4374.45,actual,0.10);
  }

  @Override
  protected void tearDown() throws Exception {

    super.tearDown();
    conversion = null;
  }
```

For more information on JUnit test cases, refer to `http://junit.org/`.

Running unit tests

Running the unit test in Maven is just specifying the phase test. To execute the unit test we wrote in the preceding section, right-click on the `MyDistance` project, select **Run As**, and click on **Maven Test**. It will run the unit tests against the class and generate the report in the `/target/surefire-reports/` folder, as shown in the following screenshot:



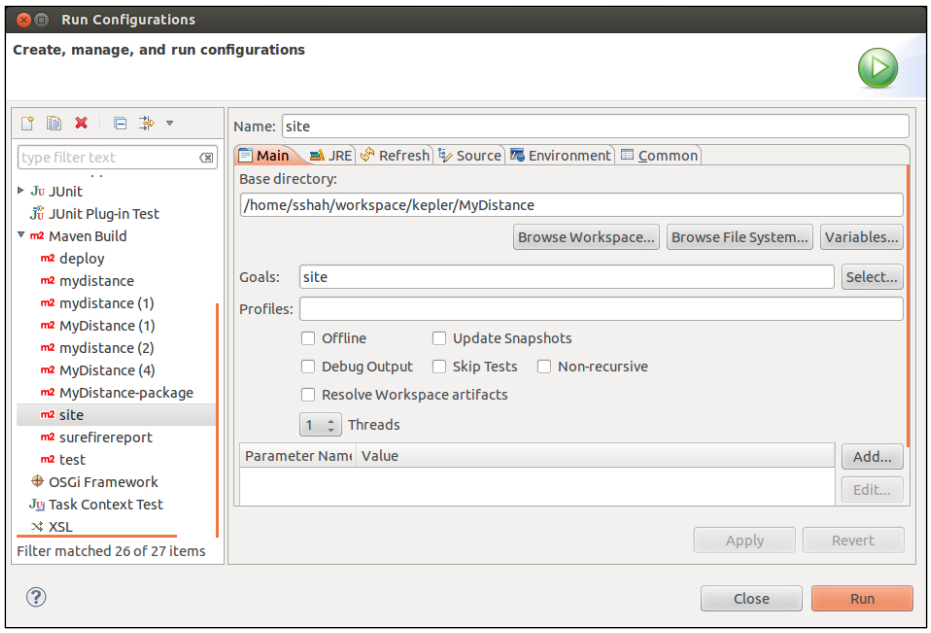
You can see the results of unit test execution in the `txt` and `xml` format.

Generating site documentation

One of the integral features of Maven is that it eases artifacts and site documentation generation. To generate site documentation, add the following dependency in the `pom` file:

```
<reporting>
    <plugins>
    <!--  Reporting -document generation -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>3.3</version>
      </plugin>
        <plugin>
      <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-project-info-reports-plugin</artifactId>
  <version>2.7</version>
      </plugin>
    </plugins>

  </reporting>
```

After adding the preceding dependencies, run the project with the goal as site, that is, in the **Run Configurations** window, specify **Goals** as `site`, as shown in following screenshot:

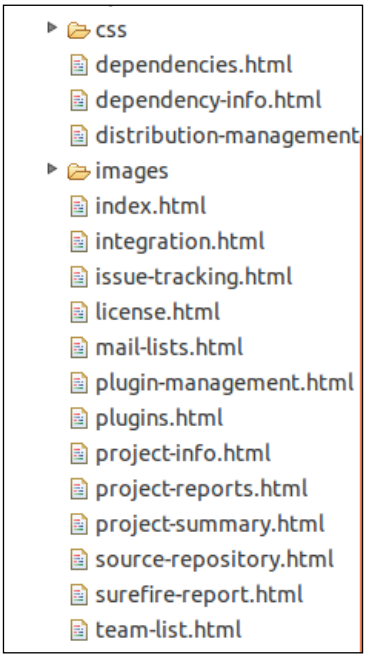


Click on the **Run** button and the documentation will be generated. The excerpts of the output in **Maven Console** would look like the following:

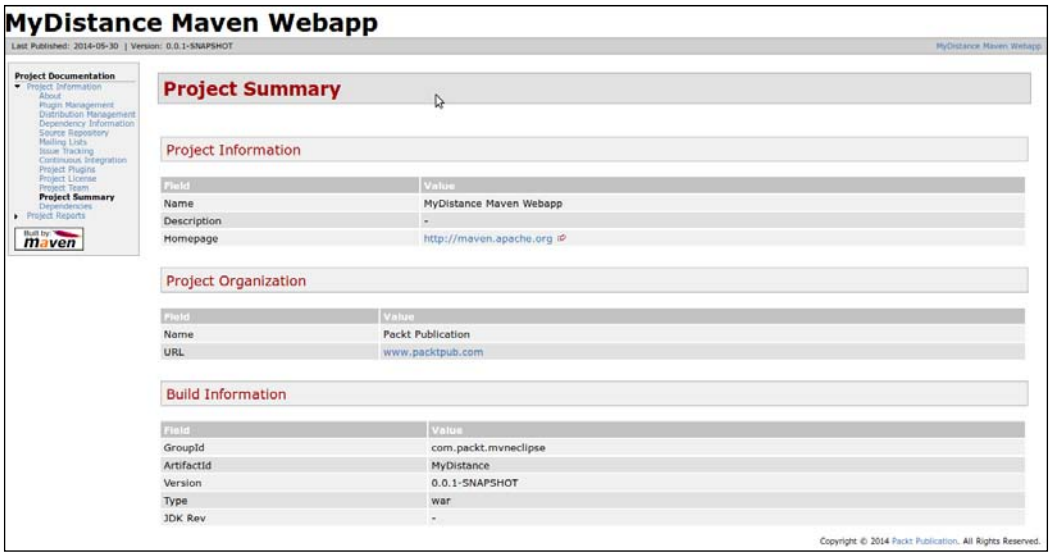
```
[INFO] Generating "About" report    --- maven-project-info-reports-
plugin:2.7
[INFO] Generating "Plugin Management" report    --- maven-project-
info-reports-plugin:2.7
[INFO] Generating "Distribution Management" report    --- maven-
project-info-reports-plugin:2.7
[INFO] Generating "Dependency Information" report    --- maven-
project-info-reports-plugin:2.7
[INFO] Generating "Source Repository" report    --- maven-project-
info-reports-plugin:2.7
[INFO] Generating "Mailing Lists" report    --- maven-project-info-
reports-plugin:2.7
[INFO] Generating "Issue Tracking" report    --- maven-project-info-
reports-plugin:2.7
```

```
[INFO] Generating "Continuous Integration" report    --- maven-
project-info-reports-plugin:2.7
[INFO] Generating "Project Plugins" report    --- maven-project-info-
reports-plugin:2.7
[INFO] Generating "Project License" report    --- maven-project-info-
reports-plugin:2.7
[INFO] Generating "Project Team" report    --- maven-project-info-
reports-plugin:2.7
[INFO] Generating "Project Summary" report    --- maven-project-info-
reports-plugin:2.7
[INFO] Generating "Dependencies" report    --- maven-project-info-
reports-plugin:2.7
[INFO] ------------------------------------------------------------------
----------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
----------
```

The documentation would be generated in the `target/site` folder and the expansion of the folder would look like the following:



There is an HTML file for each type of detail ranging from `project-info`, `project reports`, `project summary`, `license`, `plugin`, and so on, and `index.html` being the start point that links every document. The **Project Summary** page is shown in the following screenshot:



For more information on site and site plugins, please refer to `http://maven.apache.org/guides/mini/guide-site.html` and `http://maven.apache.org/plugins/maven-site-plugin/`.

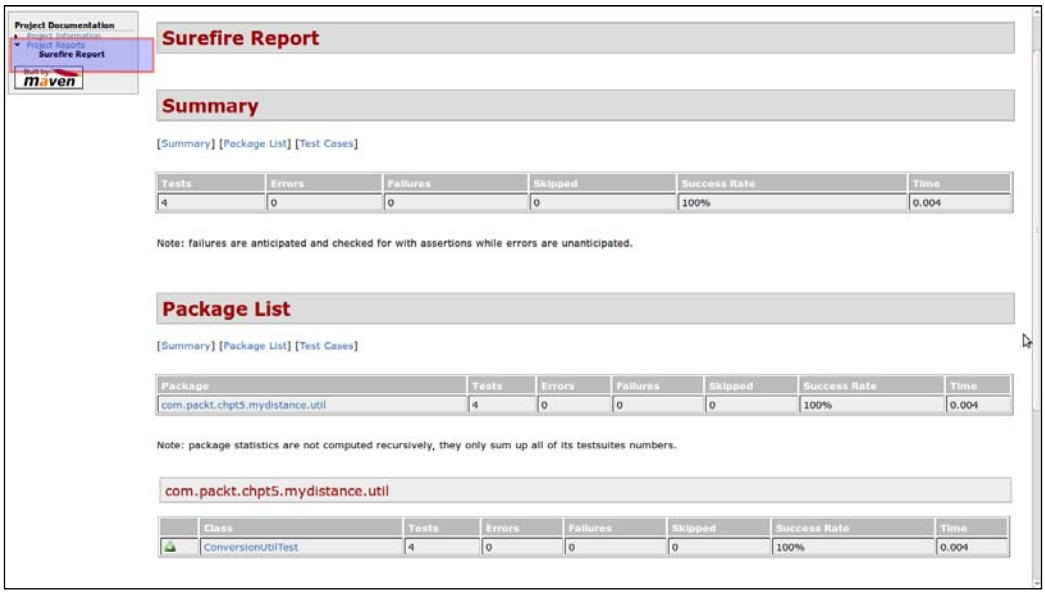
Generating unit tests – HTML reports

In the preceding section, we ran the unit tests, and the results were generated in the `txt` and `xml` format. Often, developers need to generate more readable reports. Also, as a matter of fact, the reports should be a part of site documentation for better collaboration and information available in one place. To generate an HTML report and make it a part of site documentation, add the dependency under the `reporting` element as `plugin` in the `pom` file as follows:

```
<reporting>
    <plugins>
….. .
<!-- For HTML test report generation -->
      <plugin>
```

```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>2.17</version>
          </plugin>
    ….. .
        </plugins>
    </reporting>
```

After the addition shown in the preceding code, run the `site` phase from the previous section. The test reports would be available by navigating to **Project Documentation** | **Project Reports** | **Surefire Report** of the navigation in `index.html`, as shown in the following screenshot:



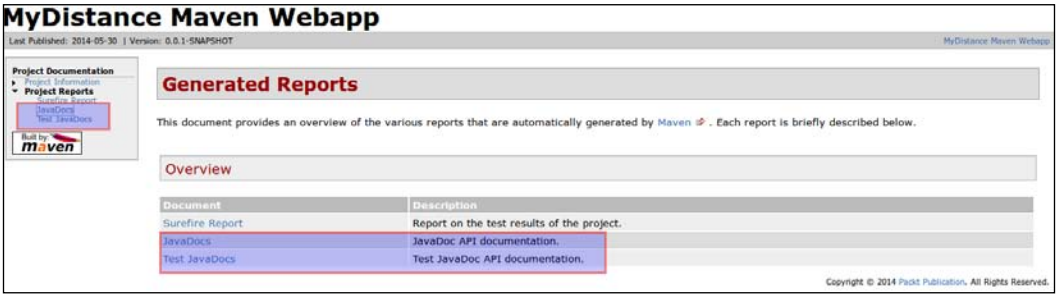
Generating javadocs

We often need to generate API documentation of our code base. Having an API documentation increases collaboration, understanding, migration, and the transfer of knowledge becomes handy. To generate javadocs, add the following dependency in the `reporting` element as follows:

```
<reporting>
    <plugins>
  ….. .
  <!-- For Javadoc generation-->
        <plugin>
```

```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>2.9.1</version>
          </plugin>
    </plugins>
  </reporting>
```

After making the preceding changes to the `pom` file, run the `site` phase from the previous section. The APIs will be generated in the `apidocs` and `testapidocs` folders under `target/site`. This can be navigated in the `index.html` file, under the **Project Reports** head with **JavaDocs** and the **Test JavaDocs** label, along with **Surefire-Reports** as shown in the following screenshot:



Summary

Well done! We have developed `MyDistance`, a distance conversion utility web application. During the course of development, we learned adding dependencies, writing unit tests, executing them, generating site documentation, and generating javadocs for them. In the next chapter, we will learn about multimodule projects with Maven.

6
Creating a Multimodule Project

Now that we have already launched the rocket, let's explore more of it. In this chapter, we will develop a `MyDistance` application from the previous chapter as a multimodule Maven project and learn how to create multimodule projects, build, and run them. The topics covered in this chapter are as follows:

- Introduction
- Creating a parent project—POM
- Creating a core module
- Creating a webapp module
- Building a multimodule project

Introduction

Software architecture states modularity as the degree to which a system's components may be separated and recombined. In software engineering, modularity refers to the extent to which a software/application can be divided into multiple modules to achieve the business goal. Modularity enhances manageability and reusability. The growing days has seen software getting more complex, and modularity is the need of the hour.

Multimodule projects consist of many modules that adapt to modularity. A multimodule project is identified by a parent/master POM referencing one or more `.sub` modules.

A multimodule project constitutes of the following:

- **Parent project POM**: This glues and references all the other modules of a project
- **Modules**: This includes submodules that serve different functions of the application and constitute the application

Parent POM is where you can put common dependencies in a single place and let other modules inherit it, that is, POM inheritance in modules. Usually, universal dependencies such as JUnit or log4j are the candidates of POM inheritance.

The mechanism by which Maven handles multimodule projects is referred to as **reactor**. The reactor of Maven's core has the following functions:

- Collects all the modules to build
- Sorts the projects (modules) into the current build order
- Builds the sorted projects in order

The modules of the project are enclosed inside the `<modules> </modules>` tag by specifying each module with the `<module> </module>` tag. Similarly, the parents are enclosed inside the `<parent> </parent>` tag by specifying Maven coordinates.

Now, for illustration, we will take the `MyDistance` application from *Chapter 5*, *Spicing Up a Maven Project*, and develop it as a multimodule project. The modules of the project would be as follows:

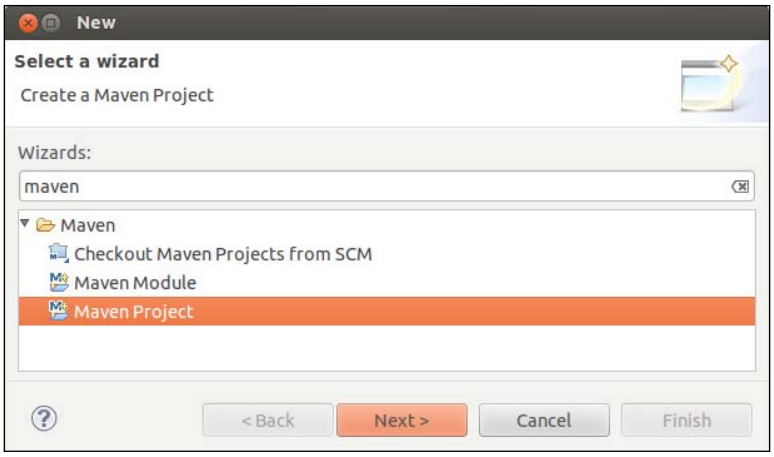
- `Distance-main`: This is the parent project, also known as parent POM, that glues and references different modules of the project, that is, `distance-core` and `distance-webapp`
- `distance-core`: This module provides a simple distance conversion utility class
- `distance-webapp`: This is a web interface in which you can input the units that depend on the `distance-core` module to perform a conversion and respond to the results

In the subsequent sections, we will get into the details of the preceding modules.

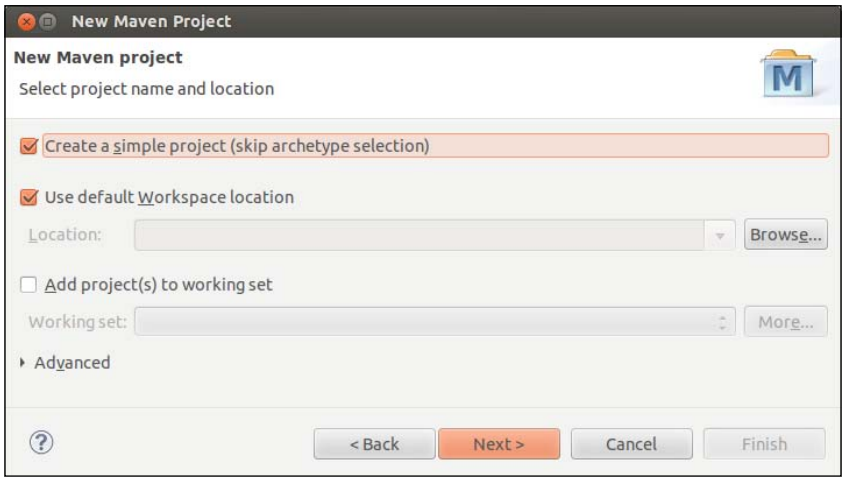
Creating a parent project – POM

The first step towards building a multimodule project is setting up a parent POM. To do this, follow the ensuing steps:

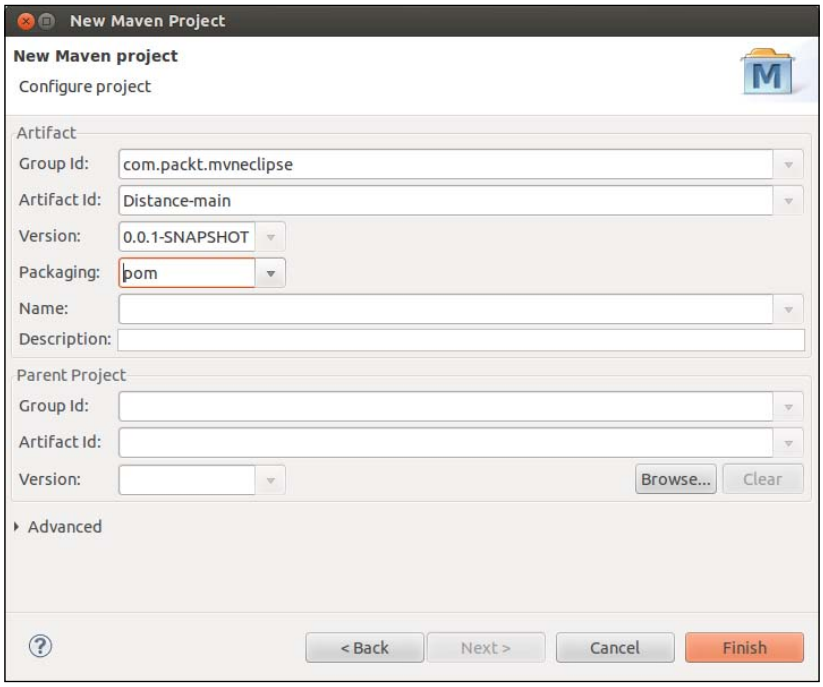
1. Navigate to **File | New** and click on **Maven Project**. Alternatively, navigate to **File | New** and click on **Other...**. In the **Select a wizard** screen, search for `maven` via the search box, select **Maven Project**, and click on the **Next** button, as shown in the following screenshot:



2. The **New Maven Project** wizard appears; make sure that you tick the checkbox **Create a simple project (skip archetype selection)**, as shown in the following screenshot, and click on **Next**:



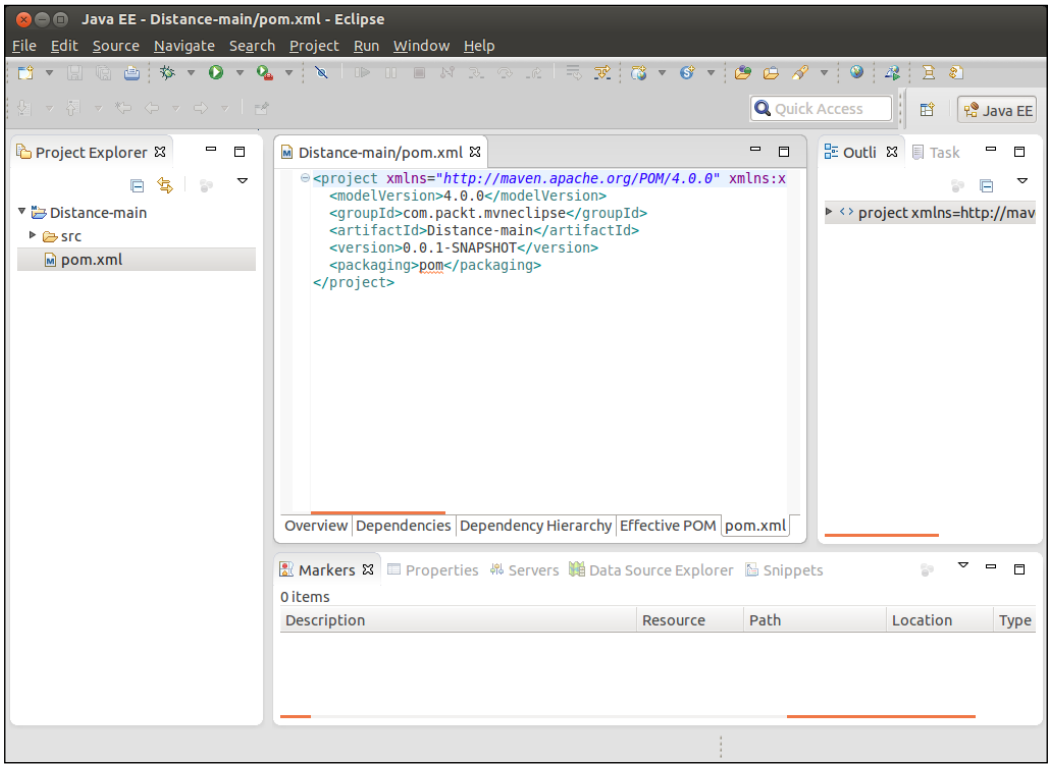
3. The **New Maven project** configuration wizard appears; fill in the details as shown in the screenshot and click on **Finish**:



Make sure that you choose the packaging as POM from the dropdown:

Field	Value
Group Id	`com.packt.mvneclipse`
Artifact Id	`Distance-main`
Version	`0.0.1-SNAPSHOT`
Packaging	`pom`

4. The corresponding project will get created, and the resulting screen would look as follows:

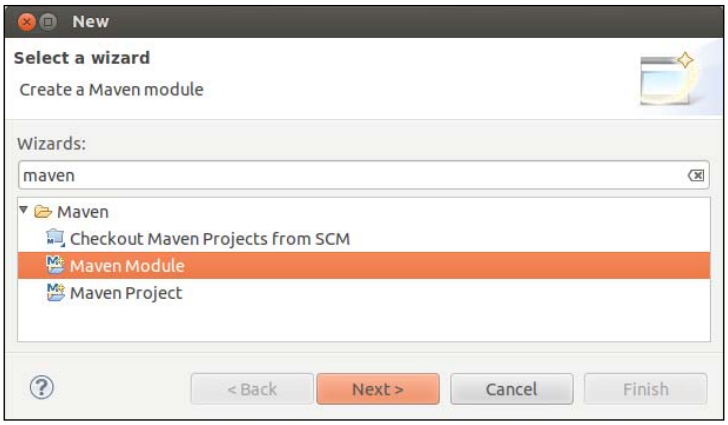


We have the parent POM in place now.

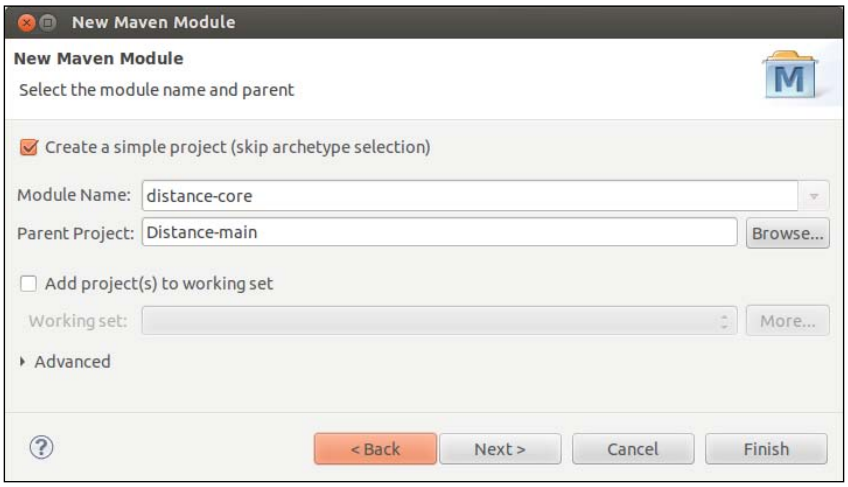
Creating a core module

The core module of `MyDistance` will contain a class that can be converted across different units, that is, from km/meter to yard/miles. Let's name this core module `distance-core`. To create a core module, perform the following steps:

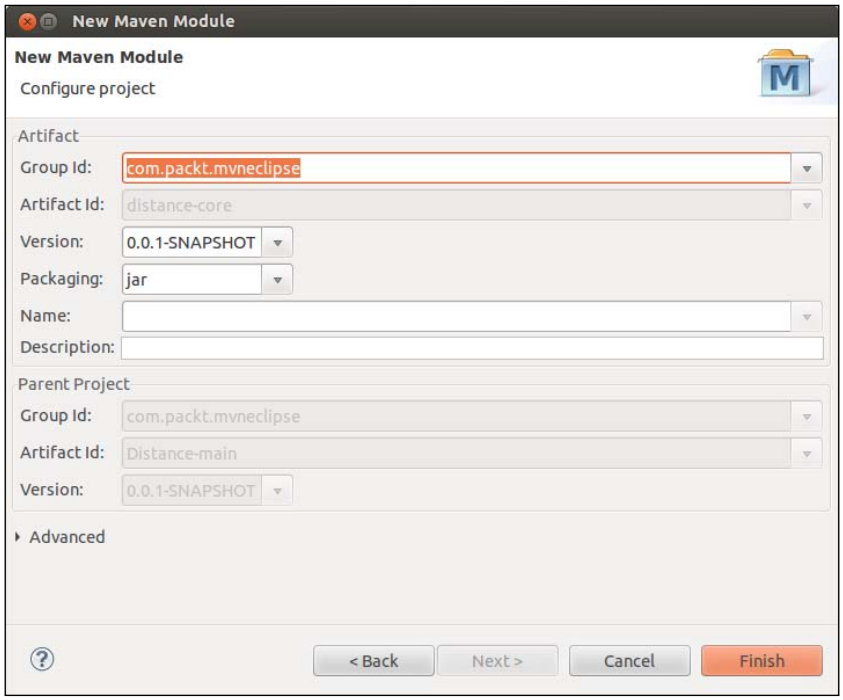
1. Navigate to **File | New** and click on **Other…**. In the **Select a wizard** screen, search for `maven` via the search box, select **Maven Module**, and click on the **Next** button, as shown in the following screenshot:



2. The **New Maven Module** wizard appears; make sure to tick the checkbox **Create a simple project (skip archetype selection)**, provide the module name as `distance-core`, and browse to select the parent as `Distance-main`, as shown in the following screenshot:



3. In the **Configure project** Maven module wizard, fill in the details provided in the table after the following screenshot and click on **Finish**:



Field	Value
Group Id	`com.packt.mvneclipse`
Version	`0.0.1-SNAPSHOT`
Packaging	`jar`

Since the core module just contains a Java class and is available to be used as a library for a web module of an application, the packaging type is `jar`.

4. The `distance-core` module gets created and the contents of the POM will look as follows:

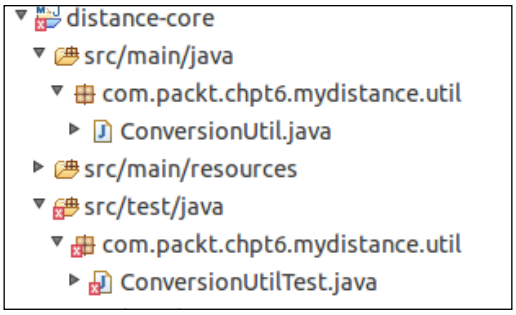
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
    <parent>
    <groupId>com.packt.mvneclipse</groupId>
        <artifactId>Distance-main</artifactId>
```

```
    <version>0.0.1-SNAPSHOT</version>
</parent>
<artifactId>distance-core</artifactId>
</project>
```

Please note that the parent of the module is enclosed in the `<parent></parent>` tag. Also, the `groupId` and `version` tags will not be present for a module since we specified the same `groupId` and `version` as the parent in the **Configure Maven Module** wizard. We did this in the preceding code while creating a module.

At any point, if we wish to change or add `groupId/version/artifactId`, we can always edit the `pom.xml` file since it's an XML file.

5. The core module consists of a class file that performs the conversion across distance units. Now let's add a class; right-click on the project, navigate to **New**, select **Package**, and specify the package name as `com.packt.chpt6.mydistance.util`.
6. Create a class named `ConversionUtil` in the preceding package. If you remember, we created this class in *Chapter 5*, *Spicing Up a Maven Project*. So, copy the contents of this class and save it.
7. Now let's put a unit test class in place. Create a package, `com.packt.chpt6.mydistance.util`, in `src/test/java`. Add the class `ConversionUtilTest` to the corresponding package. Refer to *Chapter 5*, *Spicing Up a Maven Project*, where we created this test class; copy the contents of this class and save it. The resulting `src` structure will look as follows:



You might notice that we have some errors, and the errors are due to the `TestCase` class not being resolved. To solve this error, add `jUnit` as a dependency to the parent module, the `pom.xml` file, as shown in the following code:

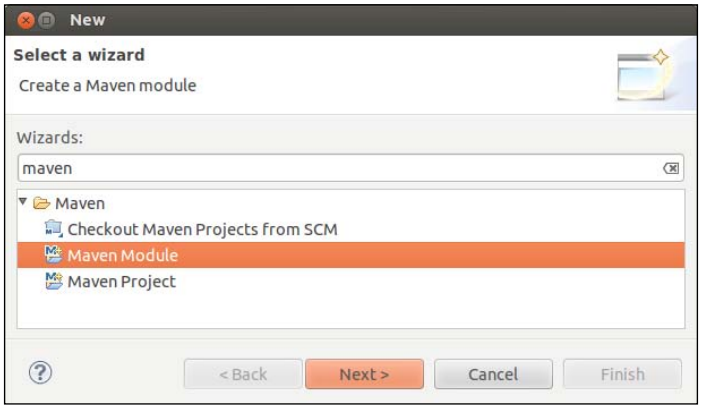
```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>
```

Usually, JUnit and log4j dependencies, that is, common dependencies across modules, are put in one place in the parent POM and the modules inherit them.

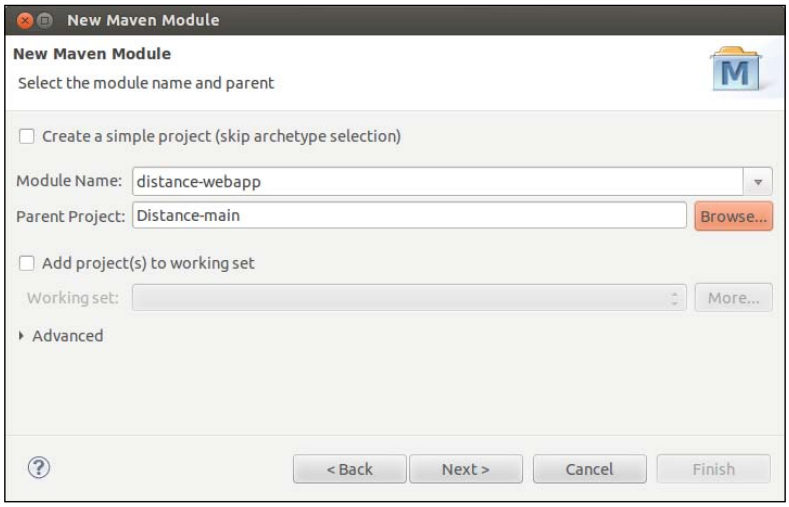
Creating a webapp module

The webapp module has a JSP file that has a form to accept the input. It also has a servlet that accepts the request parameters and performs the conversion using a core module and provides the response. Now let's see how to get the webapp module in place by performing the following steps:

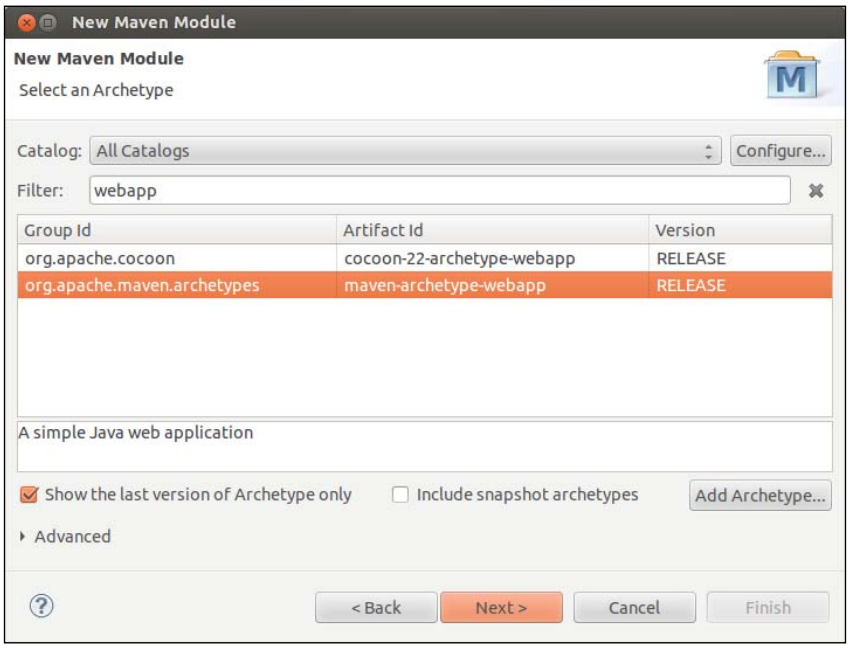
1. Navigate to **File** | **New** and click on **Other...**; in the **Select a wizard** screen, search for `maven` via the search box, select **Maven Module**, and click on the **Next** button as shown in the following screenshot:



2. In the **New Maven Module** window that will follow, provide the module name as `distance-webapp` and browse to select the parent as `Distance-main`.

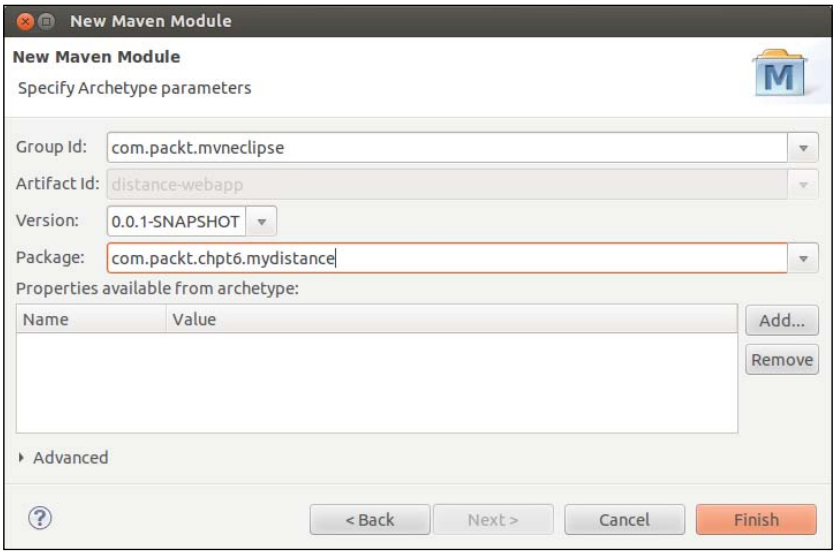


3. In the **Select an Archetype** window, search for `webapp` via the search box, select **maven-archetype-webapp**, and click on **Next** to proceed, as shown in the following screenshot:

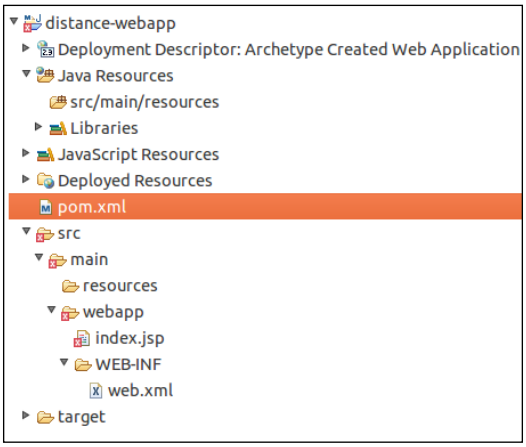


4. In the **New Maven module** window, fill in the details provided in the following table and click on **Finish** as shown in the screenshot that follows this table:

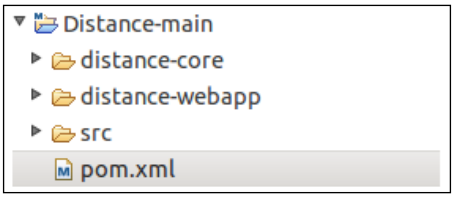
Field	Value
Group Id	`com.packt.mvneclipse`
Version	`0.0.1-SNAPSHOT`
Package	`com.packt.chpt6.mydistance`



5. The webapp module will be created, and the resulting structure will look like the following screenshot:



6. Now if we look at the parent project's structure, we might notice that it has a reference to each of the modules, as shown in the following screenshot:



7. Also, if we take note of the `pom.xml` file of the parent project, we will see how modules are being added to the `<module>` tag as follows:

```
<modules>
  <module>distance-core</module>
  <module>distance-webapp</module>
</modules>
```

8. Open the webapp module's `pom.xml` file and add the dependencies for `log4j`, servlet, and Tomcat, as shown in the following code; this is also discussed in *Chapter 5*, *Spicing Up a Maven Project*, in more detail:

```
    <!-- Include servlet  API -->
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>
  <!-- For logging purpose could be put in parent POM for modules
to inherit  -->
    <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
  </dependency>

<!--  For tomcat
<plugins>
    <plugin>
          <groupId>org.apache.tomcat.maven</groupId>
          <artifactId>tomcat7-maven-plugin</artifactId>
          <version>2.1</version>
    </plugin>
    </plugins>
```

9. Also, add `distance-core` as a dependency since it is used by the webapp to perform the conversion, as shown in the following code:

```
<dependency>
<groupId>com.packt.mvneclipse</groupId>
  <artifactId>distance-core</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <scope>compile</scope>
</dependency>
```

10. Add the `log4j.properties` file to the `resources` folder. Refer to the *Adding Resources* section in *Chapter 5*, *Spicing Up a Maven Project*.
11. Add the form to get input and add servlets (refer to sections *Adding a form for getting input* and *Adding Servlet* of *Chapter 5*, *Spicing Up a Maven Project*).

Building a multimodule project

Now that we are done with writing the code for modules, let's build the project. Right-click on the parent project—in this case, `Distance-main`—select **Run As**, and click on **Maven test**. This should compile and run the unit tests. An excerpt of the output in the console is as follows:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
------
[INFO] Reactor Build Order:
[INFO]
[INFO] Distance-main
[INFO] distance-core
[INFO] distance-webapp Maven Webapp
[INFO]
[INFO] Using the builder org.apache.maven.lifecycle.internal.builder.
singlethreaded.SingleThreadedBuilder with a thread count of 1
[INFO]
[INFO] ------------------------------------------------------------------
------
[INFO] Building Distance-main 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------
[INFO] ------------------------------------------------------------------
------
[INFO] Building distance-core 0.0.1-SNAPSHOT
```

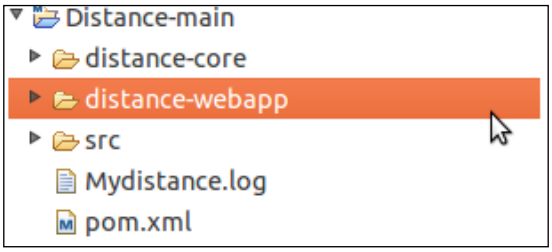
```
[INFO] -------------------------------------------------------
-------------------------------------------------
 T E S T S
-------------------------------------------------
Running com.packt.chpt6.mydistance.util.ConversionUtilTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.009 sec

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] ------------------------------------------------------------------
------
[INFO] Building distance-webapp Maven Webapp 0.0.1-SNAPSHOT
[INFO] -------------------------------------------------------
[INFO] ------------------------------------------------------------------
------
[INFO] Reactor Summary:
[INFO]
[INFO] Distance-main ..................................... SUCCESS [
0.002 s]
[INFO] distance-core ..................................... SUCCESS [
2.250 s]
[INFO] distance-webapp Maven Webapp ...................... SUCCESS [
0.161 s]
[INFO] ------------------------------------------------------------------
------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
------
```

The mechanism referred to as a reactor knows the order of building a project. Now again, right-click on **Distance-main**, select **Run As**, and click on **Maven install** to install the modules in a local repository.

Always make sure to clean the project by running Maven `clean` via the **Run As** option if any changes occur; alternatively, you can reinstall the project using **Maven install**.

Running the application

To run the application, right-click on the webapp module of the parent project highlighted in the following screenshot, select **Run As**, and then click on **Run Configurations...**. In the **Run configuration** window, specify the goal as `tomcat7:run` and click on the **Run** button. The web application will run at `http://localhost:8080/distance-webapp/`; point the browser to this location and perform the conversion:



Summary

In this chapter, we learned how to create a multimodule project and then build and run the application. In the next chapter, we will take a sneak peek into m2eclipse and learn how to customize it.

7
Peeking into m2eclipse

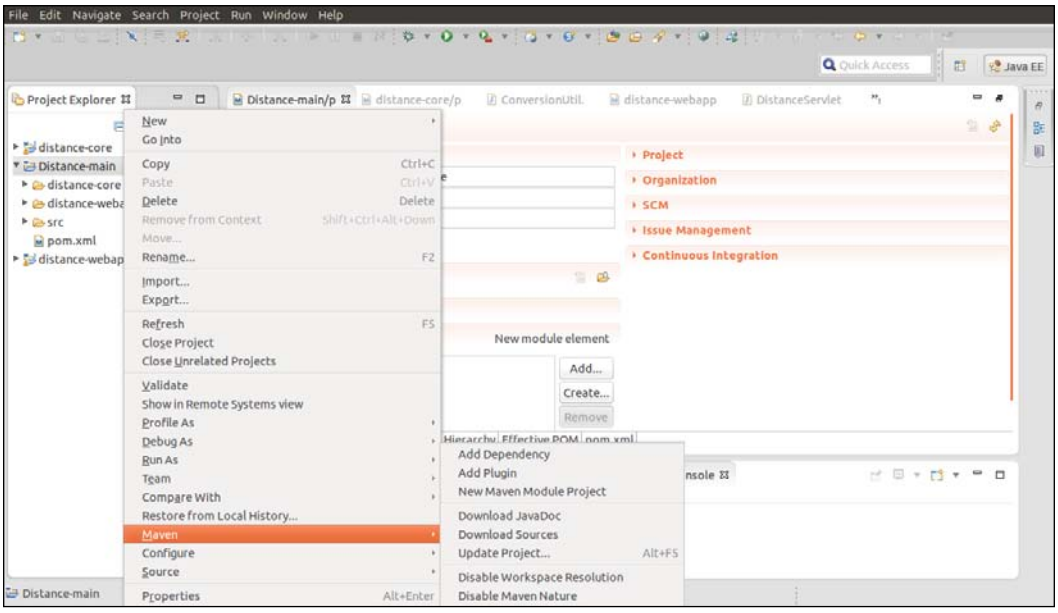
We are toward the end of the journey; now in this chapter, we will look into other additional features in m2eclipse, getting familiar with the form-based POM editor, and learn about repositories.

The topics covered in this chapter are as follows:

- Other features in m2eclipse
- A form-based POM editor
- Analyzing project dependencies
- Working with repositories
- m2eclipse preferences

Other features in m2eclipse

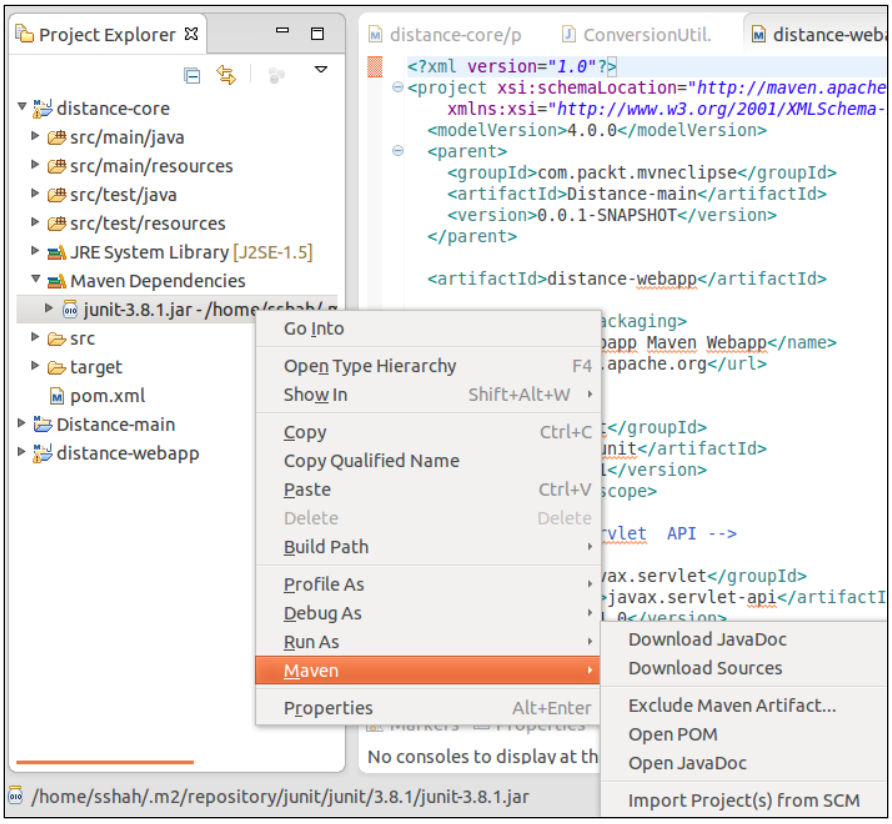
The following steps have to be performed in order to understand the other features of m2eclipse. Right-click on the Maven project and navigate to the **Maven** menu item. Then, you can see the available features as shown in the following screenshot:



If you see the screenshot, the available features are as follows:

- **Add Dependency**
- **Add Plugin**
- **New Maven Module Project**
- **Download JavaDoc**
- **Download Sources**
- **Update Project**
- **Disable Workspace Resolution**
- **Disable Maven Nature**

Similarly, right-click on **Maven Dependencies** and navigate to the **Maven** menu item. The available features seen are shown in the following screenshot:



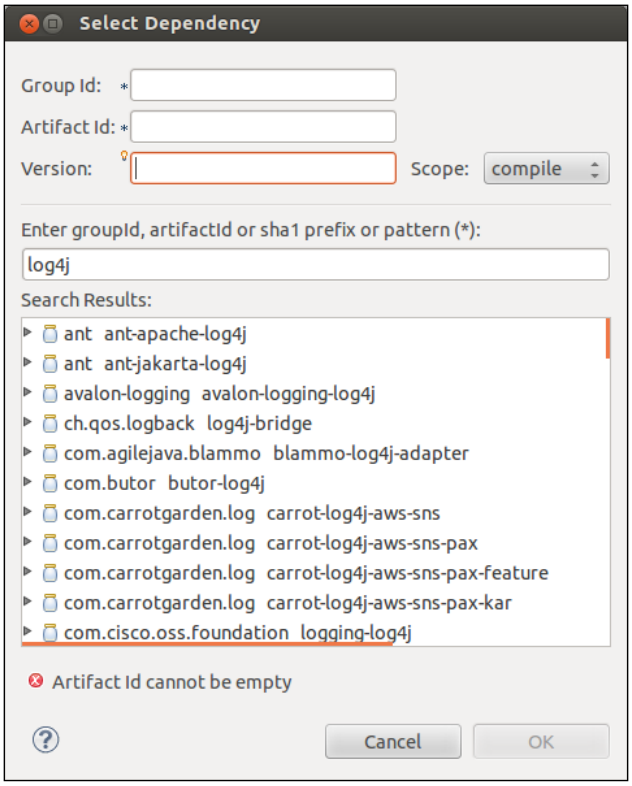
The available features are as follows:

- **Download JavaDoc**
- **Download Sources**
- **Exclude Maven Artifact**
- **Open POM**
- **Open JavaDoc**
- **Import Project(s) from SCM**

In the sections to follow, we will collectively discuss these features.

Add Dependency

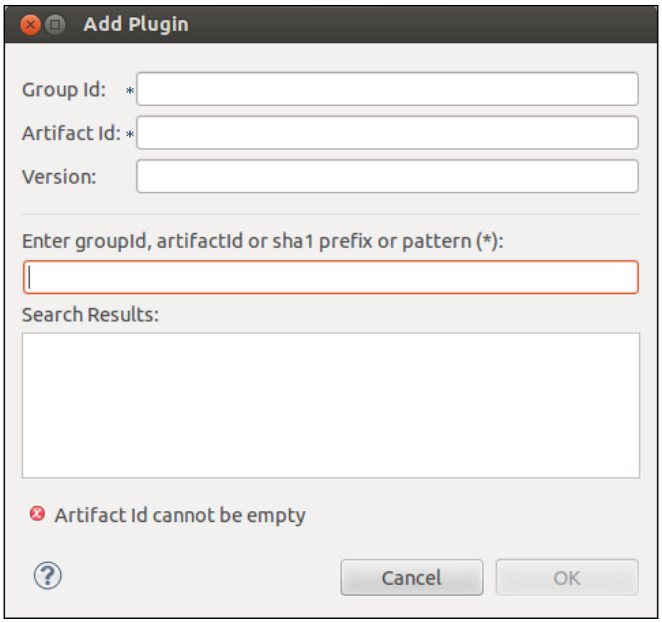
It allows us to add dependencies to the Maven project. The screenshot for this is shown as follows:



Up until now, we have been editing the `pom.xml` file and adding dependencies to it. Adding dependencies is another way to achieve the same objective using the graphical interface. When you use this option, the information you need to know is less, that is, knowing `artifactId`/`groupId` is enough to search across repositories and select the appropriate one. In the previous method, you need to know complete Maven coordinates to add the dependencies; hence, the latter is a time saver.

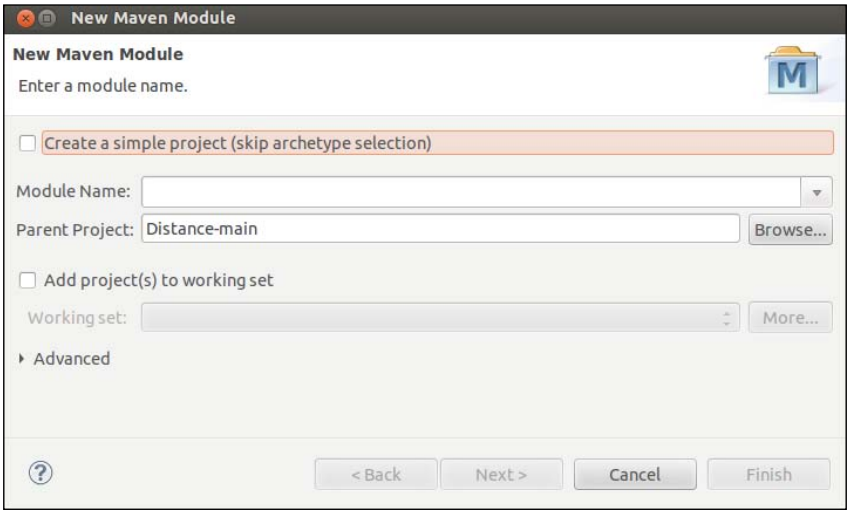
Add Plugin

Similar to **Add Dependency**, **Add Plugin** allows you to add plugins via the graphical interface. This requires us to have minimal information to search through the repository and add plugins. The screenshot for this is as follows:



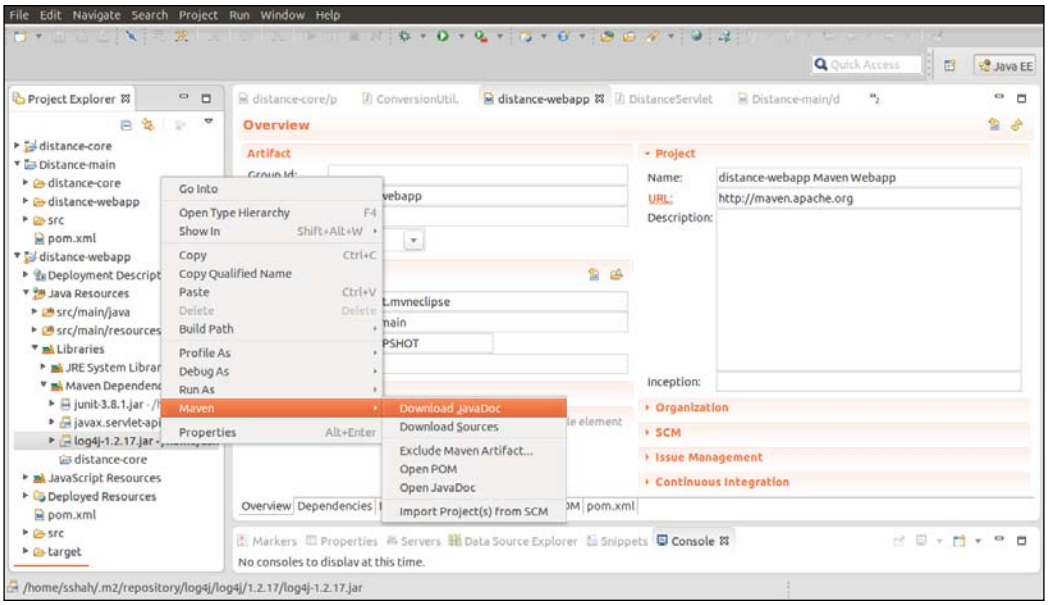
New Maven Module Project

In *Chapter 6, Creating a Multimodule Project*, we created a multimodule project and learned about creating module projects. This is another way to invoke the same **Add module** wizard to create modules. The screenshot of the window is shown as follows, which is the same as the screenshot obtained when you navigate to **New** | **Other** | **Maven Module** and right-click on the project (as we saw in the previous chapter):



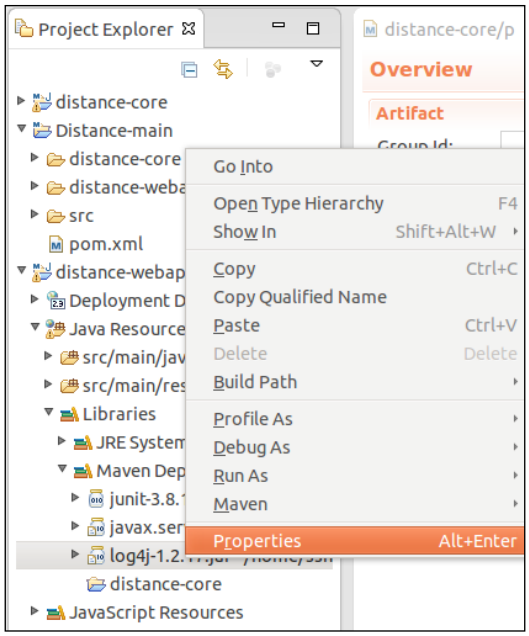
Download JavaDoc

Download JavaDoc is used to download the javadoc of the project if present in the central repository to the local repository. For example, right-click on the `log4j-1.2.17.jar` file under **Maven Dependencies** and click on **Download JavaDoc**, as shown in the following screenshot. The javadoc will be downloaded to the local repository along with other artifacts at the `$HOME/.m2/repository/log4j/log4j/1.2.17/ as log4j-1.2.17-javadoc.jar` location:

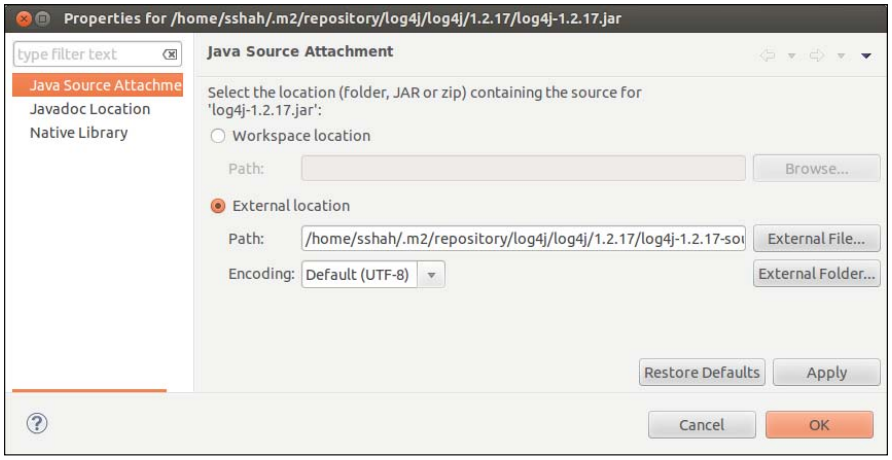


Download Source

If the central repository has a source artifact for the corresponding project, we can use this option to download it to the local repository and use it with the Eclipse environment. This is a life saver, particularly when we are debugging a complex issue and there is need to drill down the code of dependencies. For example, the source code of `log4j` will be downloaded at the `$HOME/.m2/repository/log4j/log4j/1.2.17/` as `log4j-1.2.17-javadoc.jar` location. After downloading the source code, right-click on the `log4j-1.2.17.jar` file and click on **Properties**, as shown in the following screenshot:



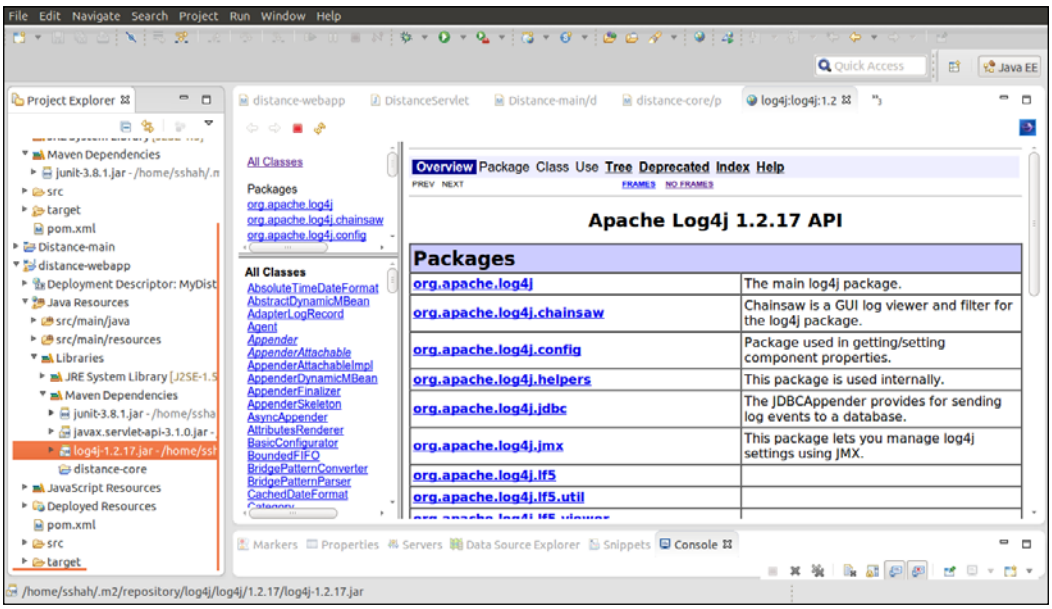
The **Properties** window appears; the **Java Source Attachment** navigation button shows the attached source code location, as shown in the following screenshot:



Note that we can also find the javadoc location by clicking on the **Javadoc Location** navigation button on the left pane.

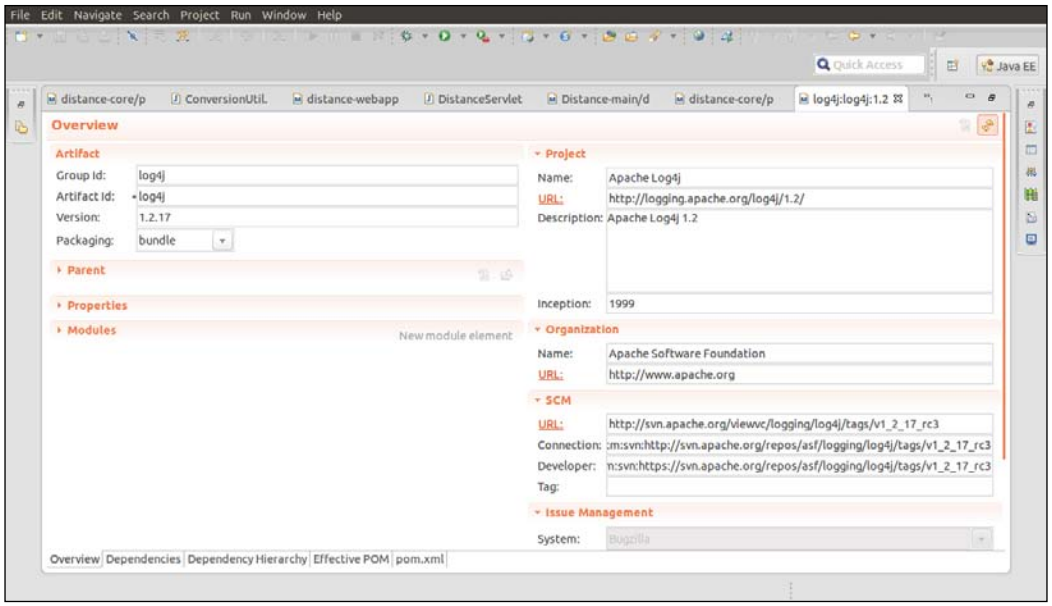
Open Javadoc

When we want to browse through the javadoc of the corresponding project, we use this option. The javadoc opens in the editor area as a separate tab as shown in the following screenshot for `log4j` docs:



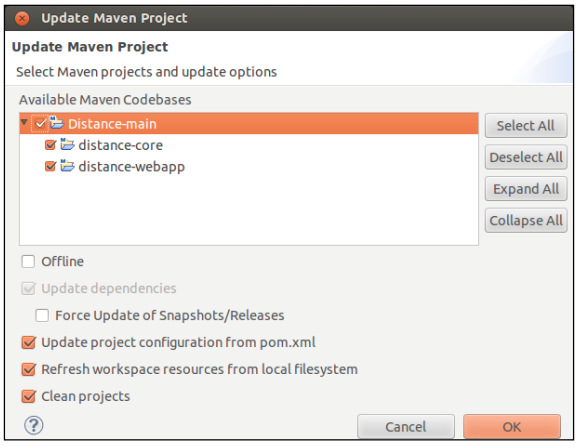
Open POM

At any point, if there is a need to look at the POM file of the dependencies, we can use this option. The respective POM file of the dependency opens in the editor area of the workspace. The following screenshot depicts the POM file in the editor area for `log4j`, which has been the dependency in the `MyDistance` application we built in *Chapter 4, Building and Running a Project*:



Update Project

There are instances where we have a Java project and we want to convert it to a Maven project. We can do this by right-clicking on the project, navigating to **Configure | Convert to Maven Project**, and adding a POM file. Now, **Update Project** is used to update the project from its dependencies and resources. **Update Project** is also handy if you have multiple Maven projects in your workspace, and the projects depend on each other. Then, after you build (`mvn install`) one project, you can perform **Update Project** on other projects to pick up the new artifact. The update option is shown in the following screenshot:



Choosing **Offline** will not check the central repository for updates.

Disable Workspace Resolution

Imagine a case where a project A depends on project B and they reside in the same workspace, mostly in multimodule projects. Now, if workspace resolution is disabled, to have project A, a successful build and a project B artifact are needed in the local repository. However, if the workspace resolution is enabled, the dependencies are resolved using an Eclipse workspace, and there is no need for an artifact in the local repository.

Disable Maven Nature

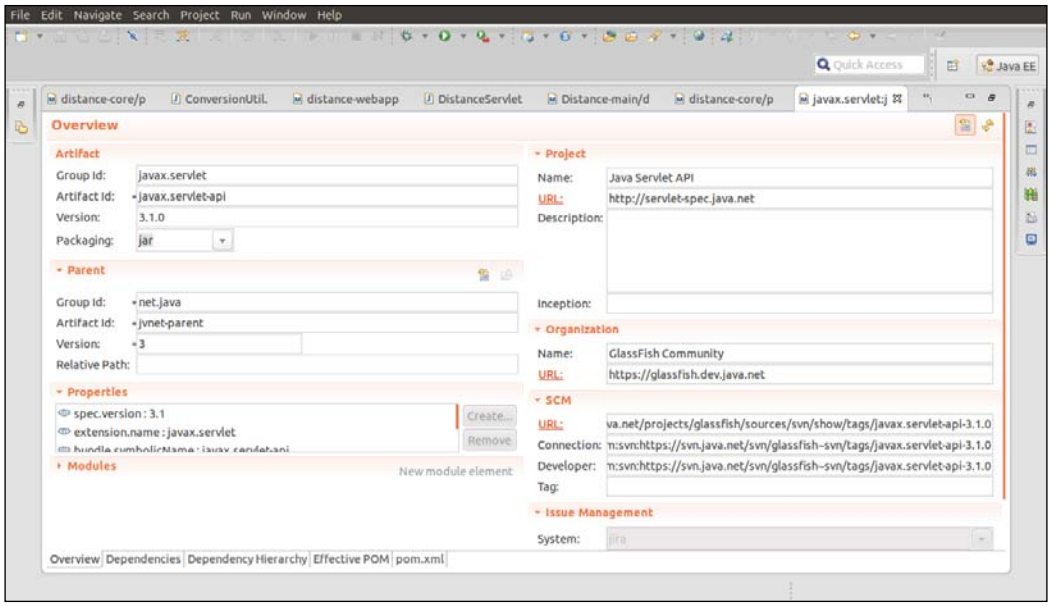
It disables the Maven aspect of the project, that is, the Maven dependencies are removed from the build path. In that case, we may need to include the JARs manually in the classpath from the build window.

Import Project(s) from SCM

It allows you to pull the source code to your Eclipse workspace for the dependency. In other words, it enables you to create a new Maven project for a dependency based on the sources for that dependency. The sources are pulled from the **Source Code Management** (**SCM**) system as defined in the POM for the dependency. If the dependency POM fails to mention the SCM, then this option does nothing. Make sure you have a proper m2e connector installed for the corresponding SCMs. We discussed importing and installing Subversion m2e connector in *Chapter 3*, *Creating and Importing Projects*.

A form-based POM editor

m2eclipse provides the option of editing the `pom` file using a form-based POM editor. In earlier chapters, we played with XML tags and edited the `pom` file. While directly editing an XML file, the knowledge of tags is required, and there is a high chance that the user will make some errors. However, a form-based editor reduces the chance of a simple error and eases the editing of a `pom` file without or very minimal XML knowledge behind the scene. I would prefer playing around with XML tags and use that option, but you are open to choose your option. The form-based editor is shown in the following screenshot and has five tabs: **Overview**, **Dependencies**, **Dependency Hierarchy**, **Effective POM**, and **pom.xml**:



An overview

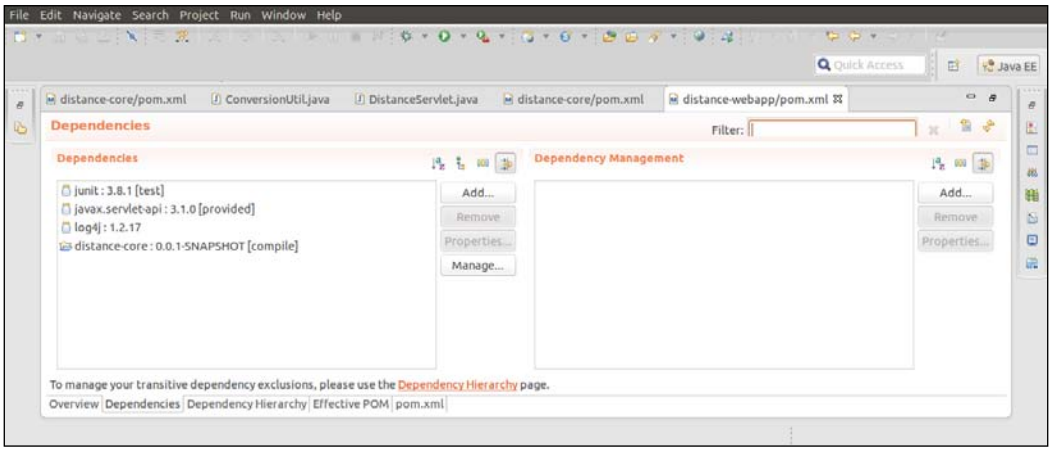
Overview provides general information of the project. It consists of the following sections and provides information about them as shown in the preceding screenshot:

- **Artifact**
- **Parent**
- **Project**
- **Modules**
- **Properties**
- **Organization**
- **SCM**
- **Issue Management**
- **Continuous Integration**

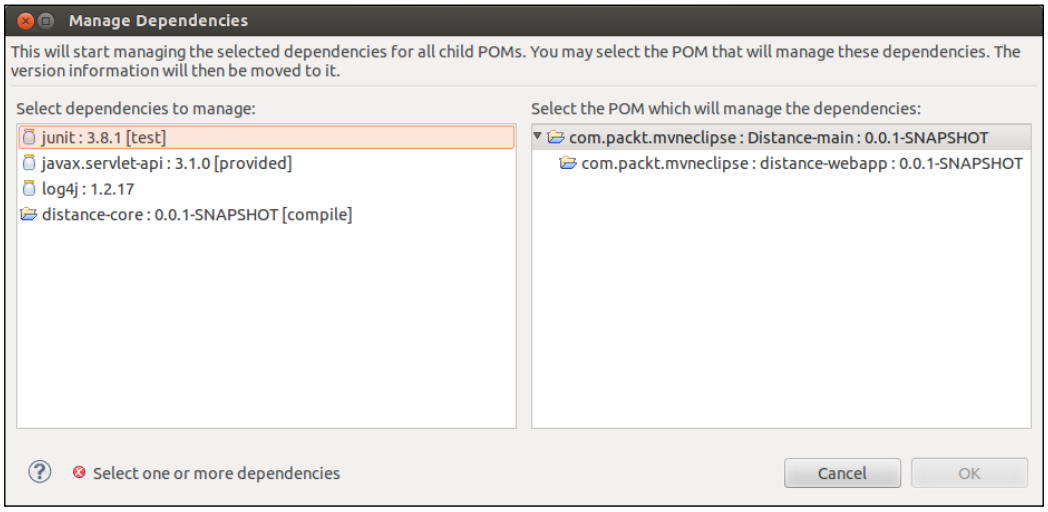
You can change any information in this form, and this will be reflected in the XML file. We will discuss the **Dependencies** and the **Dependencies Hierarchy** tab in the next section.

Analyzing project dependencies

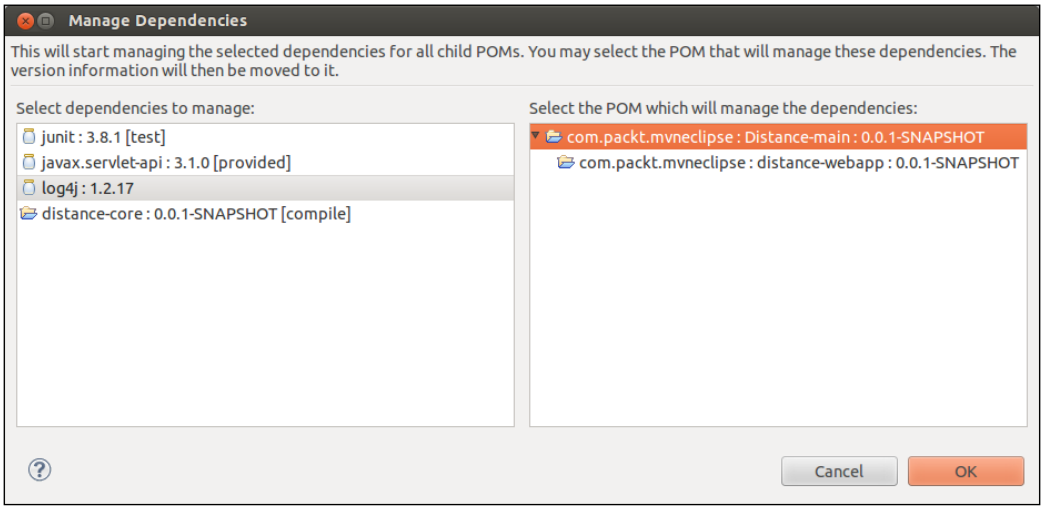
A POM editor has a **Dependencies** tab that provides a glance of dependencies and an option to manage dependencies of the project. The **Dependencies** tab has two sections as shown in the following screenshot:



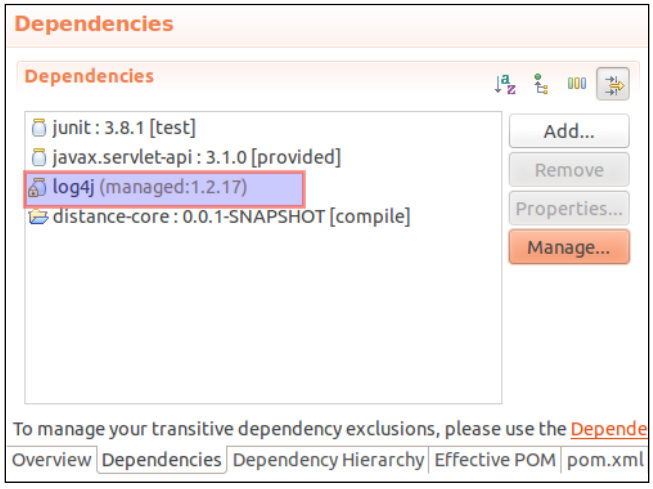
It shows all the dependencies of the project on the left side. We can also add dependencies to the project using the **Add** button of the **Dependencies** section. The **Manage** button allows you to choose the POM that will manage the corresponding dependencies, and the screenshot for this is shown as follows:



As stated very clearly on the top of the window, the managed dependencies version information will move to the POM that manages it. For example, let's choose to manage the `log4j` dependency of `distance-webapp` by the `Distance-main` POM. Select `log4j` on the list to the and select `Distance-main` in the list to the right, and click on **OK** as shown in the following screenshot:



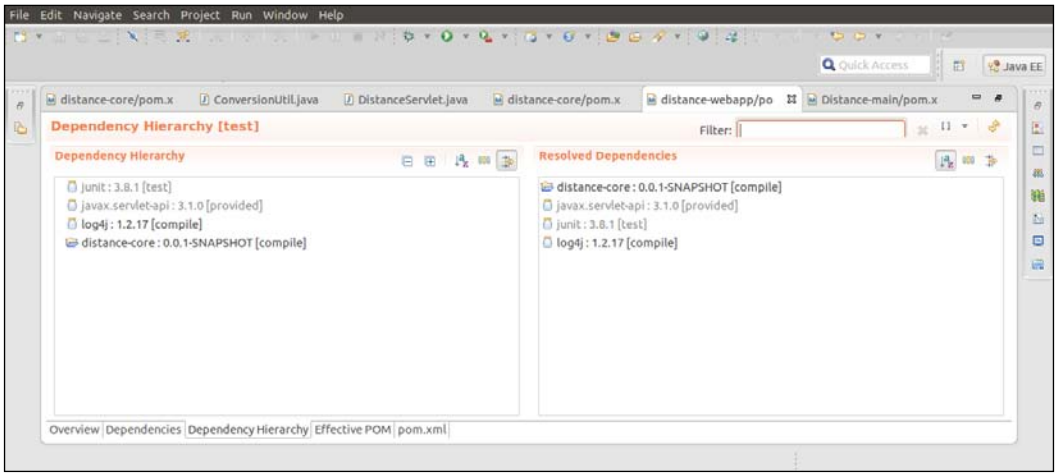
After the `log4j` file is managed, the word "managed" appears to its right, as shown in the following screenshot:



If we happen to see its effect in the XML file, we can see that the version information from the `distance-webapp` POM is moved and is added as a dependency in the `Distance-main` POM, as shown in the following code:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Another way to achieve the same functionality is via the **Dependency Management** section to the right across the parent-child POM. The **Dependency Hierarchy** tab contains two sections: **Dependency Hierarchy** and **Resolved Dependencies** as follows:



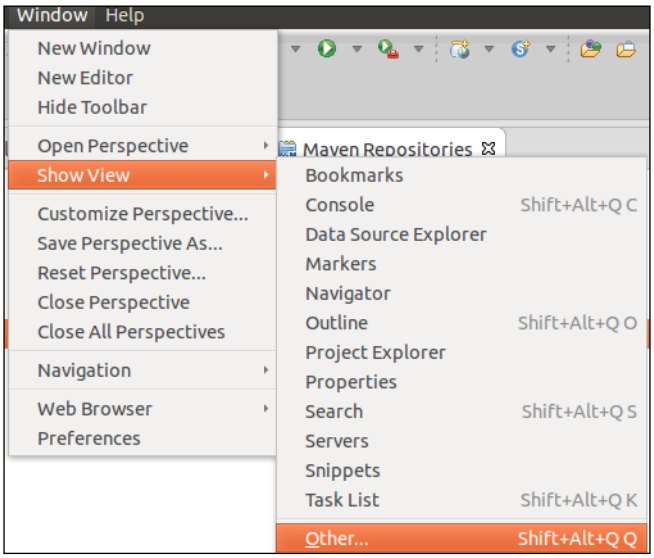
The **Dependency Hierarchy** section on the left provides the tree view of the dependencies. The first level of the tree is direct dependency of the project and then each subsequent level shows the dependencies of each dependency. The preceding screenshot is for the `distance-webapp` module, where we have four direct dependencies, and these dependencies have no further dependency, so the tree structure is not visible. However, for large projects and large direct dependencies, we can easily visualize it. The jar icon indicates that it is referenced from the Maven repository and the open folder icon indicates its presence in the Eclipse workspace.

The **Resolved Dependencies** section on the right shows the list of all resolved dependencies, that is, resulting dependencies after all conflicts and scopes applied. It gives a general idea of resolution chain propagation and route to **Resolved Dependencies**. Click on any resolved dependency and its shows the dependency chain in the **Dependency Hierarchy** section.

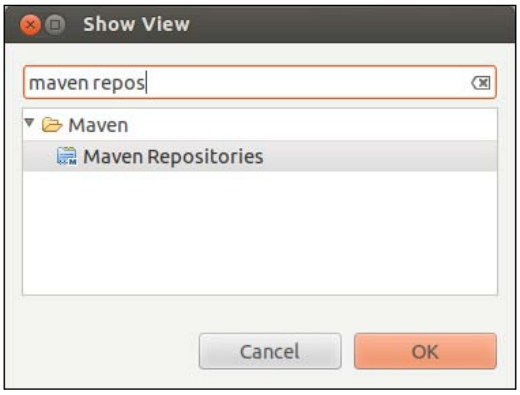
For more information on dependencies, refer to `http://books.sonatype.com/m2eclipse-book/reference/dependencies-sect-analyze-depend.html`.

Working with repositories

To browse through the repository, navigate to **Window** | **Show View** and click on **Other...** as follows:



Next, the **Show View** window appears. Search for `maven repository`, as shown in the following screenshot, and click on **Maven Repositories**:



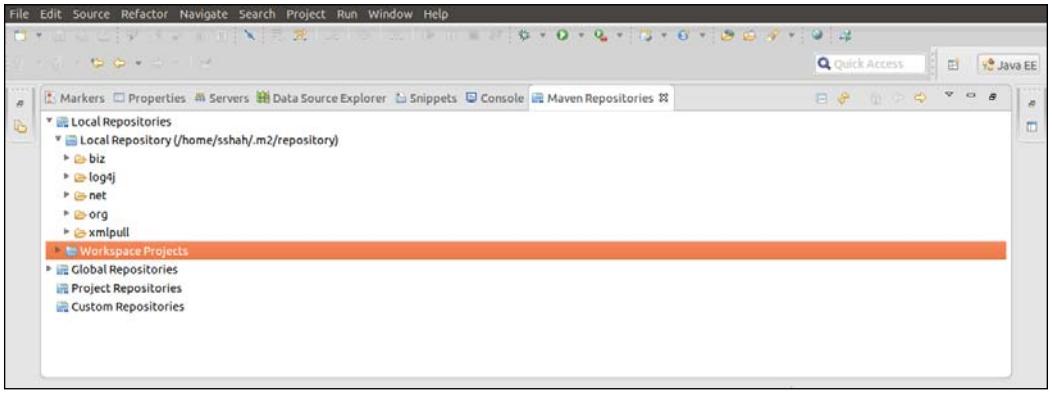
The **Maven Repositories** view constitutes of the following types:

- **Local Repositories**
- **Global Repositories**
- **Project Repositories**
- **Custom Repositories**

The repositories that are of interest are local, global, and project repositories.

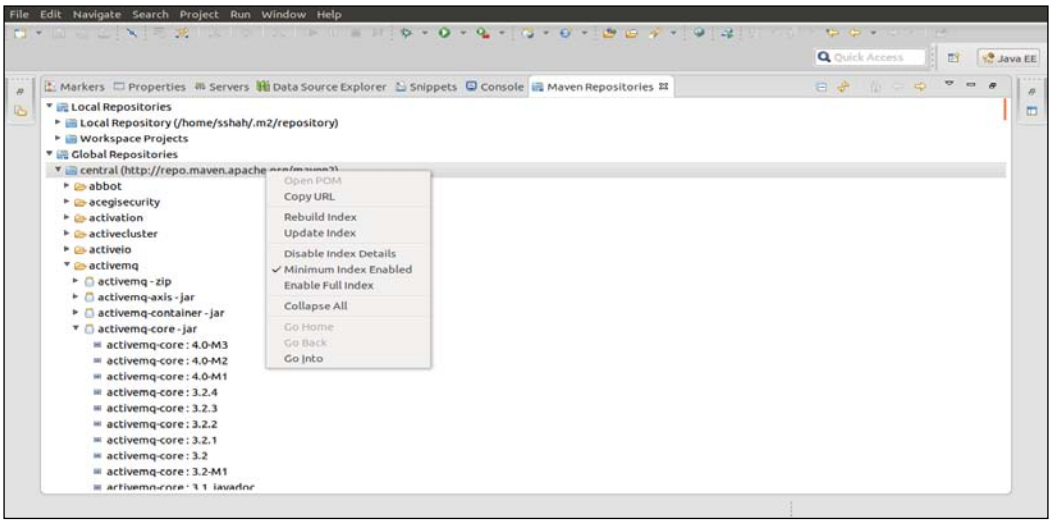
Local Repositories

It shows the artifacts of the local repository, and we can drill down to see its POM contents. It also consists of Eclipse workspace projects. The following is the screenshot of the local repository:



Global Repositories

It references the artifacts of the central repository. We can browse through the artifacts of the central repository and view its POM. Right-click on **Global repositories**, which provides the ability to re-index, build full index, minimum index, and update index from the central repository. The following screenshot illustrates the global repository:

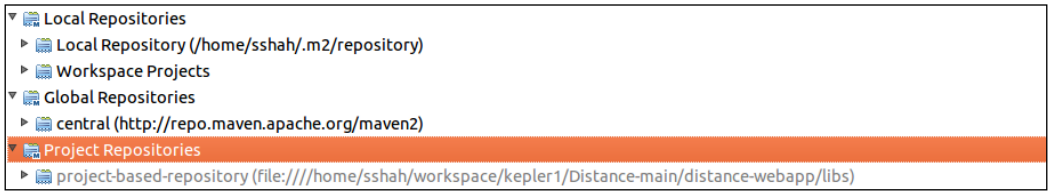


Project Repositories

This repository shows the project-based repositories. Maven discourages the inclusion of repositories mentioned in the project POM file; however, we tend to disobey it when we have to reference local custom artifacts not available in the central repository, or where we may have to package repositories along with distribution. For example, let's take a scenario where we have to reference the artifacts from a `libs` folder in the Eclipse workspace. To achieve this, add the following snippet in the `pom.xml` file:

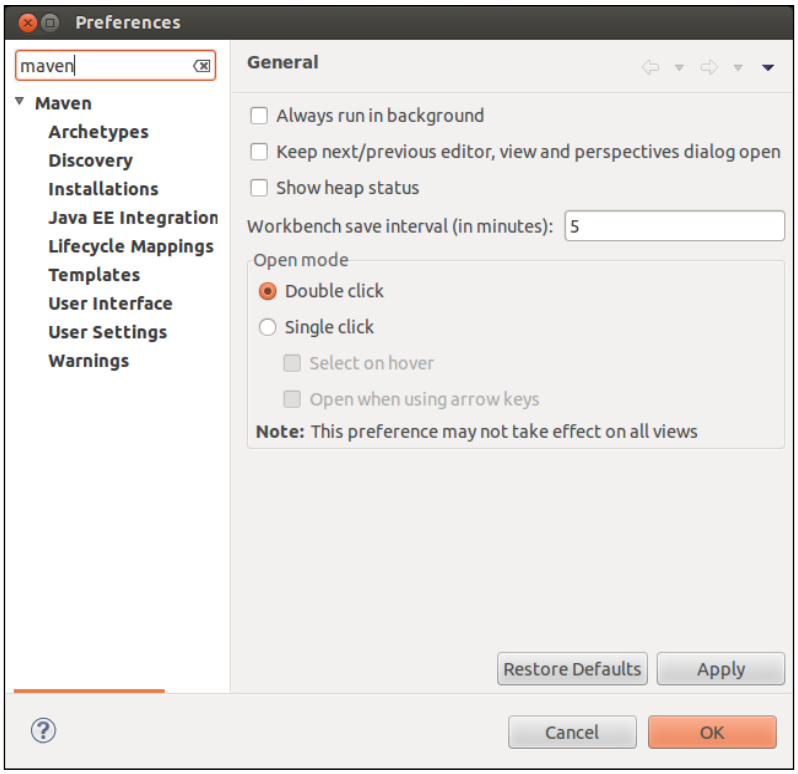
```
<repositories>
    <repository>
      <id>project-based-repository</id>
      <name>Project-specific jars</name>
      <url>file:///${basedir}/libs</url>
    </repository>
<repositories>
```

Refresh the **Maven Repositories** window by clicking on the two cyclic arrows on the top-right side of the window. We can see the corresponding reference in **Project Repositories** as follows:



m2eclipse preferences

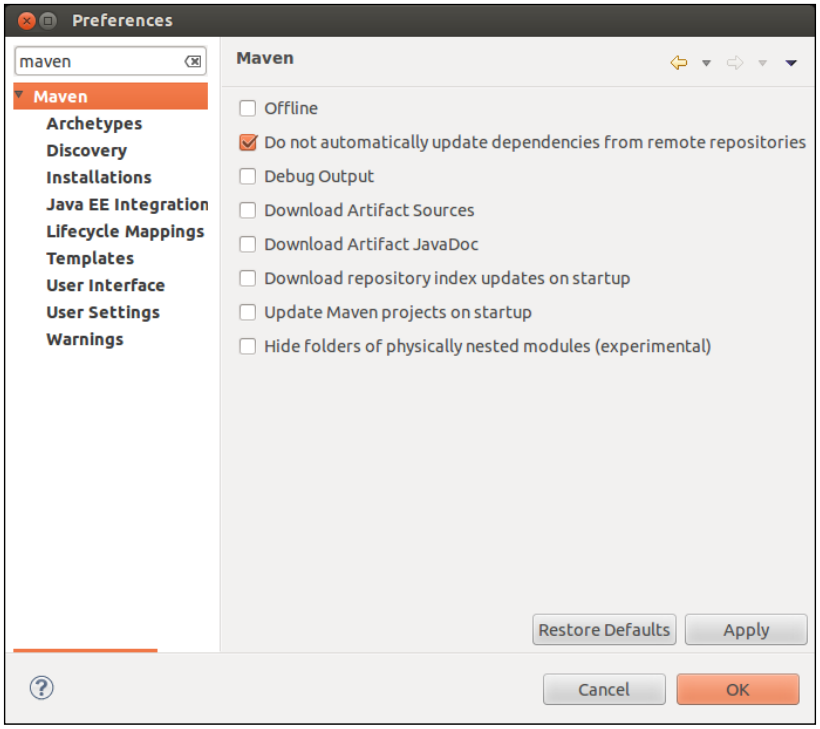
To open m2eclipse preferences, navigate to **Window** | **Preferences**. In the **Preferences** window and search for `maven` in the filter textbox as follows:



Maven

Click on **Maven** as shown in the screenshot that follows later; it allows us to set the following options for Maven:

- **Offline**: This option will not check the central repository for updates
- **Debug Output**: This option sets Maven in the debug mode
- **Download Artifact Sources**: This option downloads sources to local repositories such as JAR
- **Download Artifact Javadoc**: This option downloads the javadoc to the local repository
- **Update Maven projects on startup**: This option updates the dependencies of the Maven project
- **Hide folders of physically nested modules (experimental)**: This option is in the experimental mode, which hides the nested folders of a multimodule project



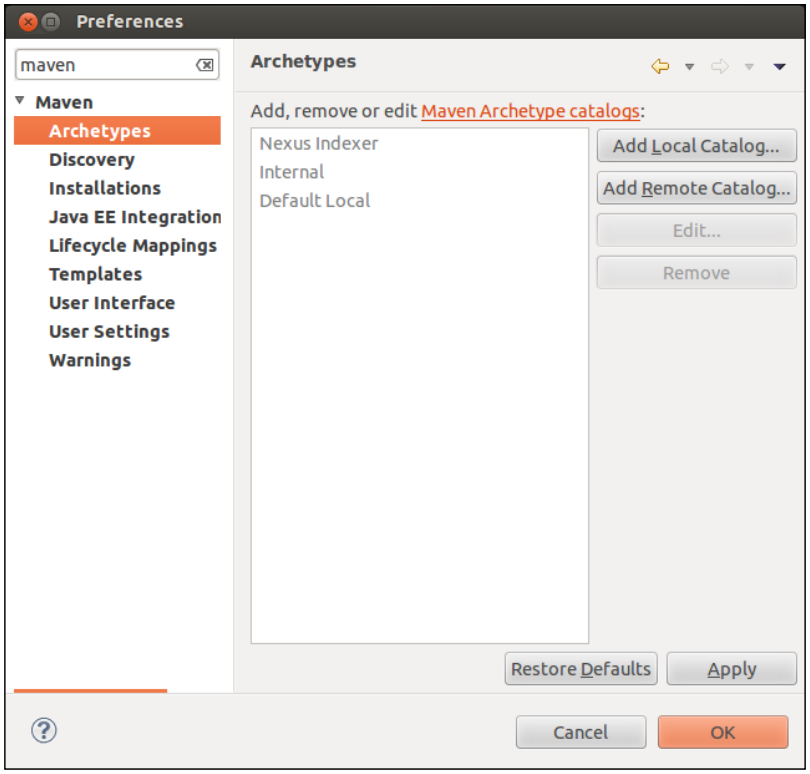
Discovery

Discovery is used to discover the m2e connectors available for use. Please refer to the *Checking out a Maven project* section in *Chapter 3*, *Creating and Importing Projects*, on how we used this feature.

Archetypes

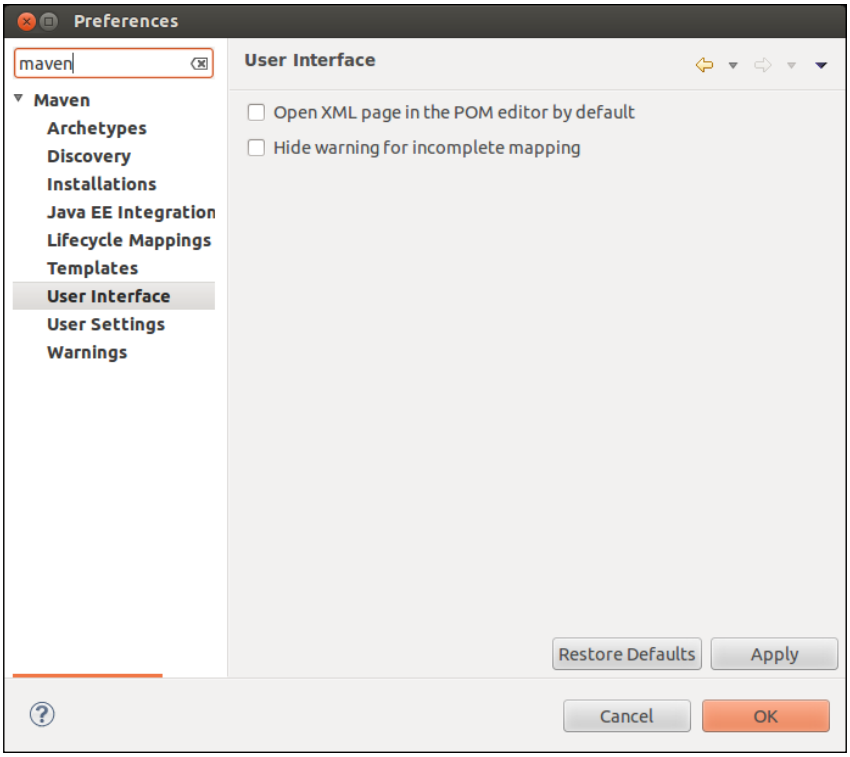
Archetypes allows us to add, remove, and edit the Maven archetype catalog, as shown in the following screenshot:

For more information on archetypes, please refer to `http://maven.apache.org/archetype/index.html`.



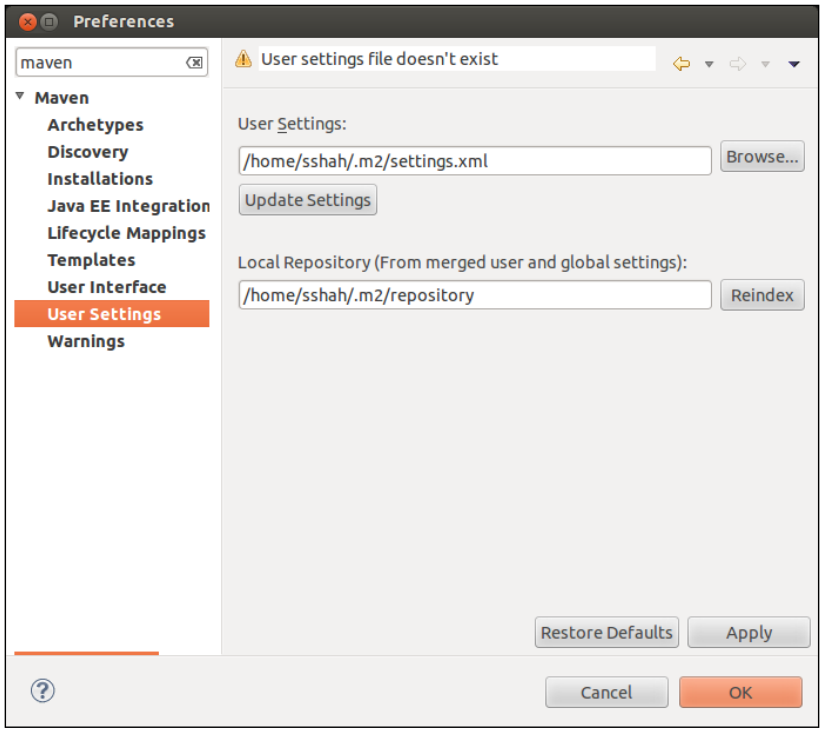
User Interface and User Settings

User Interface allows us to set XML file options, as shown in the following screenshot:



The `settings.xml` file contains elements used to define values that configure Maven execution in various ways, such as the `pom.xml` file. The `settings` file is at `$ {M2_HOME}/settings.xml`, where `M2_HOME` is `{USER_HOME}/.m2`. In the *Local Repository* section of *Chapter 3*, *Creating and Importing Projects*, we use this file to set the alternate local repository other than the default one.

User Settings allows us to use the custom settings file and re-index the local repository, as shown in the following screenshot:



For more information on settings, please refer to `http://maven.apache.org/settings.html#Servers`.

Installations

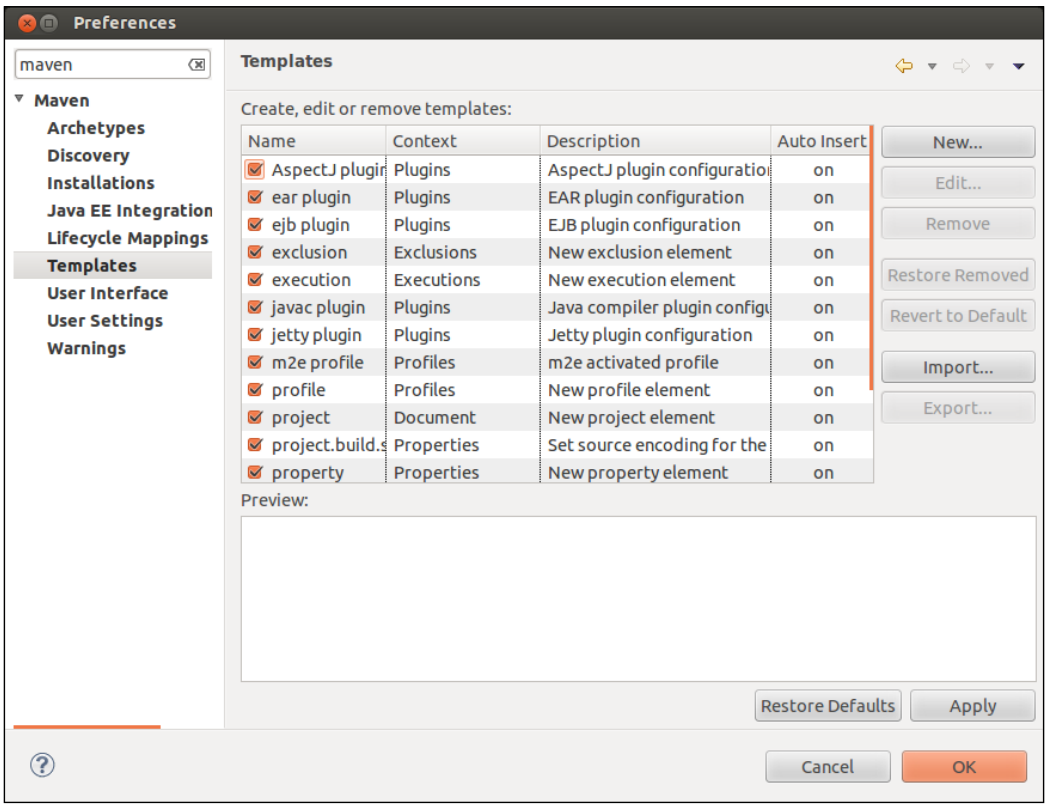
Installations shows Maven installations and allows us to choose the Maven to use. We used it to set the external Maven installation in the *Setting Maven to Use* section of *Chapter 2*, *Installing m2Eclipse*; please refer to it for more details.

Warnings

Warnings allows us to enable/disable the warning for a duplicate group ID and version across the parent-child POM. At the time of writing this, though this option is enabled, m2eclipse still complains about duplicates; hopefully, this feature will work in days to come with other new releases.

Templates

Templates shows the list of all the templates used by Maven. It also provides an option to add new templates, edit, remove, import, and export the templates, as shown in the following screenshot:



Lifecycle Mappings

Lifecycle Mappings allows us to customize the project build lifecycle for Maven projects used by m2eclipse. This feature is still experimental at the time of writing this book; hence, we will limit its discussion.

For more information, please refer to `http://wiki.eclipse.org/M2E_plugin_execution_not_covered`.

Summary

In this chapter, you learned about other additional features available in m2eclipse, and got familiar with the repositories, form-based POM editor, and m2eclipse preferences. So, in this book, you learned Maven concepts; m2eclipse and features; and its ease in creating, building, and running Maven projects.

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W