2

Getting Started with Maven

This chapter covers:

- Preparing to use Maven
- Creating your first project
- Compiling application sources
- Compiling test sources and running unit tests
- Packaging an installation to your local repository
- Handling classpath resources
- Using Maven plugins

The key to performance is elegance, not battalions of special cases. The terrible temptation to tweak should be resisted unless the payoff is really noticeable.

- Jon Bentley and Doug McIlroy

2.1. Preparing to Use Maven

In this chapter, it is assumed that you are a first time Maven user and have already set up Maven on your local system. If you have not set up Maven yet, then please refer to Maven's Download and Installation Instructions before continuing. Depending on where your machine is located, it may be necessary to make a few more preparations for Maven to function correctly. If you are behind a firewall, then you will have to set up Maven to understand that. To do this, create a `<your-home-directory>/.m2/settings.xml` file with the following content:

```
<settings>
 <proxies>
  <proxy>
   <active>true</active>
   <protocol>http</protocol>
   <host>proxy.mycompany.com</host>
   <port>8080</port>
   <username>your-username</username>
   <password>your-password</password>
  </proxy>
 </proxies>
</settings>
```

If Maven is already in use at your workplace, ask your administrator if there if there is an internal Maven proxy. If there is an active Maven proxy running, then note the URL and let Maven know you will be using a proxy. Create a `<your-home-directory>/.m2/settings.xml` file with the following content.

```
<settings>
<mirrors>
  <mirror>
   <id>maven.mycompany.com</id>
   <name>My Company's Maven Proxy</name>
   <url>http://maven.mycompany.com/maven2</url>
   <mirrorOf>central</mirrorOf>
  </mirror>
 </mirrors>
</settings>
```

In its optimal mode, Maven requires network access, so for now simply assume that the above settings will work. The settings.xml file will be explained in more detail in the following chapter and you can refer to the Maven Web site for the complete details on the `settings.xml` file. Now you can perform the following basic check to ensure Maven is working correctly:

```
mvn -version
```

If Maven's version is displayed, then you should be all set to create your first Maven project.

2.2. Creating Your First Maven Project

To create your first project, you will use Maven's **Archetype** mechanism. An archetype is defined as an original pattern or model from which all other things of the same kind are made. In Maven, an archetype is a template of a project, which is combined with some user input to produce a fully-functional Maven project. This chapter will show you how the archetype mechanism works, but if you would like more information about archetypes, please refer to the Introduction to Archetypes.

To create the Quick Start Maven project, execute the following:

```
C:\mvnbook> mvn archetype:create -DgroupId=com.mycompany.app \
-DartifactId=my-app
```

You will notice a few things happened when you executed this command. First, you will notice that a directory named `my-app` has been created for the new project, and this directory contains your `pom.xml`, which looks like the following:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <packaging>jar</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>Maven Quick Start Archetype</name>
 <url>http://maven.apache.org</url>
 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <scope>test</scope>
  </dependency>
 </dependencies>
</project>
```

At the top level of every project is your `pom.xml` file. Whenever you see a directory structure which contains a `pom.xml` file you know you are dealing with a Maven project. After the archetype generation has completed you will notice that the following directory structure has been created, and it in fact adheres to Maven's standard directory layout discussed in Chapter 1.

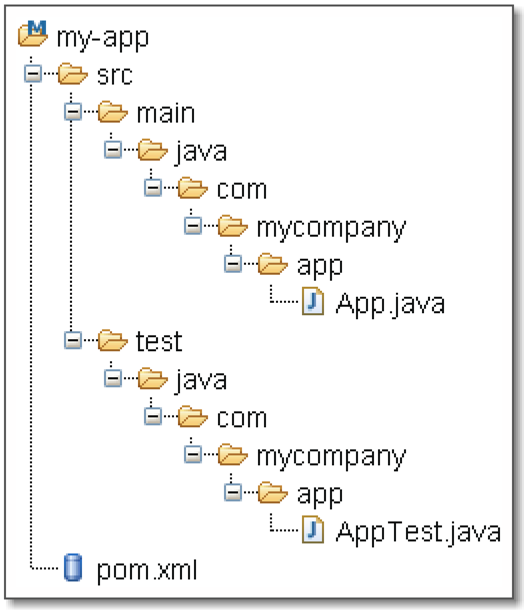


Figure 2-1: Directory structure after archetype generation

The `src` directory contains all of the inputs required for building, testing, documenting, and deploying the project (source files, configuration files, various descriptors such as assembly descriptors, the site, and so on). In this first stage you have Java source files only, but later in the chapter you will see how the standard directory layout is employed for other project content.

Now that you have a POM, some application sources, and some test sources, you are ready to build your project.

2.3. Compiling Application Sources

As mentioned in the introduction, at a very high level, you tell Maven what you need, in a declarative way, in order to accomplish the desired task. Before you issue the command to compile the application sources, note that manifest in this one simple command are Maven's four foundational principles:

- Convention over configuration
- Reuse of build logic
- Declarative execution
- Coherent organization of dependencies

These principles are ingrained in all aspects of Maven, but the following analysis of the simple compile command shows you the four principles in action and makes clear their fundamental importance in simplifying the development of a project.

Change into the `<my-app>` directory. The `<my-app>` directory is the base directory, `${basedir}`, for the `my-app` project. Then, in one fell swoop, you will compile your application sources using the following command:

```
C:\mvnbook\my-app> mvn compile
```

After executing this command you should see output similar to the following:

```
[INFO-------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]  task-segment: [compile]
[INFO]------------------------------------------------------------------
[INFO] artifact org.apache.maven.plugins:maven-resources-plugin: checking for
updates from central
...
[INFO] artifact org.apache.maven.plugins:maven-compiler-plugin: checking for
updates from central
...
[INFO] [resources:resources]
...
[INFO] [compiler:compile]
Compiling 1 source file to c:\mvnbook\my-app\target\classes
[INFO]------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------
[INFO] Total time: 3 minutes 54 seconds
[INFO] Finished at: Fri Sep 23 15:48:34 GMT-05:00 2005
[INFO] Final Memory: 2M/6M
[INFO]------------------------------------------------------------------
```

Now let's dissect what actually happened and see where Maven's four principles come into play with the execution of this seemingly simple command.

How did Maven know where to look for sources in order to compile them? And how did Maven know where to put the compiled classes? This is where Maven's principle of “convention over configuration” comes into play. By default, application sources are placed in `src/main/java`. This default value (though not visible in the POM above) was, in fact, inherited from the Super POM. Even the simplest of POMs knows the default location for application sources. This means you don't have to state this location at all in any of your POMs, if you use the default location for application sources. You can, of course, override this default location, but there is very little reason to do so. The same holds true for the location of the compiled classes which, by default, is target/classes.

What actually compiled the application sources? This is where Maven's second principle of “reusable build logic” comes into play. The standard compiler plugin, along with its default configuration, is the tool used to compile your application sources. The same build logic encapsulated in the compiler plugin will be executed consistently across any number of projects.

Although you now know that the compiler plugin was used to compile the application sources, how was Maven able to decide to use the compiler plugin, in the first place? You might be guessing that there is some background process that maps a simple command to a particular plugin. In fact, there is a form of mapping and it is called Maven's default **build life cycle**.

So, now you know how Maven finds application sources, what Maven uses to compile the application sources, and how Maven invokes the compiler plugin. The next question is, how was Maven able to retrieve the compiler plugin? After all, if you poke around the standard Maven installation, you won't find the compiler plugin since it is not shipped with the Maven distribution. Instead, Maven downloads plugins when they are needed.

The first time you execute this (or any other) command, Maven will download all the plugins and related dependencies it needs to fulfill the command. From a clean installation of Maven this can take quite a while (in the output above, it took almost 4 minutes with a broadband connection). The next time you execute the same command again, because Maven already has what it needs, it won't download anything new. Therefore, Maven will execute the command much quicker.

As you can see from the output, the compiled classes were placed in `target/classes`, which is specified by the standard directory layout. If you're a keen observer you'll notice that using the standard conventions makes the POM above very small, and eliminates the requirement for you to explicitly tell Maven where any of your sources are, or where your output should go. By following the standard Maven conventions you can get a lot done with very little effort!

2.4. Compiling Test Sources and Running Unit Tests

Now that you're successfully compiling your application's sources, you probably have unit tests that you want to compile and execute as well (after all, programmers always write and execute their own unit tests *nudge nudge, wink wink*).

Again, simply tell Maven you want to test your sources. This implies that all prerequisite phases in the life cycle will be performed to ensure that testing will be successful. Use the following simple command to test:

```
C:\mvnbook\my-app> mvn test
```

After executing this command you should see output similar to the following:

```
[INFO]----------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]  task-segment: [test]
[INFO]----------------------------------------------------------------
[INFO] artifact org.apache.maven.plugins:maven-surefire-plugin: checking for
updates from central
...
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
Compiling 1 source file to C:\Test\Maven2\test\my-app\target\test-classes
...
[INFO] [surefire:test]
[INFO] Setting reports dir: C:\Test\Maven2\test\my-app\target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.mycompany.app.AppTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0 sec

Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0

[INFO]----------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]----------------------------------------------------------------
[INFO] Total time: 15 seconds
[INFO] Finished at: Thu Oct 06 08:12:17 MDT 2005
[INFO] Final Memory: 2M/8M
[INFO]----------------------------------------------------------------
```

Some things to notice about the output:

- Maven downloads more dependencies this time. These are the dependencies and plugins necessary for executing the tests (recall that it already has the dependencies it needs for compiling and won't download them again).
- Before compiling and executing the tests, Maven compiles the main code (all these classes are up-to-date, since we haven't changed anything since we compiled last).

If you simply want to compile your test sources (but not execute the tests), you can execute the following command:

```
C:\mvnbook\my-app> mvn test-compile
```

However, remember that it isn't necessary to run this every time; `mvn` test will always run the `compile` and `test-compile` phases first, as well as all the others defined before it.

Now that you can compile the application sources, compile the tests, and execute the tests, you'll want to move on to the next logical step, how to package your application.

2.5. Packaging and Installation to Your Local Repository

Making a JAR file is straightforward and can be accomplished by executing the following command:

```
C:\mvnbook\my-app> mvn package
```

If you take a look at the POM for your project, you will notice the packaging element is set to jar. This is how Maven knows to produce a JAR file from the above command (you'll read more about this later). Take a look in the the target directory and you will see the generated JAR file.

Now, you'll want to install the artifact (the JAR file) you've generated into your local repository. It can then be used by other projects as a dependency. The directory `<your-home-directory>/.m2/repository` is the default location of the repository. To install, execute the following command:

```
C:\mvnbook\my-app> mvn install
```

Upon executing this command you should see the following output:

```
[INFO]-------------------------------------------------------------------
[INFO] Building Maven Quick Start Archetype
[INFO]  task-segment: [install]
[INFO]-------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] [compiler:compile]
Compiling 1 source file to <dir>/my-app/target/classes
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
Compiling 1 source file to <dir>/my-app/target/test-classes
[INFO] [surefire:test]
[INFO] Setting reports dir: <dir>/my-app/target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.mycompany.app.AppTest
[surefire] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.001 sec

Results :
[surefire] Tests run: 1, Failures: 0, Errors: 0

[INFO] [jar:jar]
[INFO] Building jar: <dir>/my-app/target/my-app-1.0-SNAPSHOT.jar
[INFO] [install:install]
[INFO] Installing c:\mvnbook\my-app\target\my-app-1.0-SNAPSHOT.jar to <local-
repository>\com\mycompany\app\my-app\1.0-SNAPSHOT\my-app-1.0-SNAPSHOT.jar
[INFO]-------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]-------------------------------------------------------------------
[INFO] Total time: 5 seconds
[INFO] Finished at: Tue Oct 04 13:20:32 GMT-05:00 2005
[INFO] Final Memory: 3M/8M
[INFO]-------------------------------------------------------------------
```

Note that the Surefire plugin (which executes the test) looks for tests contained in files with a particular naming convention. By default, the following tests are included:

- `**/*Test.java`
- `**/Test*.java`
- `**/*TestCase.java`

Conversely, the following tests are excluded:

- `**/Abstract*Test.java`
- `**/Abstract*TestCase.java`

You have now completed the process for setting up, building, testing, packaging, and installing a typical Maven project. For projects that are built with Maven, this covers the majority of tasks users perform, and if you've noticed, everything done up to this point has been driven by an 18-line POM.

Of course, there is far more functionality available to you from Maven without requiring any additions to the POM, as it currently stands. In contrast, to get any more functionality out of an Ant build script, you must keep making error-prone additions.

So, what other functionality can you leverage, given Maven's re-usable build logic? With even the simplest POM, there are a great number of Maven plugins that work out of the box. This chapter will cover one in particular, as it is one of the highly-prized features in Maven. Without any work on your part, this POM has enough information to generate a Web site for your project! Though you will typically want to customize your Maven site, if you're pressed for time and just need to create a basic Web site for your project, simply execute the following command:

```
C:\mvnbook\my-app> mvn site
```

There are plenty of other stand-alone goals that can be executed as well, for example:

```
C:\mvnbook\my-app> mvn clean
```

This will remove the target directory with the old build data before starting, so it is fresh. Perhaps you'd like to generate an `IntelliJ` IDEA descriptor for the project:

```
C:\mvnbook\my-app> mvn idea:idea
```

This can be run over the top of a previous IDEA project. In this case, it will update the settings rather than starting fresh.

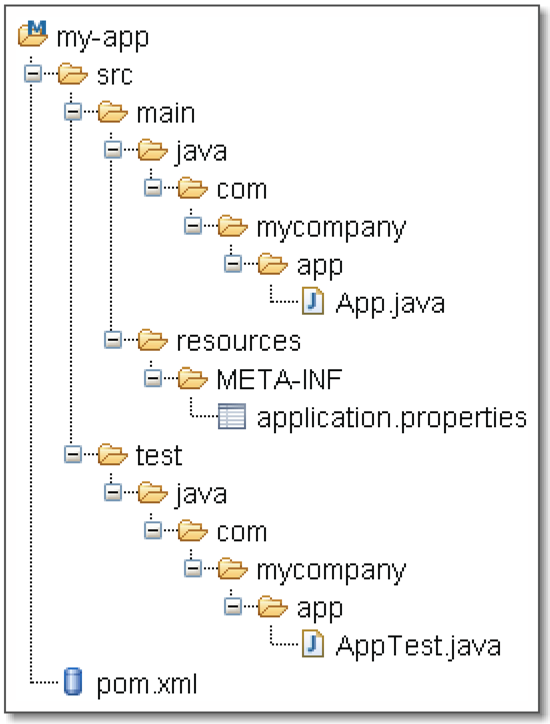
Or, alternatively you might like to generate an Eclipse descriptor:

```
C:\mvnbook\my-app> mvn eclipse:eclipse
```

2.6. Handling Classpath Resources

Another common use case which requires no changes to the POM shown previously, is the packaging of resources into a JAR file. For this common task, Maven again uses the standard directory layout. This means that by adopting Maven's standard conventions, you can package resources within JARs, simply by placing those resources in a standard directory structure.

In the example following, you need to add the directory `src/main/resources`. That is where you place any resources you wish to package in the JAR. The rule employed by Maven is that all directories or files placed within the `src/main/resources` directory are packaged in your JAR with the exact same structure, starting at the base of the JAR.



```
my-app
├─ src
│  ├─ main
│  │  ├─ java
│  │  │  └─ com
│  │  │     └─ mycompany
│  │  │        └─ app
│  │  │           └─ App.java
│  │  └─ resources
│  │     └─ META-INF
│  │        └─ application.properties
│  └─ test
│     └─ java
│        └─ com
│           └─ mycompany
│              └─ app
│                 └─ AppTest.java
└─ pom.xml
```

Figure 2-2: Directory structure after adding the resources directory

You can see in the preceding example that there is a `META-INF` directory with an `application.properties` file within that directory. If you unpacked the JAR that Maven created you would see the following:

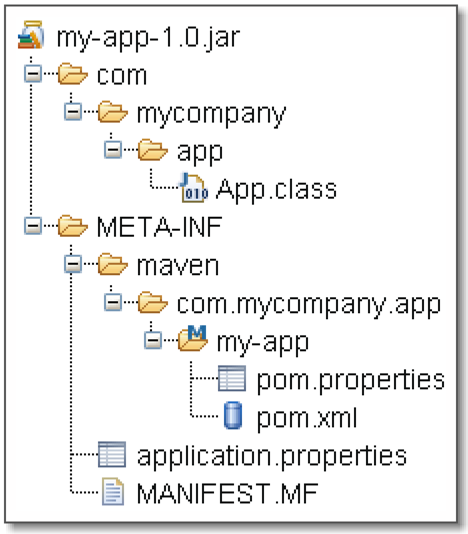


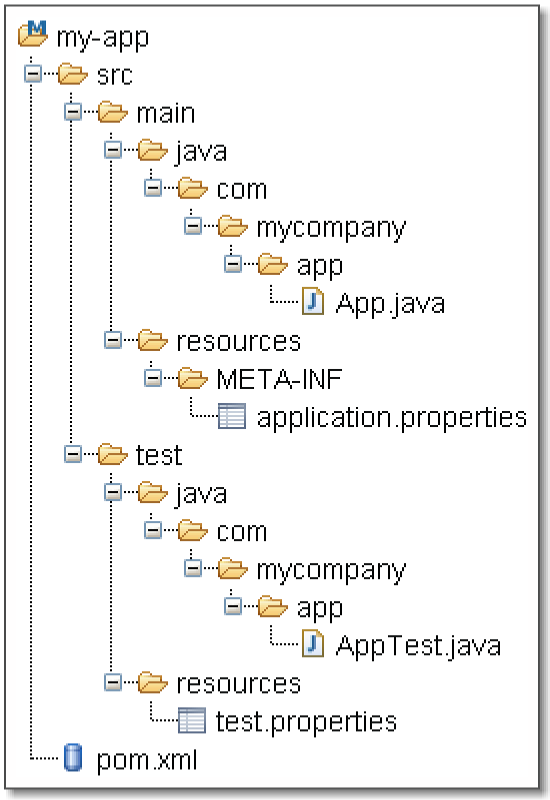
Figure 2-3: Directory structure of the JAR file created by Maven

The original contents of `src/main/resources` can be found starting at the base of the JAR and the `application.properties` file is there in the `META-INF` directory. You will also notice some other files there like `META-INF/MANIFEST.MF`, as well as a `pom.xml` and `pom.properties` file. These come standard with the creation of a JAR in Maven. You can create your own manifest if you choose, but Maven will generate one by default if you don't. If you would like to try this example, simply create the `resources` and `META-INF` directories and create an empty file called `application.xml` inside. Then run `mvn install` and examine the jar file in the target directory.

The `pom.xml` and `pom.properties` files are packaged up in the JAR so that each artifact produced by Maven is self-describing and also allows you to utilize the metadata in your own application, should the need arise. One simple use might be to retrieve the version of your application. Operating on the POM file would require you to use Maven utilities, but the properties can be utilized using the standard Java APIs.

2.6.1. Handling Test Classpath Resources

To add resources to the classpath for your unit tests, follow the same pattern as you do for adding resources to the JAR, except place resources in the `src/test/resources directory`. At this point you have a project directory structure that should look like the following:



```
my-app
  src
    main
      java
        com
          mycompany
            app
              App.java
      resources
        META-INF
          application.properties
    test
      java
        com
          mycompany
            app
              AppTest.java
      resources
        test.properties
  pom.xml
```

Figure 2-4: Directory structure after adding test resources

In a unit test, you could use a simple snippet of code like the following for access to the resource required for testing:

```
[...]
// Retrieve resource
InputStream is = getClass().getResourceAsStream( "/test.properties" );

// Do something with the resource

[...]
```

To override the manifest file yourself, you can use the follow configuration for the `maven-jar-plugin`:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>META-INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
```

2.6.2. Filtering Classpath Resources

Sometimes a resource file will need to contain a value that can be supplied at build time only. To accomplish this in Maven, you can filter your resource files dynamically by putting a reference to the property that will contain the value into your resource file using the syntax `${<property name>}`. The property can be either one of the values defined in your `pom.xml`, a value defined in the user's `settings.xml`, a property defined in an external properties file, or a system property.

To have Maven filter resources when copying, simply set filtering to `true` for the resource directory in your `pom.xml`:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <packaging>jar</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>Maven Quick Start Archetype</name>
 <url>http://maven.apache.org</url>
 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <scope>test</scope>
  </dependency>
 </dependencies>
 <build>
  <resources>
   <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
   </resource>
  </resources>
 </build>
</project>
```

You'll notice that the build, resources, and resource elements have been added, which weren't there before. In addition, the POM has to explicitly state that the resources are located in the `src/main/resources` directory. All of this information was previously provided as default values and now must be added to the `pom.xml` to override the default value for filtering and set it to `true`.

To reference a property defined in your `pom.xml`, the property name uses the names of the XML elements that define the value. So `${project.name}` refers to the name of the project, `${project.version}` refers to the version of the project, and `${project.build.finalName}` refers to the final name of the file created, when the built project is packaged. In fact, any element in your POM is available when filtering resources.

To continue the example, create an `src/main/resources/application.properties` file, whose values will be supplied when the resource is filtered as follows:

```
# application.properties
application.name=${project.name}
application.version=${project.version}
```

With that in place, you can execute the following command (`process-resources` is the build life cycle phase where the resources are copied and filtered):

```
mvn process-resources
```

The `application.properties` file under target/classes, which will eventually go into the JAR looks like this:

```
# application.properties
application.name=Maven Quick Start Archetype
application.version=1.0-SNAPSHOT
```

To reference a property defined in an external file, all you need to do is add a reference to this external file in your `pom.xml`. First, create an external properties file and call it `src/main/filters/filter.properties`:

```
# filter.properties
my.filter.value=hello!
```

Next, add a reference to this new file in the `pom.xml` file:

```
[...]
 <build>
  <filters>
   <filter>src/main/filters/filter.properties</filter>
  </filters>
  <resources>
   <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
   </resource>
  </resources>
 </build>
[...]
```

Then, add a reference to this property in the `application.properties` file as follows:

```
# application.properties
application.name=${project.name}
application.version=${project.version}
message=${my.filter.value}
```

The next execution of the `mvn process-resources` command will put the new property value into `application.properties`. As an alternative to defining the `my.filter.value` property in an external file, you could have defined it in the properties section of your `pom.xml` and you'd get the same effect (notice you don't need the references to `src/main/filters/filter.properties` either):

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <packaging>jar</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>Maven Quick Start Archetype</name>
 <url>http://maven.apache.org</url>
 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <scope>test</scope>
  </dependency>
 </dependencies>
 <build>
  <resources>
   <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
   </resource>
  </resources>
 </build>
 <properties>
  <my.filter.value>hello</my.filter.value>
 </properties>
</project>
```

Filtering resources can also retrieve values from system properties; either the system properties built into Java (like `java.version` or `user.home`), or properties defined on the command line using the standard Java -D parameter. To continue the example, change the `application.properties` file to look like the following:

```
# application.properties
java.version=${java.version}
command.line.prop=${command.line.prop}
```

Now, when you execute the following command (note the definition of the `command.line.prop` property on the command line), the `application.properties` file will contain the values from the system properties.

```
mvn process-resources "-Dcommand.line.prop=hello again"
```

2.6.3. Preventing Filtering of Binary Resources

Sometimes there are classpath resources that you want included in your JAR, but you do not want them filtered. This is most often the case with binary resources, for example image files.

If you had a `src/main/resources/images` that you didn't want to be filtered, then you would create one resource entry that handled the filtering of resources with an exclusion on the resources you wanted unfiltered. In addition you would add another resource entry, with filtering disabled, and an inclusion of your images directory. The build element would look like the following:

```
<project>
...
 <build>
  <resources>
   <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
    <excludes>
     <exclude>images/**</exclude>
    </excludes>
   </resource>
   <resource>
    <directory>src/main/resources</directory>
    <includes>
     <include>images/**</include>
    </includes>
   </resource>
  </resources>
 </build>
...
</project>
```

2.7. Using Maven Plugins

As noted earlier in the chapter, to customize the build for a Maven project, you must include additional Maven plugins, or configure parameters for the plugins already included in the build.

For example, you may want to configure the Java compiler to allow JDK 5.0 sources. This is as simple as adding this following to your POM:

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

You'll notice that all plugins in Maven 2 look very similar to a dependency,and in some ways they are. This plugin will be downloaded and installed automatically, if it is not present on your local system, in much the same way that a dependency would be handled. To illustrate the similarity between plugins and dependencies, the `groupId` and `version` elements have been shown, but in most cases these elements are not required.

If you do not specify a `groupId`, then Maven will default to looking for the plugin with the `org.apache.maven.plugins` or the `org.codehaus.mojo groupId` label. You can specify an additional `groupId` to search within your POM, or `settings.xml`.

If you do not specify a version then Maven will attempt to use the latest released version of the specified plugin. This is often the most convenient way to use a plugin, but you may want to specify the version of a plugin to ensure reproducibility. For the most part, plugin developers take care to ensure that new versions of plugins are backward compatible so you are usually OK with the latest release, but if you find something has changed - you can lock down a specific version.

The configuration element applies the given parameters to every goal from the compiler plugin. In the above case, the compiler plugin is already used as part of the build process and this just changes the configuration.

If you want to find out what the plugin's configuration options are, use the `mvn help:describe` command. If you want to see the options for the maven-compiler-plugin shown previously, use the following command:

```
mvn help:describe -DgroupId=org.apache.maven.plugins \
 -DartifactId=maven-compiler-plugin -Dfull=true
```

You can also find out what plugin configuration is available by using the Maven Plugin Reference section at http://maven.apache.org/plugins/ and navigating to the plugin and goal you are using.

2.8. Summary

After reading this second chapter, you should be up and running with Maven. If someone throws a Maven project at you, you'll know how to exercise the basic features of Maven: creating a project, compiling a project, testing a project, and packaging a project. By learning how to build a Maven project, you have gained access to every single project using Maven. You've learned a new language and you've taken Maven for a test drive.

You should also have some insight into how Maven handles dependencies and provides an avenue for customization using Maven plugins. In seventeen pages, you've seen how you can use Maven to build your project. If you were just looking for a build tool, you could safely stop reading this book now. You might need to refer to the next chapter for more information about customizing your build to fit your project's unique needs.

If you are interested in learning how Maven builds upon the concepts introduced in the Introduction and the practical tools introduced in Chapter 2, read on. The next few chapters provide you with the tools to customize Maven's behavior and use Maven to manage interdependent software projects.