Table of Contents

46.1. The Java plugin

As we have seen, Gradle is a general-purpose build tool. It can build pretty much anything you care to implement in your build script. Out-of-the-box, however, it doesn't build anything unless you add code to your build script to do so.

Most Java projects are pretty similar as far as the basics go: you need to compile your Java source files, run some unit tests, and create a JAR file containing your classes. It would be nice if you didn't have to code all this up for every project. Luckily, you don't have to. Gradle solves this problem through the use of *plugins*. A plugin is an extension to Gradle which configures your project in some way, typically by adding some pre-configured tasks which together do something useful. Gradle ships with a number of plugins, and you can easily write your own and share them with others. One such plugin is the *Java plugin*. This plugin adds some tasks to your project which will compile and unit test your Java source code, and bundle it into a JAR file.

The Java plugin is convention based. This means that the plugin defines default values for many aspects of the project, such as where the Java source files are located. If you follow the convention in your project, you generally don't need to do much in your build script to get a useful build. Gradle allows you to customize your project if you don't want to or cannot follow the convention in some way. In fact, because support for Java projects is implemented as a plugin, you don't have to use the plugin at all to build a Java project, if you don't want to.

46.2. A basic Java project

Let's look at a simple example. To use the Java plugin, add the following to your build file:

Example 46.1. Using the Java plugin

`build.gradle`

```
apply plugin: 'java'
```

> **Note:** The code for this example can be found at **`samples/java/quickstart`** in the '-all' distribution of Gradle.

This is all you need to define a Java project. This will apply the Java plugin to your project, which adds a number of tasks to your project.

Gradle expects to find your production source code under `src/main/java` and your test source code under `src/test/java`. In addition, any files under `src/main/resources` will be included in the JAR file as resources, and any files under `src/test/resources` will be included in the classpath used to run the tests. All output files are created under the `build` directory, with the JAR file ending up in the `build/libs` directory.

What tasks are available?

You can use **`gradle tasks`** to list the tasks of a project. This will let you see the tasks that the Java plugin has added to your project.

46.2.1. Building the project

The Java plugin adds quite a few tasks to your project.
However, there are only a handful of tasks that you will need to use to build the project. The most commonly used task is the `build` task, which does a full build of the project. When you run **`gradle build`**, Gradle will compile and test your code, and create a JAR file containing your main classes and resources:

Example 46.2. Building a Java project

Output of **`gradle build`**

```
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 1 secs
```

Some other useful tasks are:

clean
: Deletes the `build` directory, removing all built files.

assemble
: Compiles and jars your code, but does not run the unit tests. Other plugins add more artifacts to this task. For example, if you use the War plugin, this task will also build the WAR file for your project.

check
: Compiles and tests your code. Other plugins add more checks to this task. For example, if you use the `checkstyle` plugin, this task will also run Checkstyle against your source code.

46.2.2. External dependencies

Usually, a Java project will have some dependencies on external JAR files. To reference these JAR files in the project, you need to tell Gradle where to find them. In Gradle, artifacts such as JAR files, are located in a *repository*. A repository can be used for fetching the dependencies of a project, or for publishing the artifacts of a project, or both. For this example, we will use the public Maven repository:

Example 46.3. Adding Maven repository

`build.gradle`

Let's add some dependencies. Here, we will declare that our production classes have a compile-time dependency on commons collections, and that our test classes have a compile-time dependency on junit:

Example 46.4. Adding dependencies

`build.gradle`

```
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

You can find out more in Chapter 7, *Dependency Management Basics*.

46.2.3. Customizing the project

The Java plugin adds a number of properties to your project. These properties have default values which are usually sufficient to get started. It's easy to change these values if they don't suit. Let's look at this for our sample. Here we will specify the version number for our Java project, along with the Java version our source is written in. We also add some attributes to the JAR manifest.

Example 46.5. Customization of MANIFEST.MF

`build.gradle`

```
sourceCompatibility = 1.7
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}
```

The tasks which the Java plugin adds are regular tasks, exactly the same as if they were declared in the build file. This means you can use any of the mechanisms shown in earlier chapters to customize these tasks. For example, you can set the properties of a task, add behaviour to a task, change the

Example 46.6. Adding a test system property

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

are available?

You can use **gradle properties** to list the properties of a project. This will allow you to see the properties added by the Java plugin, and their default values.

46.2.4. Publishing the JAR file

Usually the JAR file needs to be published somewhere. To do this, you need to tell Gradle where to publish the JAR file. In Gradle, artifacts such as JAR files are published to repositories. In our sample, we will publish to a local directory. You can also publish to a remote location, or multiple locations.

Example 46.7. Publishing the JAR file

build.gradle

```
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

To publish the JAR file, run **gradle uploadArchives**.

46.2.5. Creating an Eclipse project

To create the Eclipse-specific descriptor files, like `.project`, you need to add another plugin to your build file:

Example 46.8. Eclipse plugin

build.gradle

```
apply plugin: 'eclipse'
```

Here's the complete build file for our sample:

Example 46.9. Java example - complete build file

`build.gradle`

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility = 1.7
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

46.3. Multi-project Java build

```
multiproject/
  api/
  services/webservice/
  shared/
  services/shared/
```

> **Note:** The code for this example can be found at **`samples/java/multiproject`** in the '-all' distribution of Gradle.

Here we have four projects. Project `api` produces a JAR file which is shipped to the client to provide them a Java client for your XML webservice. Project `webservice` is a webapp which returns XML. Project `shared` contains code used both by `api` and `webservice`. Project `services/shared` has code that depends on the `shared` project.

46.3.1. Defining a multi-project build

To define a multi-project build, you need to create a *settings file*. The settings file lives in the root directory of the source tree, and specifies which projects to include in the build. It must be called `settings.gradle`. For this example, we are using a simple hierarchical layout. Here is the corresponding settings file:

Example 46.11. Multi-project build - settings.gradle file

`settings.gradle`

```
include "shared", "api", "services:webservice", "services:shared"
```

You can find out more about the settings file in Chapter 26, *Multi-project Builds*.

46.3.2. Common configuration

For most multi-project builds, there is some configuration which is common to all projects. In our sample, we will define this common configuration in the root project, using a technique called *configuration injection*. Here, the root project is like a container and the `subprojects` method iterates over the elements of this container - the projects in this instance - and injects the specified

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.12'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

Notice that our sample applies the Java plugin to each subproject. This means the tasks and configuration properties we have seen in the previous section are available in each subproject. So, you can compile, test, and JAR all the projects by running **`gradle build`** from the root project directory.

Also note that these plugins are only applied within the `subprojects` section, not at the root level, so the root build will not expect to find Java source files in the root project, only in the subprojects.

46.3.3. Dependencies between projects

You can add dependencies between projects in the same build, so that, for example, the JAR file of one project is used to compile another project. In the `api` build file we will add a dependency on the `shared` project. Due to this dependency, Gradle will ensure that project `shared` always gets built before project `api`.

Example 46.13. Multi-project build - dependencies between projects

api/build.gradle

See Section 26.7.1, “Disabling the build of dependency projects” for how to disable this functionality.

46.3.4. Creating a distribution

We also add a distribution, that gets shipped to the client:

Example 46.14. Multi-project build - distribution file

`api/build.gradle`

```
task dist(type: Zip) {
    dependsOn spiJar
    from 'src/dist'
    into('libs') {
        from spiJar.archivePath
        from configurations.runtime
    }
}

artifacts {
   archives dist
}
```

46.4. Where to next?

In this chapter, you have seen how to do some of the things you commonly need to build a Java based project. This chapter is not exhaustive, and there are many other things you can do with Java projects in Gradle. You can find out more about the Java plugin in Chapter 47, *The Java Plugin*, and you can find more sample Java projects in the `samples/java` directory in the Gradle distribution.

Otherwise, continue on to Chapter 7, *Dependency Management Basics*.