GRADLE

tutorialspoint

SIMPLYEASYLEARNING

About the Tutorial

Gradle is an open source, advanced general purpose build management system. It is built on ANT, Maven, and lvy repositories. It supports Groovy based Domain Specific Language (DSL) over XML.

This tutorial explains how you can use Gradle as a build automation tool for Java as well as Groovy projects.

Audience

This tutorial is designed for those software professionals who would like to learn the basics of Gradle as a build tool in simple and easy steps.

Prerequisites

Gradle is a Groovy-based build automation tool. So, it will certainly help if you have some prior exposure to Groovy. In addition, you should have working knowledge of Java.

Disclaimer & Copyright

Table of Contents

1. Gradle –Overview

ANT and Maven shared considerable success in the Java marketplace. ANT was the first build tool released in 2000 and it is developed based on procedural programming idea. Later, it was improved with an ability to accept plug-ins and dependency management over the network with the help on Apache-Ivy. The main drawback was XML as a format to write build scripts. XML being hierarchical is not good for procedural programming and tends to become unmanageably big.

Maven was introduced in 2004. It comes with a lot of improvement than ANT. It changes its structure and it continues using XML for writing build specifications. Maven relies on the conventions and is able to download the dependencies over the network. The main benefit of Maven is its life cycle. While following the same life cycle for multiple projects continuously, this comes a cost of flexibility. Maven also faces some problems in dependency management. It does not handle well conflicts between versions of the same library, and complex customized build scripts are actually harder to write in Maven than in ANT.

Finally, Gradle came into the picture in 2012. Gradle carries some efficient features from both the tools.

Features of Gradle

Following is the list of features that Gradle provides.

- **Declarative builds and build-by-convention:** Gradle is available with separate Domain Specific Language (DSL) based on Groovy language. Gradle provides declarative language elements. The elements also provide build-by-convention support for Java, Groovy, OSGi, Web and Scala.

- **Language for dependency based programming:** The declarative language lies on top of a general purpose task graph, which you can fully leverage in your build.

- **Structure your build:** Gradle allows you to apply common design principles to your build. It gives you a perfect structure for build, so that you can design well-structured and easily maintained, comprehensible build.

- **Deep API:** Using this API, you can monitor and customize its configuration and execution behavior to its core.

- **Gradle scales:** Gradle can easily increase productivity, from simple and single project builds to huge enterprise multi-project builds.

- **Multi-project builds:** Gradle supports multi-project builds and also partial builds. If you build a subproject, Gradle takes care of building all the subprojects that it depends on.

- **Different ways to manage your builds:** Gradle supports different strategies to manage your dependencies.

- **First build integration tool:** Gradle completely supports ANT tasks, Maven and Ivy repository infrastructure for publishing and retrieving dependencies. It also provides a converter for turning a Maven pom.xml to Gradle script.

- **Ease of migration:** Gradle can easily adapt to any structure you have. Therefore, you can always develop your Gradle build in the same branch where you can build live script.

- **Gradle Wrapper:** Gradle Wrapper allows you to execute Gradle builds on machines where Gradle is not installed. This is useful for continuous integration of servers.

- **Free open source:** Gradle is an open source project, and licensed under the Apache Software License (ASL).

- **Groovy:** Gradle's build script is written in Groovy. The whole design of Gradle is oriented towards being used as a language, not as a rigid framework. Groovy allows you to write your own script with some abstractions. The entire Gradle API is designed in Groovy language.

Why Groovy?

The complete Gradle API is designed using Groovy language. This is an advantage of an internal DSL over XML. Gradle is general purpose build tool at its core; its main focus is Java projects. In such projects, the team members will be very familiar with Java and it is better that a build should be as transparent as possible to all team members.

Languages like Python, Groovy or Ruby are better for build framework. Why Groovy was chosen is, because it offers by far the greatest transparency for people using Java. The base syntax of Groovy is same as Java. Groovy provides much more on top of that.

2. Gradle – Installation

Gradle is a build tool, based on Java. There are some prerequisites that needs to be installed before installing the Gradle framework.

Prerequisites

JDK and Groovy are the prerequisites for Gradle installation.

- Gradle requires JDK version 6 or later to be installed in your system. It uses the JDK libraries which is installed and sets to the JAVA_HOME environmental variable.

- Gradle carries its own Groovy library, therefore, we do no need to install Groovy explicitly. If it is installed, that is ignored by Gradle.

Following are the steps to install Gradle in your system.

Step 1: Verify JAVA Installation

First of all, you need to have Java Software Development Kit (SDK) installed on your system. To verify this, execute **Java –version** command in any of the platform you are working on.

In Windows

Execute the following command to verify Java installation. I have installed JDK 1.8 in my system.

```
C:\> java - version
```

If the command is executed successfully, you will get the following output.

```
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b18, mixed mode)
```

In Linux

Execute the following command to verify Java installation. I have installed JDK 1.8 in my system.

```
$ java - version
```

If the command is executed successfully, you will get the following output.

```
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b18, mixed mode)
```

We assume the readers of this tutorial have Java SDK version 1.8.0_66 installed on their system.

Step 2: Download Gradle Build File

Download the latest version of Gradle from the Download Gradle link. In the reference page, click on the **Complete Distribution** link. This step is common for any platform. For this you will get the complete distribution file into your Downloads folder.

Step 3: Set Up Environment for Gradle

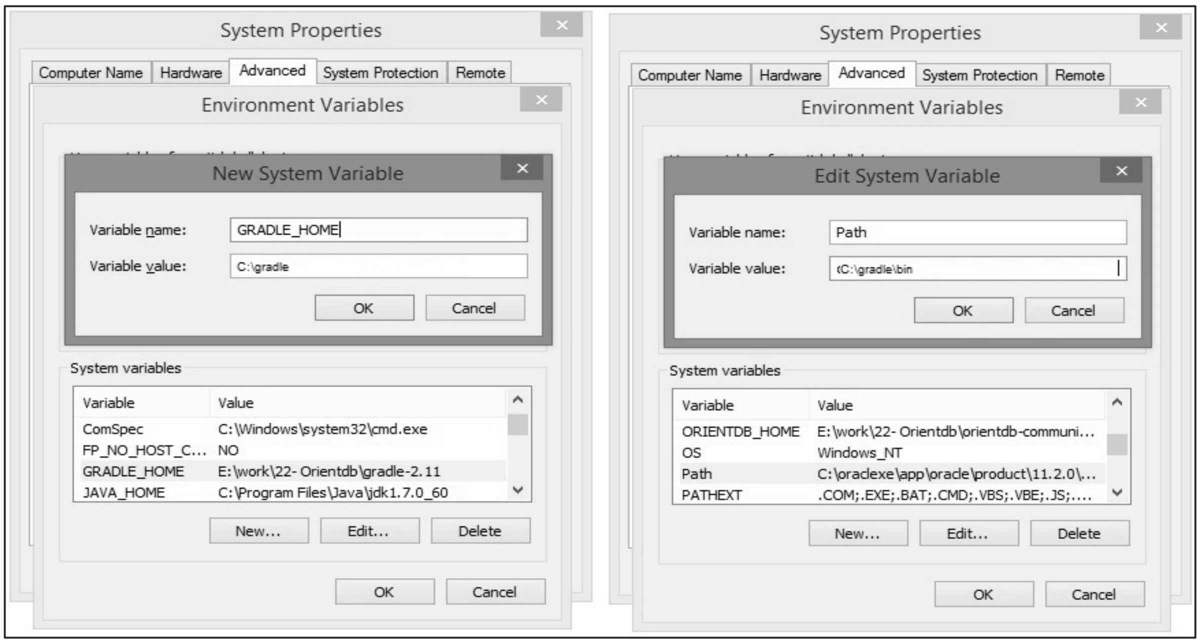
Setting up the environment means we have to extract the distribution file and copy the library files into proper location. Setting up **GRADLE_HOME** and **PATH** environmental variables.

This step is platform dependent.

In Windows

Extract the downloaded zip file named **gradle-2.11-all.zip** and copy the distribution files from **Downloads\gradle-2.11\** to **C:\gradle\** location.

Later, add the **C:\gradle** and **C:\gradle\bin** directories to the **GRADLE_HOME** and **PATH** system variables. Right-click on My Computer -> Click properties -> Advanced system settings -> Environment variables. There you will find a dialog box for creating and editing system variables. Click 'New' button for creating GRADLE_HOME variable (follow the left side screenshot). Click 'Edit' for editing the existing Path system variable (follow the right side screenshot). The process is shown in the following screenshots.



New System Variable

Variable name: GRADLE_HOME

Variable value: C:\gradle

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
GRADLE_HOME	E:\work\22- Orientdb\gradle-2.11
JAVA_HOME	C:\Program Files\Java\jdk1.7.0_60

Edit System Variable

Variable name: Path

Variable value: ;C:\gradle\bin

System variables

Variable	Value
ORIENTDB_HOME	E:\work\22- Orientdb\orientdb-communi...
OS	Windows_NT
Path	C:\oraclexe\app\oracle\product\11.2.0\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....

In Linux

Extract the downloaded zip file named **gradle-2.11-all.zip** then you will find an extracted file named **gradle-2.11**.

You can use the following to move the distribution files from **Downloads/gradle-2.11/** to **/opt/gradle/** location. Execute this operation from the Downloads directory.

```
$ sudo mv gradle-2.11 /opt/gradle
```

Edit the ~/.bashrc file and paste the following content to it and save it.

```
export ORIENT_HOME = /opt/gradle

export PATH = $PATH:
```

Execute the following command to execute **~/.bashrc** file.

```
$ source ~/.bashrc
```

Step 4: Verify the Gradle Installation

In Windows

You can execute the following command in command prompt.

```
C:\> gradle –v
```

Output: You will find the Gradle version.

```
------------------------------------------------------------

Gradle 2.11
```

```
------------------------------------------------------------
Build time:   2016-02-08 07:59:16 UTC
Build number: none
Revision:     584db1c7c90bdd1de1d1c4c51271c665bfcba978

Groovy:       2.4.4
Ant:          Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM:          1.7.0_60 (Oracle Corporation 24.60-b09)
OS:           Windows 8.1 6.3 amd64
```

In Linux

You can execute the following command in terminal.

```
$ gradle -v
```

Output: You will find the Gradle version.

```
------------------------------------------------------------
Gradle 2.11
------------------------------------------------------------

Build time:   2016-02-08 07:59:16 UTC
Build number: none
Revision:     584db1c7c90bdd1de1d1c4c51271c665bfcba978

Groovy:       2.4.4
Ant:          Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM:          1.7.0_60 (Oracle Corporation 24.60-b09)
OS:           Linux 3.13.0-74-generic amd64
```

3. Gradle – Build Script

Gradle builds a script file for handling two things; one is **projects** and another one is **tasks**. Every Gradle build represents one or more projects. A project represents a library JAR or a web application or it might represent a ZIP that assembled from the JARs produced by other projects. In simple words, a project is made up of different tasks. A task means a piece of work which a build performs. A task might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

Gradle uses **Groovy language** for writing scripts.

Writing Build Script

Gradle provides a Domain Specific Language (DSL), for describing builds. This uses the Groovy language to make it easier to describe a build. Each build script of Gradle is encoded using UTF-8, saved offline and named as build.gradle.

build.gradle

We are describing about tasks and projects by using a Groovy script. You can run a Gradle build using the Gradle command. This command looks for a file called **build.gradle**. Take a look at the following example which represents a small script that prints **tutorialspoint**. Copy and save the following script into a file named **build.gradle**. This build script defines a task name hello, which is used to print tutorialspoint string.

```
task hello {
    doLast {
        println 'tutorialspoint'
    }
}
```

Execute the following command in the command prompt. It executes the above script. You should execute this, where the build.gradle file is stored.

```
C:\> gradle –q hello
```

If the command is executed successfully, you will get the following output.

```
tutorialspoint
```

If you think task works similar to ANT’s target, then that’s right - Gradle task is equivalent to ANT target.

You can simplify this hello task by specifying a shortcut (represents a symbol **<<**) to the **doLast** statement. If you add this shortcut to the above task **hello** it will look like the following script.

```
task hello << {
  println 'tutorialspoint'
}
```

You can execute the above script using **gradle –q hello** command.

The Gradle script mainly uses two real Objects; one is Project Object and another one is Script Object.

Project Object: Each script describes about one or multiple projects. While in the execution, this script configures the Project Object. You can call some methods and use property in your build script which are delegated to the Project Object.

Script Object: Gradle takes script code into classes, which implements Script Interface and then executes. This means that of all the properties and methods declared by the script interface are available in your script.

The following table defines the list of **standard project properties**. All these properties are available in your build script.

Sr. No.	Name	Type	Default Value
1	project	Project	The Project instance
2	name	String	The name of the project directory.
3	path	String	The absolute path of the project.
4	description	String	A description for the project.
5	projectDir	File	The directory containing the build script.
6	buildDir	File	projectDir/build
7	group	Object	Unspecified
8	version	Object	Unspecified
9	ant	AntBuilder	An AntBuilder instance

How Gradle Uses Groovy

Gradle build scripts use the full length Groovy API. As a startup, take a look at the following examples. The following example explains about converting a string to upper case.

Copy and save the following code into **build.gradle** file.

```
task upper << {
    String expString = 'TUTORIALS point'
    println "Original: " + expString
    println "Upper case: " + expString.toUpperCase()}
```

Execute the following command in the command prompt. It executes the above given script. You should execute this, where the build.gradle file is stored.

```
C:\> gradle –q upper
```

If the command is executed successfully, you will get the following output.

```
Original: TUTORIALS point

Upper case: TUTORIALS POINT
```

The following example explains about printing the value of an implicit parameter ($it) for 4 times.

Copy and save the following code into **build.gradle** file.

```
task count << {
    4.times { print "$it " }
}
```

Execute the following command in the command prompt. It executes the above given script. You should execute this, where the build.gradle file is stored.

```
$ gradle –q count
```

If the command is executed successfully, you will get the following output.

```
0 1 2 3
```

Groovy language provides plenty of features. Following are some important features.

Groovy JDK Methods

Groovy adds lots of useful methods to the standard Java classes. For example, Iterable API from JDK implements an **each()** method which iterates over the elements of the Iterable Interface.

Copy and save the following code into **build.gradle** file.

```
task groovyJDK << {
String myName = "Marc";
myName.each() { println "${it}" };
}
```

Execute the following command in the command prompt. It executes the above given script. You should execute this, where the build.gradle file is stored.

```
C:\> gradle –q groovyJDK
```

If the command is executed successfully, you will get the following output.

```
M
a
r
c
```

Property Accessors

You can automatically accesses appropriate getter and setter methods of a particular property by specifying its reference.

The following snippet defines the syntaxes of getter and setter methods of a property **buildDir**.

```
// Using a getter method
println project.buildDir
println getProject().getBuildDir()

// Using a setter method
project.buildDir = 'target'
getProject().setBuildDir('target')
```

Optional Parentheses on Method Calls

Groovy contains a special feature in methods calling that is the parentheses are optional for method calling. This feature applies to Gradle scripting as well.

Take a look at the following syntax. That defines a method calling **systemProperty** of **test** object.

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

Closure as the Last Parameter of the Method

Gradle DSL uses closures in many places. Where the last parameter of a method is a closure, you can place the closure after the method call.

The following snippet defines the syntaxes Closures use as repositories() method parameters.

```
repositories {
    println "in a closure"
}
repositories() { println "in a closure" }
repositories({ println "in a closure" })
```

Default Imports for Gradle Scripts

Gradle automatically adds a set of import statements to the Gradle scripts. The following list shows you the default import packages to the Gradle script.

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.cache.*
import org.gradle.api.artifacts.component.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.ivy.*
import org.gradle.api.artifacts.maven.*
import org.gradle.api.artifacts.query.*
import org.gradle.api.artifacts.repositories.*
import org.gradle.api.artifacts.result.*
import org.gradle.api.component.*
import org.gradle.api.credentials.*
import org.gradle.api.distribution.*
import org.gradle.api.distribution.plugins.*
import org.gradle.api.dsl.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.initialization.*
import org.gradle.api.initialization.dsl.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.logging.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.announce.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.buildcomparison.gradle.*
import org.gradle.api.plugins.jetty.*
import org.gradle.api.plugins.osgi.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.scala.*
import org.gradle.api.plugins.sonar.*
import org.gradle.api.plugins.sonar.model.*
import org.gradle.api.publish.*
import org.gradle.api.publish.ivy.*
import org.gradle.api.publish.ivy.plugins.*
import org.gradle.api.publish.ivy.tasks.*
import org.gradle.api.publish.maven.*
import org.gradle.api.publish.maven.plugins.*
import org.gradle.api.publish.maven.tasks.*
import org.gradle.api.publish.plugins.*
import org.gradle.api.reporting.*
import org.gradle.api.reporting.components.*
import org.gradle.api.reporting.dependencies.*
import org.gradle.api.reporting.model.*
import org.gradle.api.reporting.plugins.*
import org.gradle.api.resources.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.ant.*
import org.gradle.api.tasks.application.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.incremental.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.scala.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.testing.junit.*
import org.gradle.api.tasks.testing.testng.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
import org.gradle.authentication.*
import org.gradle.authentication.http.*
import org.gradle.buildinit.plugins.*
import org.gradle.buildinit.tasks.*
import org.gradle.external.javadoc.*
import org.gradle.ide.cdt.*
import org.gradle.ide.cdt.tasks.*
import org.gradle.ide.visualstudio.*
import org.gradle.ide.visualstudio.plugins.*
import org.gradle.ide.visualstudio.tasks.*
import org.gradle.ivy.*
import org.gradle.jvm.*
import org.gradle.jvm.application.scripts.*
import org.gradle.jvm.application.tasks.*
import org.gradle.jvm.platform.*
import org.gradle.jvm.plugins.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.tasks.api.*
import org.gradle.jvm.test.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.assembler.*
import org.gradle.language.assembler.plugins.*
import org.gradle.language.assembler.tasks.*
import org.gradle.language.base.*
import org.gradle.language.base.artifact.*
import org.gradle.language.base.plugins.*
import org.gradle.language.base.sources.*
import org.gradle.language.c.*
import org.gradle.language.c.plugins.*
import org.gradle.language.c.tasks.*
import org.gradle.language.coffeescript.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.*
import org.gradle.language.java.artifact.*
import org.gradle.language.java.plugins.*
import org.gradle.language.java.tasks.*
import org.gradle.language.javascript.*
import org.gradle.language.jvm.*
import org.gradle.language.jvm.plugins.*
import org.gradle.language.jvm.tasks.*
import org.gradle.language.nativeplatform.*
import org.gradle.language.nativeplatform.tasks.*
import org.gradle.language.objectivec.*
import org.gradle.language.objectivec.plugins.*
import org.gradle.language.objectivec.tasks.*
import org.gradle.language.objectivecpp.*
import org.gradle.language.objectivecpp.plugins.*
import org.gradle.language.objectivecpp.tasks.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
import org.gradle.language.routes.*
import org.gradle.language.scala.*
import org.gradle.language.scala.plugins.*
import org.gradle.language.scala.tasks.*
import org.gradle.language.scala.toolchain.*
import org.gradle.language.twirl.*
import org.gradle.maven.*
import org.gradle.model.*
import org.gradle.nativeplatform.*
import org.gradle.nativeplatform.platform.*
import org.gradle.nativeplatform.plugins.*
import org.gradle.nativeplatform.tasks.*
import org.gradle.nativeplatform.test.*
import org.gradle.nativeplatform.test.cunit.*
import org.gradle.nativeplatform.test.cunit.plugins.*
import org.gradle.nativeplatform.test.cunit.tasks.*
import org.gradle.nativeplatform.test.googletest.*
import org.gradle.nativeplatform.test.googletest.plugins.*
import org.gradle.nativeplatform.test.plugins.*
import org.gradle.nativeplatform.test.tasks.*
import org.gradle.nativeplatform.toolchain.*
import org.gradle.nativeplatform.toolchain.plugins.*
import org.gradle.platform.base.*
import org.gradle.platform.base.binary.*
import org.gradle.platform.base.component.*
import org.gradle.platform.base.plugins.*
import org.gradle.platform.base.test.*
import org.gradle.play.*
import org.gradle.play.distribution.*
import org.gradle.play.platform.*
import org.gradle.play.plugins.*
import org.gradle.play.tasks.*
import org.gradle.play.toolchain.*
import org.gradle.plugin.use.*
import org.gradle.plugins.ear.*
import org.gradle.plugins.ear.descriptor.*
import org.gradle.plugins.ide.api.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.javascript.base.*
import org.gradle.plugins.javascript.coffeescript.*
import org.gradle.plugins.javascript.envjs.*
import org.gradle.plugins.javascript.envjs.browser.*
import org.gradle.plugins.javascript.envjs.http.*
import org.gradle.plugins.javascript.envjs.http.simple.*
import org.gradle.plugins.javascript.jshint.*
import org.gradle.plugins.javascript.rhino.*
import org.gradle.plugins.javascript.rhino.worker.*
import org.gradle.plugins.signing.*
import org.gradle.plugins.signing.signatory.*
import org.gradle.plugins.signing.signatory.pgp.*
import org.gradle.plugins.signing.type.*
import org.gradle.plugins.signing.type.pgp.*
import org.gradle.process.*
import org.gradle.sonar.runner.*
import org.gradle.sonar.runner.plugins.*
import org.gradle.sonar.runner.tasks.*
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.testkit.runner.*
import org.gradle.util.*
```

4. Gradle – Tasks

Gradle build script describes one or more Projects. Each project is made up of different tasks. A task is a piece of work which a build performs. The task might be compiling some classes, storing class files into separate target folder, creating JAR, generating Javadoc, or publishing some archives to repositories.

This chapter explains what is task and how to generate and execute a task.

Defining Tasks

Task is a keyword which is used to define a task into build script. Take a look at the following example which represents a task named **hello** that prints **tutorialspoint**. Copy and save the following script into a file named **build.gradle**. This build script defines a task named 'hello', which is used to print tutorialspoint string.

```
task hello {
    doLast {
        println 'tutorialspoint'
    }
}
```

Execute the following command in the command prompt. It executes the above script. You should execute this where the build.gradle file is stored.

```
C:\> gradle –q hello
```

If the command is executed successfully, you will get the following output.

```
tutorialspoint
```

You can simplify this hello task by specifying a shortcut (represents a symbol **<<**) to the **doLast** statement. If you add this shortcut to the above task **hello** it will look like the following script.

```
task hello << {
  println 'tutorialspoint'
}
```

You can execute the above script using **gradle –q hello** command.

Here are some variations in defining a task, take a look at it. The following example defines a task **hello**.

Copy and save the following code into **build.gradle** file.

```
task(hello) << {
    println "tutorialspoint"
}
```

Execute the following command in the command prompt. You should execute this, where the build.gradle file is stored.

```
C:\> gradle –q hello
```

If the command is executed successfully, you will get the following output.

```
tutorialspoint
```

You can also use strings for the task names. Take a look at the same hello example. Here we will use String as task.

Copy and save the following code into **build.gradle** file.

```
task('hello') << {
    println "tutorialspoint"
}
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle –q hello
```

If the command is executed successfully, you will get the following output.

```
tutorialspoint
```

You can also use alternative syntax for defining a task. That is using create() method to defining a task. Take a look at the same hello example.

Copy and save the following code into **build.gradle** file.

```
Task.create(name: 'hello') << {
    println "tutorialspoint"
}
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle –q hello
```

If the command is executed successfully, you will get the following output.

```
tutorialspoint
```

Locating Tasks

If you want to locate tasks that you defined in the build file, then you have to use respective standard project properties. That means each task is available as a property of the project, using the task name as the property name.

Take a look at the following code that accesses tasks as properties.

Copy and save the following code into **build.gradle** file.

```
task hello


println hello.name

println project.hello.name
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle -q hello
```

If the command is executed successfully, you will get the following output.

```
hello

hello
```

You can also use all the properties through the tasks collection.

Copy and save the following code into **build.gradle** file.

```
task hello


println tasks.hello.name

println tasks['hello'].name
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle -q hello
```

If the command is executed successfully, you will get the following output.

```
hello

hello
```

You can access tasks from any project using the task's path. For this you can call the getByPath() method with a task name, or a relative path, or an absolute path.

Copy and save the following code into **build.gradle** file.

```
project(':projectA') {
    task hello
}

task hello

println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle -q hello
```

If the command is executed successfully, you will get the following output.

```
:hello
:hello
:projectA:hello
:projectA:hello
```

Adding Dependencies to Tasks

You can make a task dependent on another task, which means when one task is done only then the other task will start. Each task is differentiated with a task name. Collection of task names is referred by its tasks collection. To refer to a task in another project, you should use path of the project as a prefix to the respective task name.

The following example adds a dependency from taskX to taskY.

Copy and save the following code into **build.gradle** file.

```
task taskX << {
    println 'taskX'
}

task taskY(dependsOn: 'taskX') << {
    println "taskY"
}
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle -q taskY
```

If the command is executed successfully, you will get the following output.

```
taskX
taskY
```

The above example is adding dependency on task by using its names. There is another way to achieve task dependency that is define the dependency using a Task object.

Let us take the same example of taskY being dependent on taskX but we are using task objects instead of task reference names.

Copy and save the following code into **build.gradle** file.

```
task taskY << {
    println 'taskY'
}

task taskX << {
  println 'taskX'
}

taskY.dependsOn taskX
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle -q taskY
```

If the command is executed successfully, you will get the following output.

```
taskX
taskY
```

There is another way to add dependency to the task, that is, by using closures. In this case, the task is released through the closure. If you use closure in the build script that should return a single task or collection of task objects. The following example adds a dependency from taskX to all the tasks in the project, whose name starts with **'lib'**.

Copy and save the following code into **build.gradle** file.

```
task taskX << {
    println 'taskX'
}

taskX.dependsOn {
    tasks.findAll { task -> task.name.startsWith('lib') }
}

task lib1 << {
    println 'lib1'
}

task lib2 << {
    println 'lib2'
}

task notALib << {
    println 'notALib'
}
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle -q taskX
```

If the command is executed successfully, you will get the following output.

```
lib1
lib2
taskX
```

Adding a Description to a Task

You can add a description to your task. This description is displayed when executing **Gradle tasks**. This is possible by using the description keyword.

Copy and save the following code into **build.gradle** file.

```
task copy(type: Copy) {
   description 'Copies the resource directory to the target directory.'
   from 'resources'
   into 'target'
   include('**/*.txt', '**/*.xml', '**/*.properties')
   println("description applied")
}
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle –q copy
```

If the command is executed successfully, you will get the following output.

```
description applied
```

Skipping Tasks

Skipping tasks can be done by passing a predicate closure. This is possible only if method of a task or a closure throwing a **StopExecutionException** before the actual work of a task is executed.

Copy and save the following code into **build.gradle** file.

```
task eclipse << {
  println 'Hello Eclipse'
}


// #1st approach - closure returning true, if the task should be executed,
false if not.
eclipse.onlyIf {
  project.hasProperty('usingEclipse')
}


// #2nd approach - alternatively throw an StopExecutionException() like this
eclipse.doFirst {
  if(!usingEclipse) {
    throw new StopExecutionException()
  }
}
```

Execute the following command in the command prompt. You should execute this where the build.gradle file is stored.

```
C:\> gradle –q copy
```

Task Structure

Gradle has different phases, when working with tasks. First of all, there is a configuration phase, where the code, which is specified directly in a task's closure, is executed. The configuration block is executed for every available task and not only for those tasks which are later actually executed.

After the configuration phase, the execution phase runs the code inside the **doFirst** or **doLast** closures of those tasks, which are actually executed.

5. Gradle – Dependency Management

Gradle build script defines a process to build projects; each project contains some dependencies and some publications. Dependencies means the things that support to build your project such as required JAR file from other projects and external JARs like JDBC JAR or Eh-cache JAR in the class path. Publications means the outcomes of the project, such as test class files and build files, like war files.

Almost all the projects are not self-contained. They need files build by other projects to compile and test the source files. For example, in order to use Hibernate in the project, you need to include some Hibernate JARs in the classpath. Gradle uses some special script to define the dependencies, which needs to be downloaded.

Gradle takes care of building and publishing the outcomes somewhere. Publishing is based on the task that you define. You might want to copy the files to the local directory, or upload them to a remote Maven or Ivy repository, or you might use the files from another project in the same multi-project build. The process of publishing is called as **publication**.

Declaring Your Dependencies

Gradle follows some special syntax to define dependencies. The following script defines two dependencies, one is Hibernate core 3.6.7 and second one is Junit with the version 4.0 and later. Take a look at the following code. Use this code in **build.gradle** file.

```
apply plugin: 'java'


repositories {
    mavenCentral()
}


dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: 
'3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

Dependency Configurations

Dependency configuration is nothing but defines a set of dependencies. You can use this feature to declare external dependencies, which you want to download from the web. This defines the following different standard configurations.

- **Compile:** The dependencies required to compile the production source of the project.

- **Runtime:** The dependencies required by the production classes at runtime. By default, also includes the compile time dependencies.

- **Test Compile:** The dependencies required to compile the test source of the project. By default, it includes compiled production classes and the compile time dependencies.

- **Test Runtime:** The dependencies required to run the tests. By default, it includes runtime and test compile dependencies.

External Dependencies

External dependencies is a type of dependency. This is a dependency on some files that is built outside the current build, and is stored in a repository of some kind, such as Maven central, corporate Maven or Ivy repository, or a directory in the local file system.

The following code snippet is to define the external dependency. Use this code in **build.gradle** file.

```
dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version:
'3.6.7.Final'
}
```

An external dependency is declaring external dependencies and the shortcut form looks like "group: name: version".

Repositories

While adding external dependencies. Gradle looks for them in a repository. A repository is just a collection of files, organized by group, name and version. By default, Gradle does not define any repositories. We have to define at least one repository explicitly. The following code snippet defines how to define maven repository. Use this code in **build.gradle** file.

```
repositories {
    mavenCentral()
}
```

Following code is to define remote maven. Use this code in **build.gradle** file.

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

Publishing Artifacts

Dependency configurations are also used to publish files. These published files are called artifacts. Usually, we use plug-ins to define artifacts. However, you do need to tell Gradle where to publish the artifacts. You can achieve this by attaching repositories to the upload archives task. Take a look at the following syntax for publishing Maven repository. While executing, Gradle will generate and upload a Pom.xml as per the project requirements. Use this code in **build.gradle** file.

```
apply plugin: 'maven'


uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

6. Gradle – Plugins

A plugin is nothing but a set of tasks, almost all useful tasks such as compiling tasks, setting domain objects, setting up source files, etc. are handled by plugins. Applying a plugin to a project, allows the plugin to extend the project's capabilities. Plugins can:

- Extend the basic Gradle model (e.g. add new DSL elements that can be configured).
- Configure the project according to conversions (e.g. add new tasks or configure sensible defaults).
- Apply specific configuration (e.g. add organizational repositories or enforce standards).

Types of Plugins

There are two types of plugins in Gradle, script plugins and binary plugins. Script plugins is an additional build script that gives a declarative approach to manipulating the build. This is typically used within a build. Binary plugins are the classes that implement the plugin interface and adopt a programmatic approach to manipulating the build. Binary plugins can reside with a build script, with the project hierarchy or externally in a plugin JAR.

Applying Plugins

Project.apply() API method is used to apply the particular plugin. You can use the same plugin for multiple times. There are two types of plugins one is script plugin and second is binary plugin.

Script Plugins

Script plugins can be applied from a script on the local filesystem or at a remote location. Filesystem locations are relative to the project directory, while remote script locations specifies HTTP URL. Take a look at the following code snippet. It is used to apply the **other.gradle** plugin to the build script. Use this code in **build.gradle** file.

```
apply from: 'other.gradle'
```

Binary Plugins

Each plugin is identified by a plugin id. Some core plugins use short names to apply it and some community plugins use fully qualified name for plugin id. Sometimes it allows to specify a class of plugin.

Take a look at the following code snippet. It shows how to apply Java plugin by using its type. Use this code in **build.gradle** file.

```
apply plugin: JavaPlugin
```

Take a look at the following code for applying core plugin using the short name. Use this code in **build.gradle** file.

```
plugins {
    id 'java'
}
```

Take a look at the following code for applying community plugin using the short name. Use this code in **build.gradle** file.

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

Writing Custom Plugins

While creating a custom plugin, you need to write an implementation of plugin. Gradle instantiates the plugin and calls the plugin instance using Plugin.apply() method. The following sample contains a greeting plugin, which adds a hello task to the project. Take a look at the following code. Use this code in **build.gradle** file.

```
apply plugin: GreetingPlugin

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') << {
            println "Hello from the GreetingPlugin"
        }
    }
}
```

Use the following code to execute the above script.

```
C:\> gradle -q hello
```

If the command is executed successfully, you will get the following output.

```
Hello from the GreetingPlugin
```

Getting Input from the Build

Most of the plugins need the configuration support from the build script. The Gradle project has an associated 'ExtensionContainer' object that helps to track all the setting and properties being passed to plugins.

Let's add a simple extension object to the project. Here, we add a greeting extension object to the project, which allows you to configure the greeting. Use this code in **build.gradle** file.

```
apply plugin: GreetingPlugin

greeting.message = 'Hi from Gradle'

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Add the 'greeting' extension object
        project.extensions.create("greeting", GreetingPluginExtension)
        // Add a task that uses the configuration
        project.task('hello') << {
            println project.greeting.message
        }
    }
}

class GreetingPluginExtension {
    def String message = 'Hello from GreetingPlugin'
}
```

Use the following code to execute the above script.

```
C:\> gradle -q hello
```

If the command is executed successfully, you will get the following output.

```
Hello from Gradle
```

In this example, GreetingPlugin is a plain old Groovy object with a field called 'message'. The extension object is added to the plugin list with the name greeting. This object then becomes available as a project property with the same name as the extension object.

Gradle adds a configuration closure for each extension object, so you can group the settings together. Take a look at the following code. Use this code in **build.gradle** file.

```
apply plugin: GreetingPlugin

greeting {
    message = 'Hi'
    greeter = 'Gradle'
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.extensions.create("greeting", GreetingPluginExtension)
        project.task('hello') << {
            println "${project.greeting.message} from 
${project.greeting.greeter}"
        }
    }
}

class GreetingPluginExtension {
    String message
    String greeter
}
```

Use the following code to execute the above script.

```
C:\> gradle -q hello
```

If the command is executed successfully, you will get the following output.

```
Hello from Gradle
```

Standard Gradle Plugins

There are different plugins which are included in the Gradle distribution.

Language Plugins

These plugins add support for various languages which can be compiled and executed in the JVM.

Plugin Id	Automatically Applies	Description
java	java-base	Adds Java compilation, testing, and bundling capabilities to a project. It serves as the basis for many of the other Gradle plugins.
groovy	java,groovy-base	Adds support for building Groovy projects.
scala	java,scala-base	Adds support for building Scala projects.
antlr	Java	Adds support for generating parsers using Antlr.

Incubating Language Plugins

These plugins add support for various languages.

Plugin Id	Automatically Applies	Description
assembler	-	Adds native assembly language capabilities to a project.
c	-	Adds C source compilation capabilities to a project.
cpp	-	Adds C++ source compilation capabilities to a project.
objective-c	-	Adds Objective-C source compilation capabilities to a project.
objective-cpp	-	Adds Objective-C++ source compilation capabilities to a project.
windows-resources	-	Adds support for including Windows resources in native binaries.

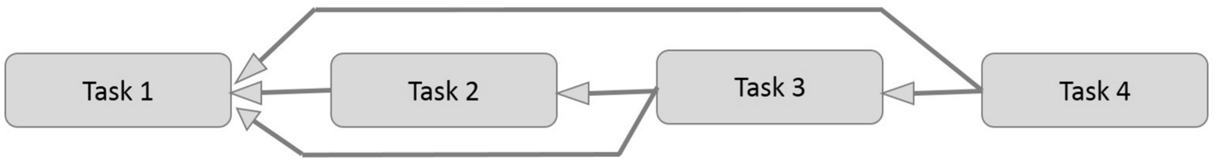
7. Gradle – Running a Build

Gradle provides a command line to execute build script. It can execute more than one task at a time. This chapter explains how to execute multiple tasks using different options.

Executing Multiple Tasks

You can execute multiple tasks from a single build file. Gradle can handle that build file using **gradle command**. This command will compile each task in the order that they are listed and execute each task along with the dependencies using different options.

Example: There are four tasks - task1, task2, task3, and task4. Task3 and task4 depends on task 1and task2. Take a look at the following diagram.



In the above four tasks are dependent on each other, represented with an arrow symbol. Take a look at the following code. Copy and paste it into **build.gradle** file.

```
task task1 << {
    println 'compiling source'
}

task task2(dependsOn: task1) << {
    println 'compiling unit tests'
}

task task3(dependsOn: [task1, task2]) << {
    println 'running unit tests'
}

task task4(dependsOn: [task1, task3]) << {
    println 'building the distribution'
}
```

You can use the following code for compiling and executing the above task.

```
C:\> gradle task4 test
```

If the command is executed successfully, you will get the following output.

```
:task1
compiling source
:task2
compiling unit tests
:task3
running unit tests
:task4
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

Excluding Tasks

While excluding a task from the execution you can use –x option along with the gradle command and mention the name of the task, which you want to exclude.

Use the following command to exclude task4 from the above script.

```
C:\> gradle task4 -x test
```

If the command is executed successfully, you will get the following output.

```
:task1
compiling source
:task4
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

Continuing the Build When a Failure Occurs

Gradle will abort the execution and fail the build as soon as any task fails. You can continue the execution even when a failure occurs. For this, you have to use –continue option with the gradle command. It handles each task separately along with their dependencies. The important point is, it will catch each encountered failure and report at the end of the execution of the build. Suppose if a task fails, then the dependent subsequent tasks also will not be executed.

Selecting Which Build to Execute

When you run the gradle command, it looks for a build file in the current directory. You can use the –b option to select a particular build file along with the absolute path. The following example shows selecting a project 'hello' from **myproject.gradle** file, which is located in the **subdir/**.

```
task hello << {
    println "using build file '$buildFile.name' in
'$buildFile.parentFile.name'."
}
```

You can use the following command to execute the above script.

```
C:\> gradle -q -b subdir/myproject.gradle hello
```

If the command is executed successfully, you will get the following output.

```
using build file 'myproject.gradle' in 'subdir'.
```

Obtaining Build Information

Gradle provides several built-in tasks for retrieving the information details regarding the task and the project. This can be useful to understand the structure and the dependencies of your build and for debugging problems. You can use project report plugin to add tasks to your project, which will generate these reports.

Listing Projects

You can list the project hierarchy of the selected project and their sub projects using **gradle –q projects** command. Here is the example, use the following command to list all the project in the build file.

```
C:\> gradle -q projects
```

Output:

```
------------------------------------------------------------
Root project
------------------------------------------------------------


Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation
```

```
To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

The report shows the description of each project, if specified. You can use the following command to specify the description. Paste it in the **build.gradle** file.

```
description = 'The shared API for the application'
```

Listing Tasks

You can list all the tasks which belong to the multiple projects by using the following command.

```
C:\> gradle -q tasks
```

If the command is executed successfully, you will get the following output.

```
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: dists

Build tasks
-----------
clean - Deletes the build directory (build)
dists - Builds the distribution
libs - Builds the JAR

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root
project 'projectReports'.
components - Displays the components produced by root project 'projectReports'.
[incubating]
dependencies - Displays all dependencies declared in root project
'projectReports'.
```

```
dependencyInsight - Displays the insight into a specific dependency in root
project 'projectReports'.
help - Displays a help message.
model - Displays the configuration model of root project 'projectReports'.
[incubating]
projects - Displays the sub-projects of root project 'projectReports'.
properties - Displays the properties of root project 'projectReports'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of
the displayed tasks may belong to subprojects).


To see all tasks and more detail, run gradle tasks --all


To see more detail about a task, run gradle help --task <task>
```

You can use the following command to display the information of all tasks.

```
C:\> gradle -q tasks --all
```

If the command is executed successfully, you will get the following output.

```
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------


Default tasks: dists


Build tasks
-----------
clean - Deletes the build directory (build)
api:clean - Deletes the build directory (build)
webapp:clean - Deletes the build directory (build)
dists - Builds the distribution [api:libs, webapp:libs]
    docs - Builds the documentation
api:libs - Builds the JAR
    api:compile - Compiles the source files
webapp:libs - Builds the JAR [api:libs]
    webapp:compile - Compiles the source files


Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
```

```
wrapper - Generates Gradle wrapper files. [incubating]


Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root
project 'projectReports'.
api:buildEnvironment - Displays all buildscript dependencies declared in
project ':api'.
webapp:buildEnvironment - Displays all buildscript dependencies declared in
project ':webapp'.
components - Displays the components produced by root project 'projectReports'.
[incubating]
api:components - Displays the components produced by project ':api'.
[incubating]
webapp:components - Displays the components produced by project ':webapp'.
[incubating]
dependencies - Displays all dependencies declared in root project
'projectReports'.
api:dependencies - Displays all dependencies declared in project ':api'.
webapp:dependencies - Displays all dependencies declared in project ':webapp'.
dependencyInsight - Displays the insight into a specific dependency in root
project 'projectReports'.
api:dependencyInsight - Displays the insight into a specific dependency in
project ':api'.
webapp:dependencyInsight - Displays the insight into a specific dependency in
project ':webapp'.
help - Displays a help message.
api:help - Displays a help message.
webapp:help - Displays a help message.
model - Displays the configuration model of root project 'projectReports'.
[incubating]
api:model - Displays the configuration model of project ':api'. [incubating]
webapp:model - Displays the configuration model of project ':webapp'.
[incubating]
projects - Displays the sub-projects of root project 'projectReports'.
api:projects - Displays the sub-projects of project ':api'.
webapp:projects - Displays the sub-projects of project ':webapp'.
properties - Displays the properties of root project 'projectReports'.
api:properties - Displays the properties of project ':api'.
webapp:properties - Displays the properties of project ':webapp'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of
the displayed tasks may belong to subprojects).
```

```
api:tasks - Displays the tasks runnable from project ':api'.
webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

Following is a list of commands with their descriptions.

Sr. No.	Command	Description
1	gradle –q help –task <task name>	Provides the usage information (such as path, type, description, group) about a specific task or multiple tasks.
2	gradle –q dependencies	Provides a list of dependencies of the selected project.
3	gradle -q api:dependencies --configuration <task name>	Provides the list of limited dependencies respective to configuration.
4	gradle –q buildEnvironment	Provides the list of build script dependencies.
5	gradle –q dependencyInsight	Provides an insight into a particular dependency.
6	Gradle –q properties	Provides the list of properties of the selected project.

8. Gradle – Build a Java Project

This chapter explains how to build a Java project using Gradle build file. First, we have to add Java plugin to the build script because it provides tasks to compile Java source code, run unit tests, create Javadoc and create a JAR file. Use the following line in **build.gradle** file.

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

Java Default Project Layout

Whenever you add a plugin to your build, it assume a certain setup of Java project (similar to Maven). Take a look at the following directory structure.

- src/main/java contains the Java source code
- src/test/java contains the Java tests

If you follow this setup, the following build file is sufficient to compile, test, and bundle a Java project.

To start the build, type the following command on the command line.

```
C:\> gradle build
```

SourceSets can be used to specify a different project structure. For example, the sources are stored in a **src** folder rather than in **src/main/java**. Take a look at the following directory structure.

```
apply plugin: 'java'
sourceSets {
   main {
      java {
         srcDir 'src'
         }
      }
   test {
      java {
         srcDir 'test'
         }
      }
}
```

init Task Execution

Gradle does not yet support multiple project templates. But it offers an **init** task to create the structure of a new Gradle project. Without additional parameters, this task creates a Gradle project, which contains the gradle wrapper files, a **build.gradle** and **settings.gradle** file.

When adding the **--type** parameter with **java-library** as value, a java project structure is created and the **build.gradle** file contains a certain Java template with Junit. Take a look at the following code for **build.gradle** file.

```
apply plugin: 'java'

repositories {
    jcenter()
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.12'
    testCompile 'junit:junit:4.12'
}
```

In the repositories section, it defines where to find the dependencies. **Jcenter** is for resolving your dependencies. Dependencies section is for providing information about external dependencies.

Specifying Java Version

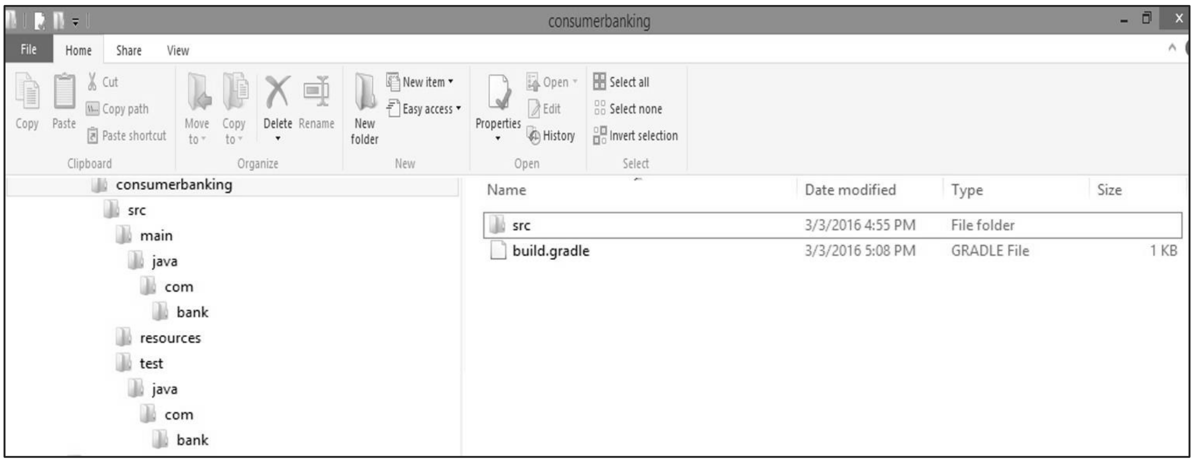
Usually, a Java project has a version and a target JRE on which it is compiled. The **version** and **sourceCompatibility** property can be set in the **build.gradle** file.

```
version = 0.1.0
sourceCompatibility = 1.8
```

If the artifact is an executable Java application, the **MANIFEST.MF** file must be aware of the class with the main method.

```
apply plugin: 'java'

jar {
  manifest {
    attributes 'Main-Class': 'com.example.main.Application'
  }
}
```

Let us consider an example. Create a directory structure as shown in the following screenshot.



consumerbanking
src
main
java
com
bank
resources
test
java
com
bank

Name	Date modified	Type	Size
src	3/3/2016 4:55 PM	File folder	
build.gradle	3/3/2016 5:08 PM	GRADLE File	1 KB

Copy the following Java code into App.java file and store it into **consumerbanking\src\main\java\com\bank** directory.

```
package com.bank;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Copy the following Java code into AppTset.java file and store into **consumerbanking\src\test\java\com\bank** directory.

```
package com.bank;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Copy the following code into build.gradle file and placed into **consumerbanking\** directory.

```
apply plugin: 'java'

repositories {
    jcenter()
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.12'
    testCompile 'junit:junit:4.12'
}


jar {
  manifest {
    attributes 'Main-Class': 'com.example.main.Application'
  }
}
```

To compile and execute the above script use the following commands.

```
consumerbanking\> gradle tasks
consumerbanking\> gradle assemble
consumerbanking\> gradle build
```

Check all the class files in the respective directories and check **consumerbanking\\build\\lib** folder for **consumerbanking.jar** file.

9. Gradle – Build a Groovy Project

This chapter explains how to compile and execute a Groovy project using **build.gradle** file.

The Groovy Plug-in

The Groovy plug-in for Gradle extends the Java plug-in and provides tasks for Groovy programs. You can use the following line for applying groovy plugin.

```
apply plugin: 'groovy'
```

The complete build script file is as follows. Copy the following code into **build.gradle** file.

```
apply plugin: 'groovy'



repositories {
  mavenCentral()
}
dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.4.5'
    testCompile 'junit:junit:4.12'
}
```

You can use the following command to execute the build script.

```
gradle build
```

Default Project Layout of Groovy Projects

The Groovy plugin assumes a certain setup of your Groovy project.

- src/main/groovy contains the Groovy source code
- src/test/groovy contains the Groovy tests
- src/main/java contains the Java source code
- src/test/java contains the Java tests

Check the respective directory from the build folder where **build.gradle** file is placed.

10. Gradle – Testing

The test task automatically detects and executes all unit tests in the test source set. It also generates a report once test execution is complete. JUnit and TestNG are the supported APIs.

The test task provides a **Test.getDebug()** method that can be set to launch to make the JVM wait for a debugger. Before proceeding to the execution, it sets the debugger post to **5005**.

Test Detection

The **Test Task** detects which classes are test classes by inspecting the compiled test classes. By default it scans all .class files. You can set custom includes / excludes, only those classes will be scanned. Depending on the test framework used (JUnit / TestNG), the test class detection uses different criteria.

When using JUnit, we scan for both JUnit 3 and 4 test classes. If any of the following criteria match, the class is considered to be a JUnit test class:

- Class or a super class extends TestCase or GroovyTestCase
- Class or a super class is annotated with @RunWith
- Class or a super class contain a method annotated with @Test
- When using TestNG, we scan for methods annotated with @Test

Note: The abstract classes are not executed. Gradle also scans the inheritance tree into jar files on the test classpath.

If you don't want to use test class detection, you can disable it by setting **scanForTestClasses** to false.

Test Grouping

JUnit and TestNG allows sophisticated groupings of test methods. For grouping, JUnit test classes and methods JUnit 4.8 introduces the concept of categories. The test task allows the specification of the JUnit categories you want to include and exclude.

You can use the following code snippet in build.gradle file to group test methods.

```
test {
    useJUnit {
        includeCategories 'org.gradle.junit.CategoryA'
        excludeCategories 'org.gradle.junit.CategoryB'
    }
}
```

Include and Exclude Particular Tests

The **Test** class has an **include** and **exclude** method. These methods can be used to specify, which tests should actually be run.

Run only the included tests:

```
test {
  include '**my.package.name/*'
}
```

Skip excluded tests:

```
test {
  exclude '**my.package.name/*'
}
```

The sample **build.gradle** file as shown in the following code shows different configuration options.

```
apply plugin: 'java' // adds 'test' task

test {
  // enable TestNG support (default is JUnit)
  useTestNG()

  // set a system property for the test JVM(s)
  systemProperty 'some.prop', 'value'

  // explicitly include or exclude tests
  include 'org/foo/**'
  exclude 'org/boo/**'

  // show standard out and standard error of the test JVM(s) on the console
  testLogging.showStandardStreams = true

  // set heap size for the test JVM(s)
  minHeapSize = "128m"
  maxHeapSize = "512m"

  // set JVM arguments for the test JVM(s)
  jvmArgs '-XX:MaxPermSize=256m'

  // listen to events in the test execution lifecycle
  beforeTest { descriptor ->
     logger.lifecycle("Running test: " + descriptor)
  }


  // listen to standard out and standard error of the test JVM(s)
  onOutput { descriptor, event ->
     logger.lifecycle("Test: " + descriptor + " produced standard out/err: " +
event.message )
  }
}
```

You can use the following command syntax to execute some test task.

```
gradle <someTestTask> --debug-jvm
```

11. Gradle – Multi-project Build

Gradle can handle smallest and largest projects easily. Small projects have a single build file and a source tree. It is very easy to digest and understand a project that has been split into smaller, inter-dependent modules. Gradle perfectly supports this scenario that is multi-project build.

Structure for Multi-project Build

Such builds come in all shapes and sizes, but they do have some common characteristics:

- A settings.gradle file in the root or master directory of the project.
- A build.gradle file in the root or master directory.
- Child directories that have their own *.gradle build files (some multi-project builds may omit child project build scripts).

For listing all the projects in the build file, you can use the following command.

```
C:\> gradle -q projects
```

If the command is executed successfully, you will get the following output.

```
------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

The report shows the description of each project, if specified. You can use the following command to specify the description. Paste it in the **build.gradle** file.

```
description = 'The shared API for the application'
```

Specifying a General Build Configuration

In a **build.gradle** file in the root_project, general configurations can be applied to all projects or just to the sub projects.

```
allprojects {
   group = 'com.example.gradle'
   version = '0.1.0'
}

subprojects {
   apply plugin: 'java'
   apply plugin: 'eclipse'
}
```

This specifies a common **com.example.gradle** group and the **0.1.0** version to all projects. The **subprojects** closure applies common configurations for all sub projects, but not to the root project, like the **allprojects** closure does.

Project Specific Configurations and Dependencies

The core **ui** and **util** subprojects can also have their own **build.gradle** file, if they have specific needs, which are not already applied by the general configuration of the root project.

For instance, the ui project usually has a dependency to the core project. So the ui project needs its own **build.gradle** file to specify this dependency.

```
dependencies {
   compile project(':core')
   compile 'log4j:log4j:1.2.17'
}
```

Project dependencies are specified with the project method.

12. Gradle – Deployment

Gradle offers several ways to deploy build artifacts repositories. When deploying signatures for your artifacts to a Maven repository, you will also want to sign the published POM file.

Using the Maven-publish Plugin

Gradle provides **maven-publish** plugin by default. It is used to publish the gradle script. Take a look at the following code.

```
apply plugin: 'java'
apply plugin: 'maven-publish'

publishing {
  publications {
    mavenJava(MavenPublication) {
      from components.java
    }
  }

  repositories {
    maven {
      url "$buildDir/repo"
    }
  }
}
```

There are several publish options, when the **Java** and the **maven-publish** plugin is applied. Take a look at the following code, it will deploy the project into a remote repository.

```
apply plugin: 'groovy'
apply plugin: 'maven-publish'

group 'workshop'
version = '1.0.0'

publishing {
  publications {
    mavenJava(MavenPublication) { from components.java }
  }

  repositories {
    maven {
      // default credentials for a nexus repository manager
      credentials {
        username 'admin'
        password 'admin123'
      }
      // url to the releases maven repository
      url "http://localhost:8081/nexus/content/repositories/releases/"
    }
  }
}
```

Converting a Project from Maven to Gradle

There is a special command for converting Apache Maven **pom.xml** files to Gradle build files, if all used Maven plug-ins are known to this task.

In this section, the following **pom.xml** maven configuration will be converted to a Gradle project. Take a look at 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.example.app</groupId>
  <artifactId>example-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

You can use the following command on the command line that results in the following Gradle configuration.

```
C:\> gradle init --type pom
```

The **init** task depends on the wrapper task so that a Gradle wrapper is created.

The resulting **build.gradle** file looks similar to the following.

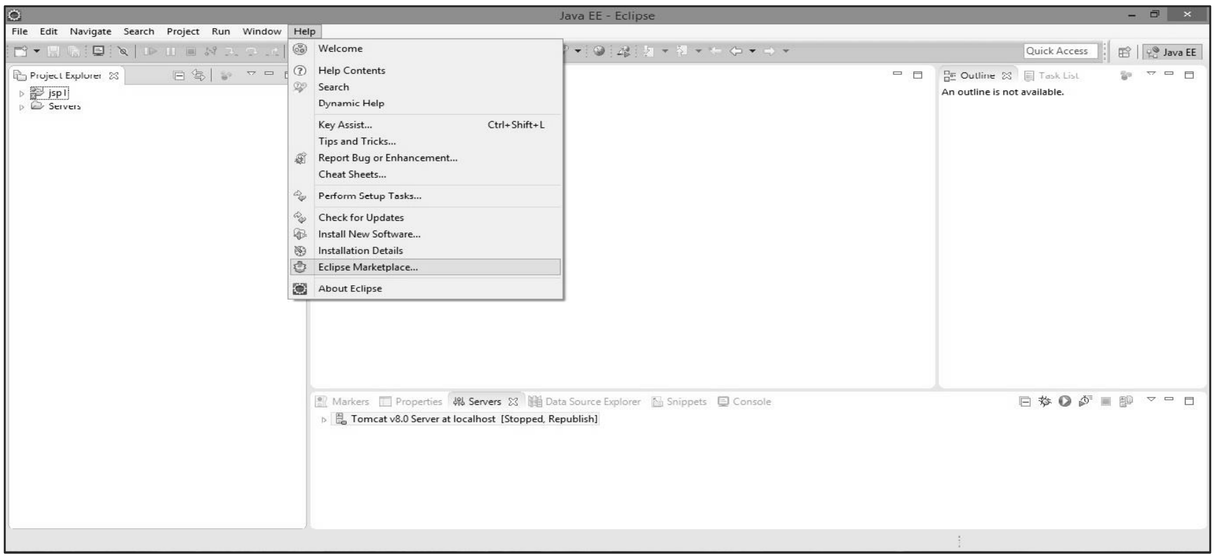
```
apply plugin: 'java'
apply plugin: 'maven'

group = 'com.example.app'
version = '1.0.0-SNAPSHOT'

description = """"""

sourceCompatibility = 1.5
targetCompatibility = 1.5

repositories {
     maven { url "http://repo.maven.apache.org/maven2" }
}

dependencies {
    testCompile group: 'junit', name: 'junit', version:'4.11'
}
```

13. Gradle – Eclipse Integration

This chapter explains the integration of Eclipse and Gradle. Following are the steps to add Gradle plugin to Eclipse.

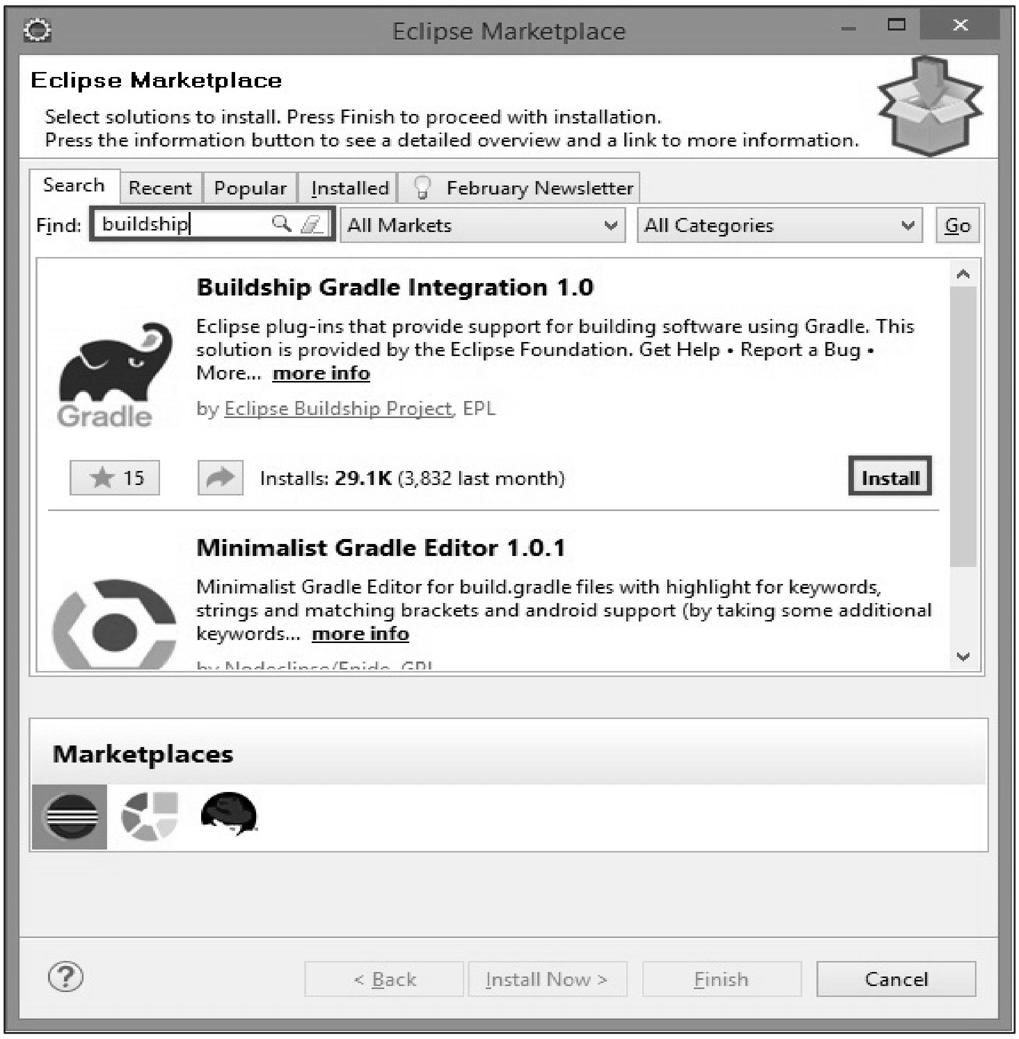
Step 1: Open Eclipse Marketplace

Open the eclipse which is installed in your system. Go to help -> click EclipseMarketplace as shown in the following screenshot.



Step 2: Install Buildship Plugin

Click Eclipse Marketplace, there you will find the following screenshot. On the left search bar, type **buildship**. Buildship is a Gradle integration plugin. When you find buildship on your screen, click Install button present on the right side of the screen as shown in the following screenshot.



Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

Buildship Gradle Integration 1.0

Eclipse plug-ins that provide support for building software using Gradle. This solution is provided by the Eclipse Foundation. Get Help • Report a Bug • More... **more info**

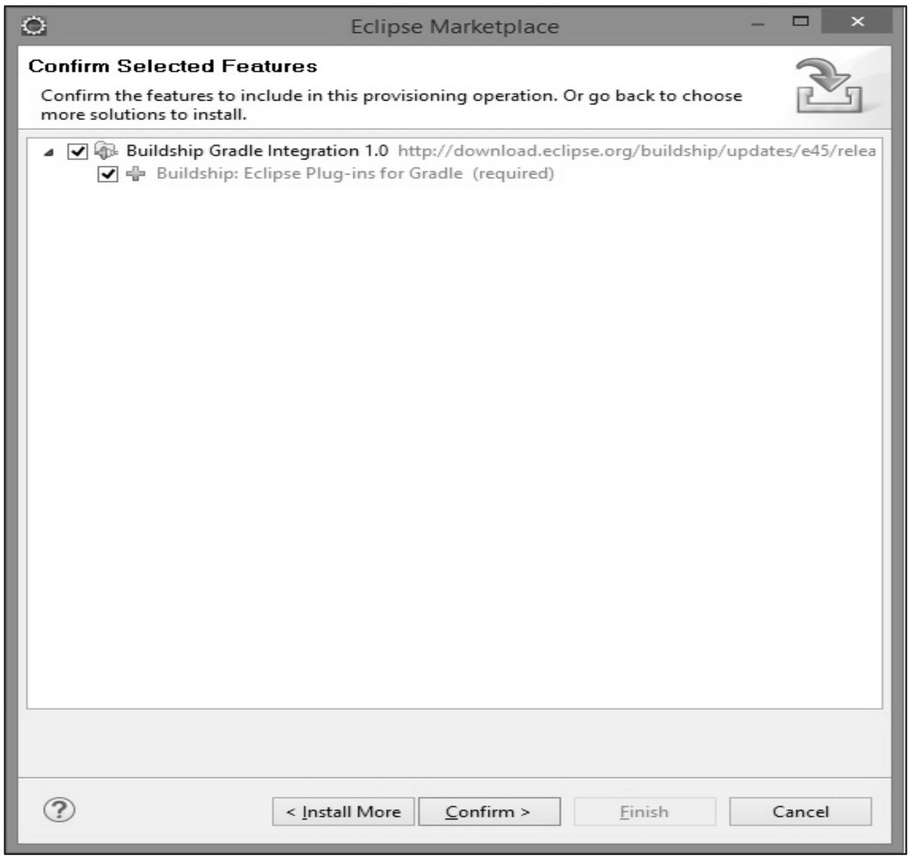
by Eclipse Buildship Project, EPL

Installs: **29.1K** (3,832 last month)

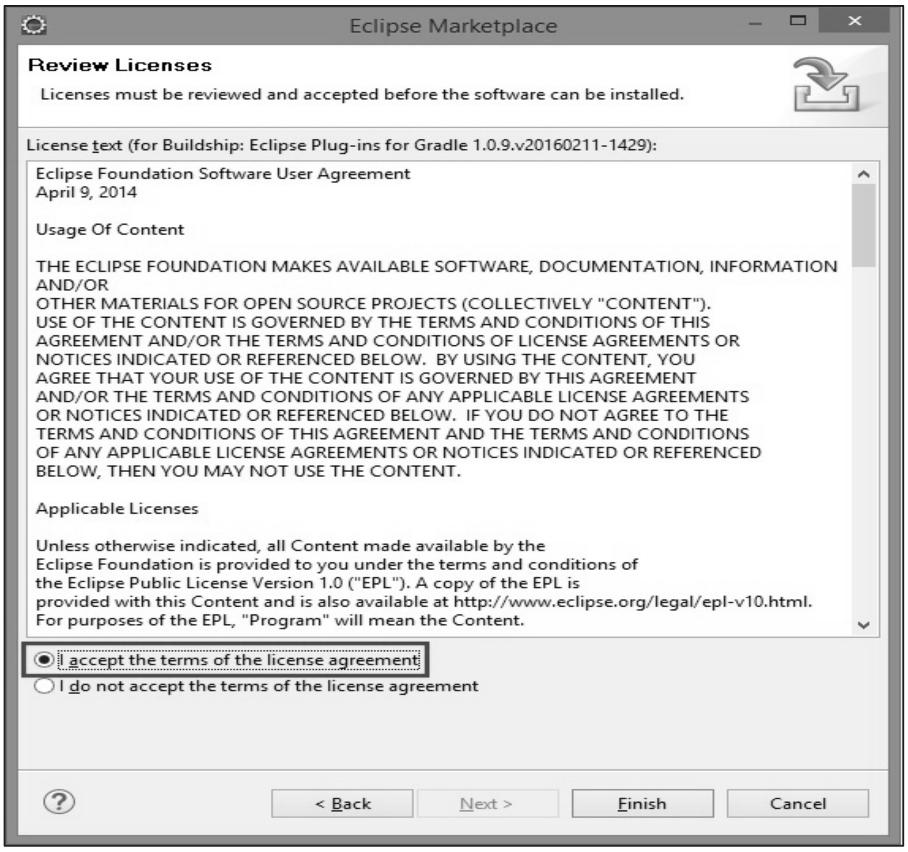
Minimalist Gradle Editor 1.0.1

Minimalist Gradle Editor for build.gradle files with highlight for keywords, strings and matching brackets and android support (by taking some additional keywords... **more info**

You will find the following screenshot. There you need to confirm the software installation by clicking the confirm button. Take a look at the following screenshot.


Eclipse Marketplace
Confirm Selected Features
Confirm the features to include in this provisioning operation. Or go back to choose more solutions to install.
Buildship Gradle Integration 1.0 http://download.eclipse.org/buildship/updates/e45/relea
Buildship: Eclipse Plug-ins for Gradle (required)
< Install More
Confirm >
Finish
Cancel

Click 'I accept the terms of the license agreement' in the following screen and then click 'Finish'. Take a look at the following screenshot.


Eclipse Marketplace
Review Licenses
Licenses must be reviewed and accepted before the software can be installed.

License text (for Buildship: Eclipse Plug-ins for Gradle 1.0.9.v20160211-1429):

Eclipse Foundation Software User Agreement
April 9, 2014

Usage Of Content

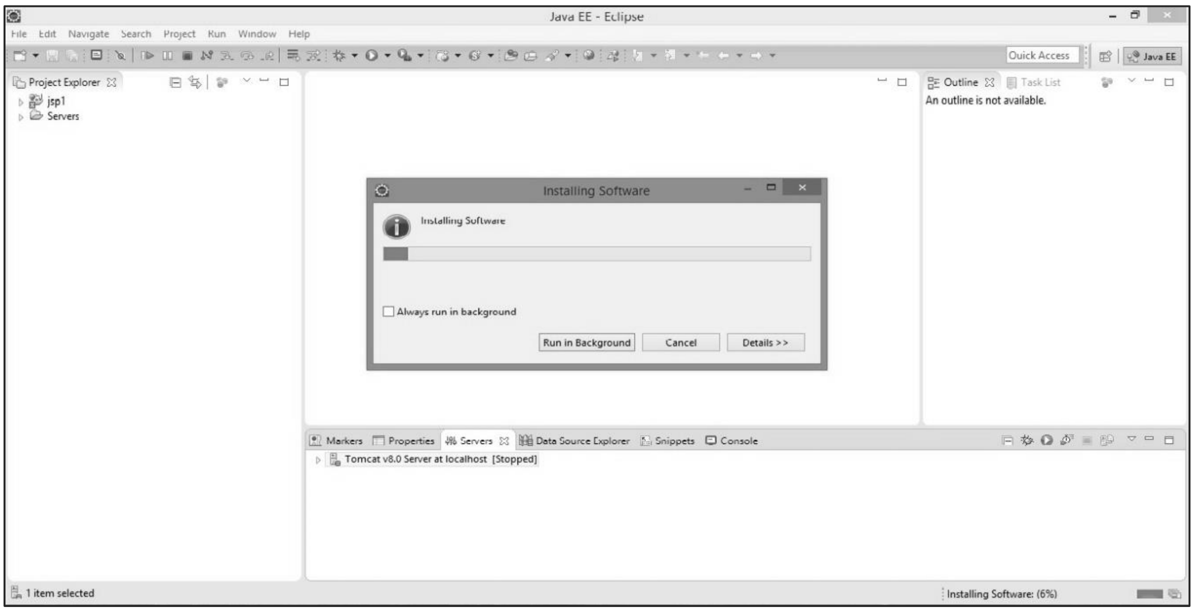
THE ECLIPSE FOUNDATION MAKES AVAILABLE SOFTWARE, DOCUMENTATION, INFORMATION AND/OR
OTHER MATERIALS FOR OPEN SOURCE PROJECTS (COLLECTIVELY "CONTENT").
USE OF THE CONTENT IS GOVERNED BY THE TERMS AND CONDITIONS OF THIS AGREEMENT AND/OR THE TERMS AND CONDITIONS OF LICENSE AGREEMENTS OR NOTICES INDICATED OR REFERENCED BELOW. BY USING THE CONTENT, YOU AGREE THAT YOUR USE OF THE CONTENT IS GOVERNED BY THIS AGREEMENT AND/OR THE TERMS AND CONDITIONS OF ANY APPLICABLE LICENSE AGREEMENTS OR NOTICES INDICATED OR REFERENCED BELOW. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT AND THE TERMS AND CONDITIONS OF ANY APPLICABLE LICENSE AGREEMENTS OR NOTICES INDICATED OR REFERENCED BELOW, THEN YOU MAY NOT USE THE CONTENT.

Applicable Licenses

Unless otherwise indicated, all Content made available by the Eclipse Foundation is provided to you under the terms and conditions of the Eclipse Public License Version 1.0 ("EPL"). A copy of the EPL is provided with this Content and is also available at http://www.eclipse.org/legal/epl-v10.html.
For purposes of the EPL, "Program" will mean the Content.

[✓ I accept the terms of the license agreement]
I do not accept the terms of the license agreement

It will take some time to install. Take a look at the following screenshot.



Installing Software

Always run in background

An outline is not available.

Tomcat v8.0 Server at localhost [Stopped]

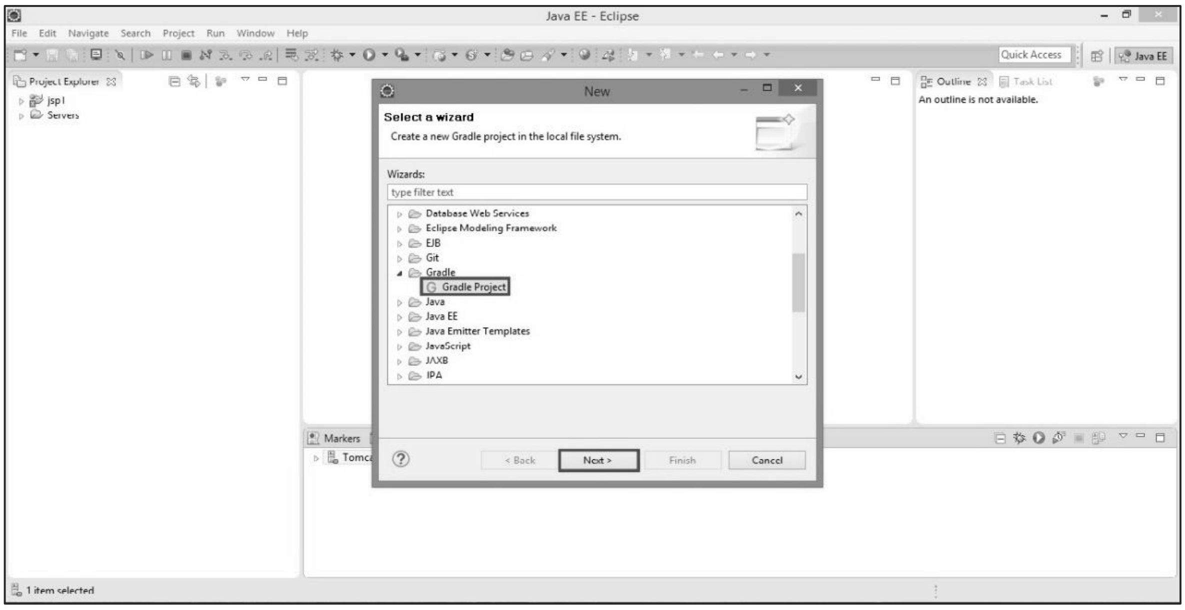
1 item selected

Installing Software: (6%)

After that, it will ask for restarting Eclipse. There you will select **Yes**.

Step 3: Verifying Gradle Plugin

While verifying, we will create a new project by following the given procedure. In the eclipse, go to file -> click New-> click Other projects. The following screen pops up. Select Gradle Project and click Next.



Select a wizard

Create a new Gradle project in the local file system.

Wizards:

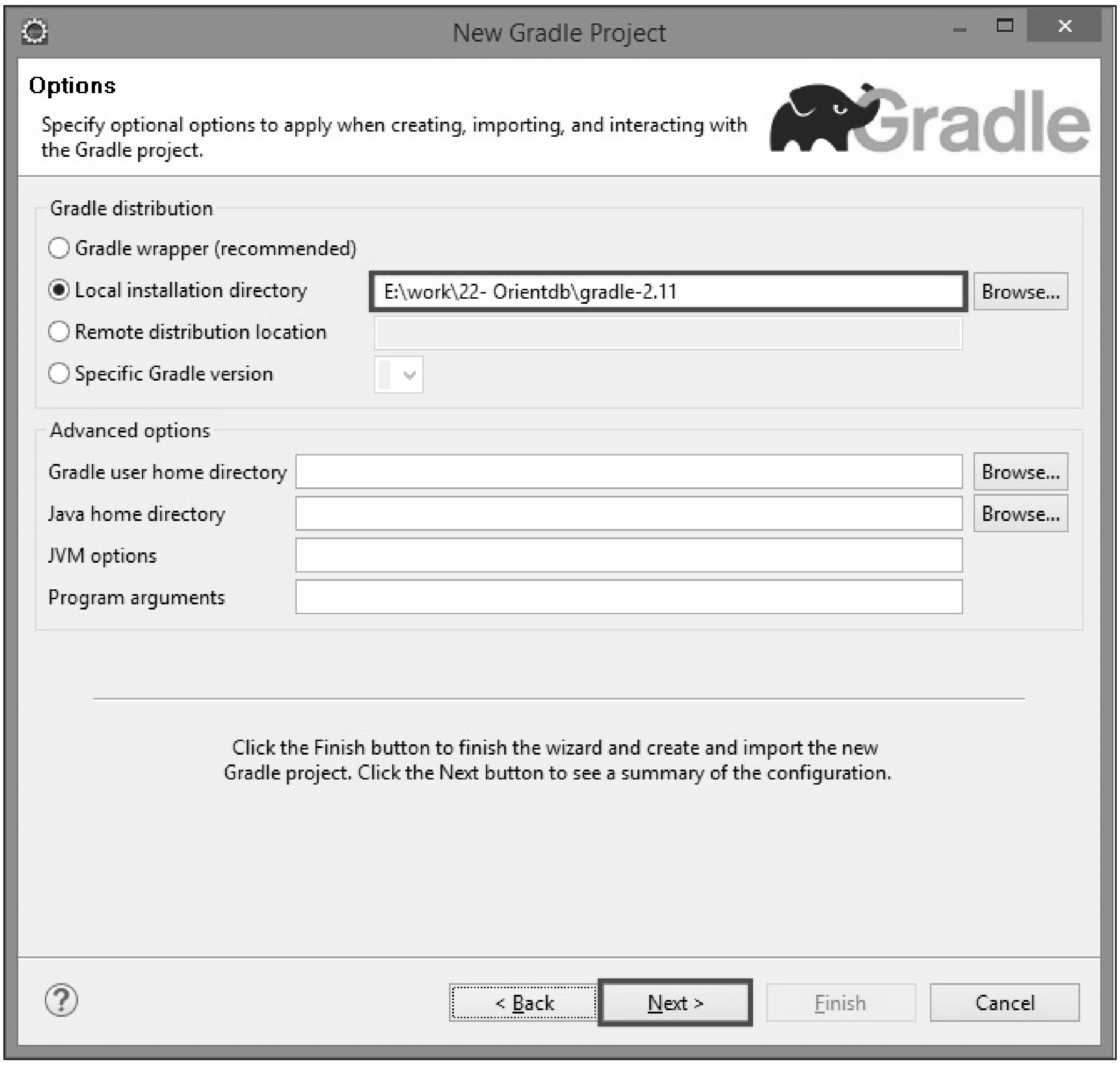
type filter text

- Database Web Services
- Eclipse Modeling Framework
- EJB
- Git
- Gradle
  - Gradle Project
- Java
- Java EE
- Java Emitter Templates
- JavaScript
- JAXB
- JPA

An outline is not available.

1 item selected

After clicking the Next button, the following screen pops up. On the screen, you will have to provide the Gradle home directory path of local file system and then click Next button as shown in the following screenshot.



Options

Specify optional options to apply when creating, importing, and interacting with the Gradle project.

Gradle distribution

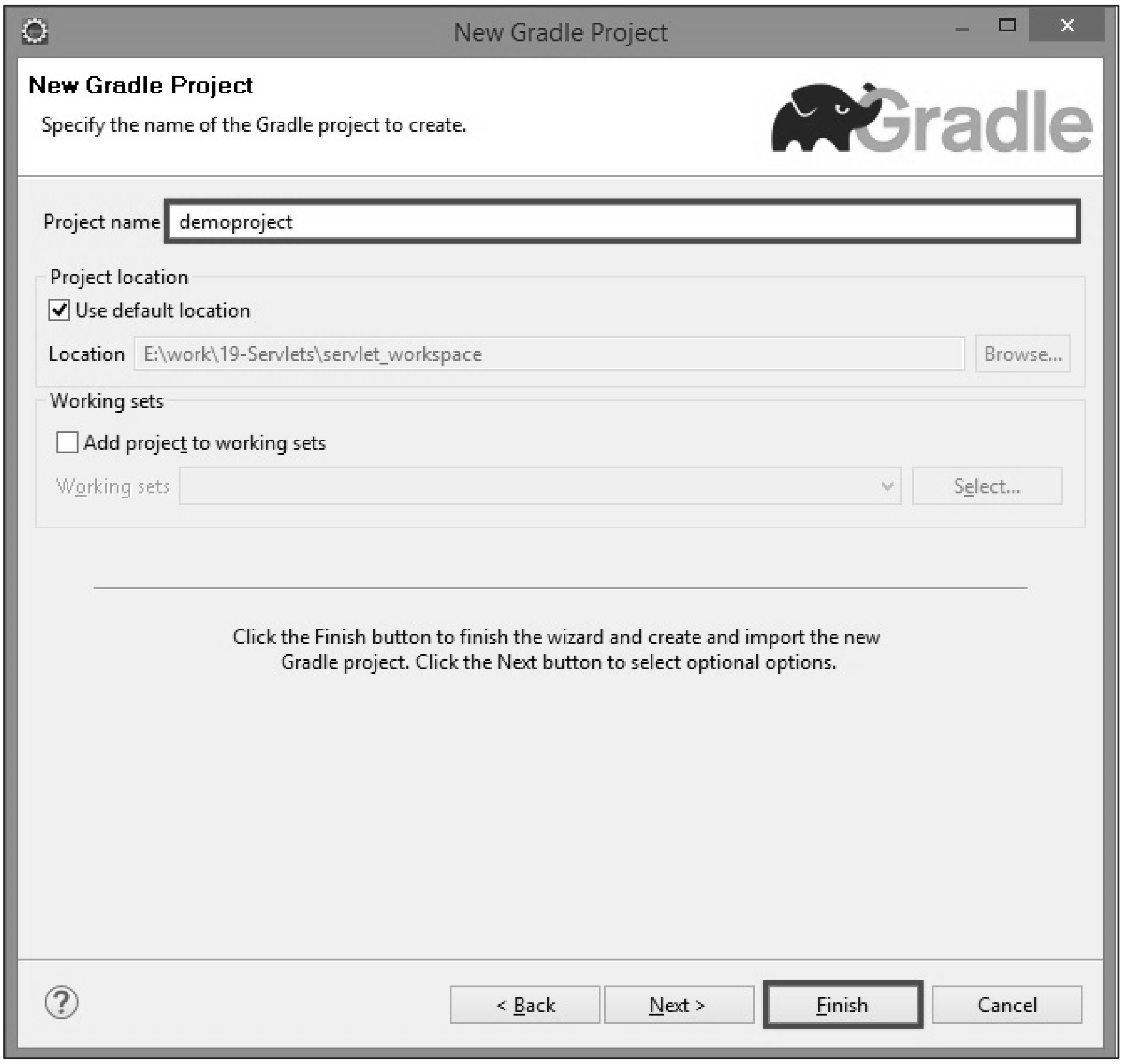
- ◯ Gradle wrapper (recommended)
- ● Local installation directory: E:\work\22- Orientdb\gradle-2.11 [Browse...]
- ◯ Remote distribution location
- ◯ Specific Gradle version

Advanced options

Gradle user home directory	[Browse...]
Java home directory	[Browse...]
JVM options	
Program arguments	

Click the Finish button to finish the wizard and create and import the new Gradle project. Click the Next button to see a summary of the configuration.

Take a look at the following screenshot. You will have to provide the name for Gradle project. In this tutorial, we are using the name **demoproject.** Click Finish button.



New Gradle Project

New Gradle Project

Specify the name of the Gradle project to create.

Project name: demoproject

Project location

[✓] Use default location

Location: E:\work\19-Servlets\servlet_workspace

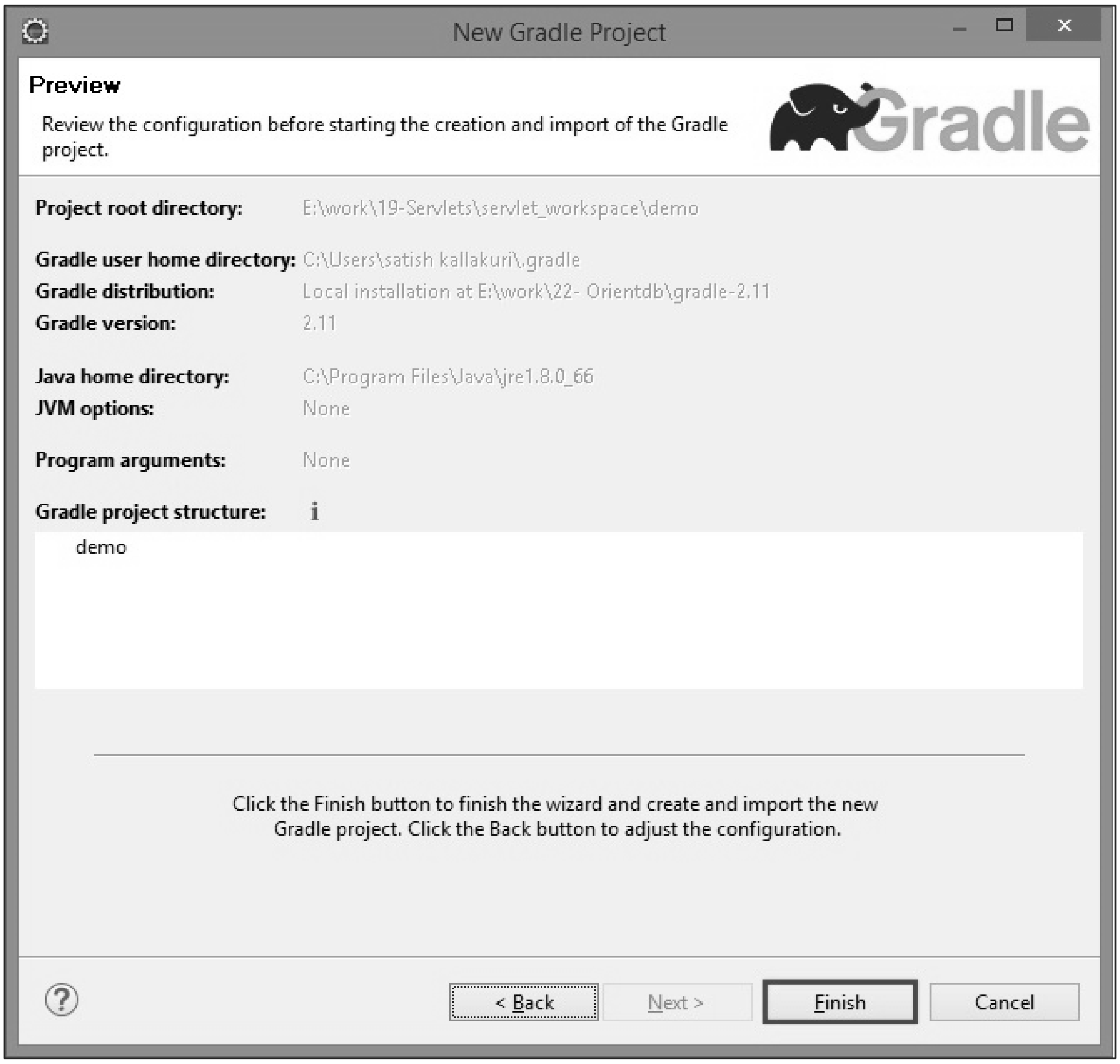
Working sets

[ ] Add project to working sets

Working sets:

Click the Finish button to finish the wizard and create and import the new Gradle project. Click the Next button to select optional options.

In the following screenshot, you will need to confirm the project. For which you will have to click Finish button.


New Gradle Project

Preview

Review the configuration before starting the creation and import of the Gradle project.

Project root directory: E:\work\19-Servlets\servlet_workspace\demo

Gradle user home directory: C:\Users\satish kallakuri\.gradle
Gradle distribution: Local installation at E:\work\22- Orientdb\gradle-2.11
Gradle version: 2.11

Java home directory: C:\Program Files\Java\jre1.8.0_66
JVM options: None

Program arguments: None

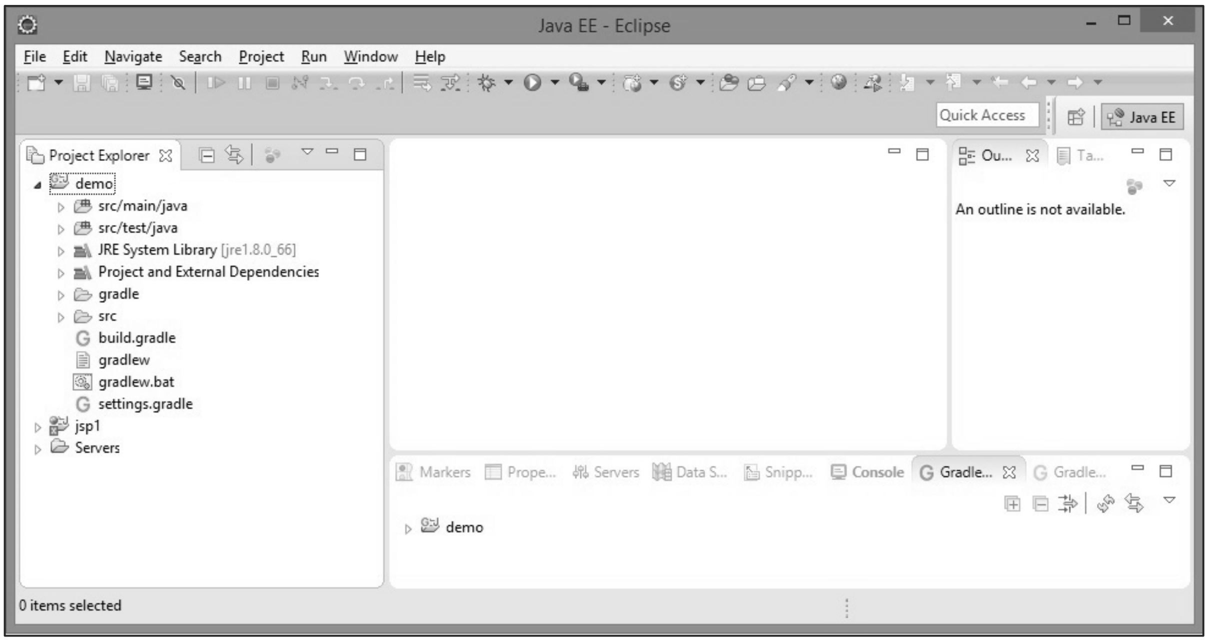
Gradle project structure:

demo

Click the Finish button to finish the wizard and create and import the new Gradle project. Click the Back button to adjust the configuration.

Step 4: Verifying Directory Structure

After successful installation of Gradle plugin, please check the demo project directory structure for the default files and folders as shown in the following screenshot.


Java EE - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java EE
Project Explorer
demo
src/main/java
src/test/java
JRE System Library [jre1.8.0_66]
Project and External Dependencies
gradle
src
build.gradle
gradlew
gradlew.bat
settings.gradle
jsp1
Servers
Ou...
Ta...
An outline is not available.
Markers
Prope...
Servers
Data S...
Snipp...
Console
Gradle...
Gradle...
demo
0 items selected