GRADLE

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

About the Tutorial

Gradle is an open source and advanced build automation tool. It builds up on ANT, Maven and Ivy repositories and supports groovy based Domain Specific Language (DSL) over the XML. In this tutorial, you will learn about different tasks, plugins with regards to gradle. Moreover, how to build a JAVA project and Groovy project with the help of gradle is also explained in detail.

Audience

This tutorial is designed for software professionals who are willing to learn Gradle build tool in simple and easy steps. It will be useful for all those enthusiasts, who are interested in working on multi-language software development.

Prerequisites

Gradle is groovy based build automation tool. Before you begin this tutorial, we expect that you have knowledge about JAVA and Groovy programming languages. You can refer to the tutorials related to JAVA and Groovy on our website for detailed information.

Copyright & Disclaimer

© Copyright 2020 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial i

Audience i

Prerequisites i

Copyright & Disclaimer i

Table of Contents ii

1. Gradle – Overview 1

History of Gradle 1

Features of Gradle 1

Why Groovy for Gradle? 3

2. Gradle — Installation 4

Prerequisites to install Gradle 4

3. Gradle — Build Script 8

Writing Build Script 8

Groovy Basics 10

Features of Groovy 11

Default Imports 12

4. Gradle — Tasks 18

Defining Tasks 18

Locating Tasks 20

Adding Dependencies to Tasks 21

Adding a Description 23

Skipping Tasks 24

Task Structure 25

5. Gradle — Dependency Management 26

Declaring Your Dependencies 26

Dependency Configurations 26

External Dependencies 27

Repositories 27

Publishing Artifacts 28

6. Gradle — Plugins 29

Types of Plugins 29

Applying Plugins 29

Writing Custom Plugins 30

Getting Input from the Build 31

Standard Gradle Plugins 33

7. Gradle - Running a Build 34

Executing Multiple Tasks 34

Excluding Tasks 35

Continuing the Build 35

Selecting Build to Execute 36

Obtaining Build Information 36

8. Gradle—Build a JAVA Project 41

Java Default Project Layout 41

init Task Execution 42

Specifying Java Version 42

9. Gradle — Build a Groovy Project 45

The Groovy Plug-in 45

Default Project Layout 45

10. Gradle — Testing 46

Test Detection 46

Test Grouping 46

Include and Exclude Tests 47

11. Gradle — Multi-Project Build 49

Structure for Multi-project Build 49

tutorialspoint
SIMPLYEASYLEARNING

General Build Configuration 50

Configurations and Dependencies.. 50

12. Gradle — Deployment 51

Maven-publish Plugin 51

Converting from Maven to Gradle 52

13. Gradle — Eclipse Integration.. 55

tutorialspoint
SIMPLYEASYLEARNING

1. Gradle – Overview

In this chapter, we will understand why was there a need to develop Gradle, what are its features and why Groovy programming language was used to develop Gradle.

History of Gradle

Ant and Maven shared considerable success in the JAVA marketplace. Ant was the first build tool released in 2000 and it was developed 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 drawbacks of Ant include:

- XML is used as a format to write the build scripts.
- Being hierarchical is not good for procedural programming, and
- XML is relatively unmanageable.

Maven was introduced in 2004. It came with lot of improvement than ANT. It was able to change its structure and XML could be used for writing build specifications. Maven relied on the conventions and was able to download the dependencies over the network.

The main benefits of Maven include:

- Life cycle of Maven, while following the same life cycle for multiple projects continuously.

Some problems faced by Maven with regards to dependency management include:

- It does not handle the conflicts between versions of the same library.
- Complex customised build scripts are difficult to write in Maven, as compared to writing the build scripts in ANT.

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

Features of Gradle

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.
- It provides the declarative language elements. Those elements also provide build-by-convention support for Java, Groovy, OSGI, Web and Scala.

Language for dependency based programming

The declarative language lies on a top of a general purpose task graph, which can be fully supported in the build.

Structure your build

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

Deep API

By using this API, you can monitor and customise its configuration and execution behavior to the core.

Gradle scales

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

Multi-project builds

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

Different ways to manage your builds

Gradle supports different strategies to manage your dependencies.

Gradle is the first build integration tool

Gradle is fully supported for your 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. 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 the 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 are written in Groovy programming language. The whole design of Gradle is oriented towards being used as a language and not as a rigid framework. Groovy

tutorialspoint
SIMPLYEASYLEARNING

allows you to write your own script with some abstractions. The whole Gradle API is fully designed in Groovy language.

Why Groovy for Gradle?

The complete Gradle API is designed using Groovy language. This is an advantage of an internal DSL over XML. Gradle is a general purpose build tool and 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 the team members.

Languages like Python, Groovy or Ruby are better for build framework. The reason for choosing Groovy is, because, it offers by far the greatest transparency for people using Java. The base syntax of Groovy is same as Java and Groovy provides much more benefits for its users.

tutorialspoint
SIMPLYEASYLEARNING

2. Gradle — Installation

Gradle is a build tool based on java. There are some prerequisites that are required to be installed before installing the Gradle frame work.

Prerequisites to install Gradle

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 are 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, then, that is ignored by Gradle.

The steps to install Gradle in your system are explained below.

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. We have installed JDK 1.8 in the system.

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

Output

The output is as follows:

```
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. We have installed JDK 1.8 in the system.

```
$ java - version
```

Output

The output is mentioned below:

```
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 link available at https://gradle.org/install/. 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 environment means, we have to extract the distribution file and copy the library files into proper location. Set 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.

After that, add the **C:\gradle** and **C:\gradle\bin** directories to the **GRADLE_HOME** and **PATH** system variables.

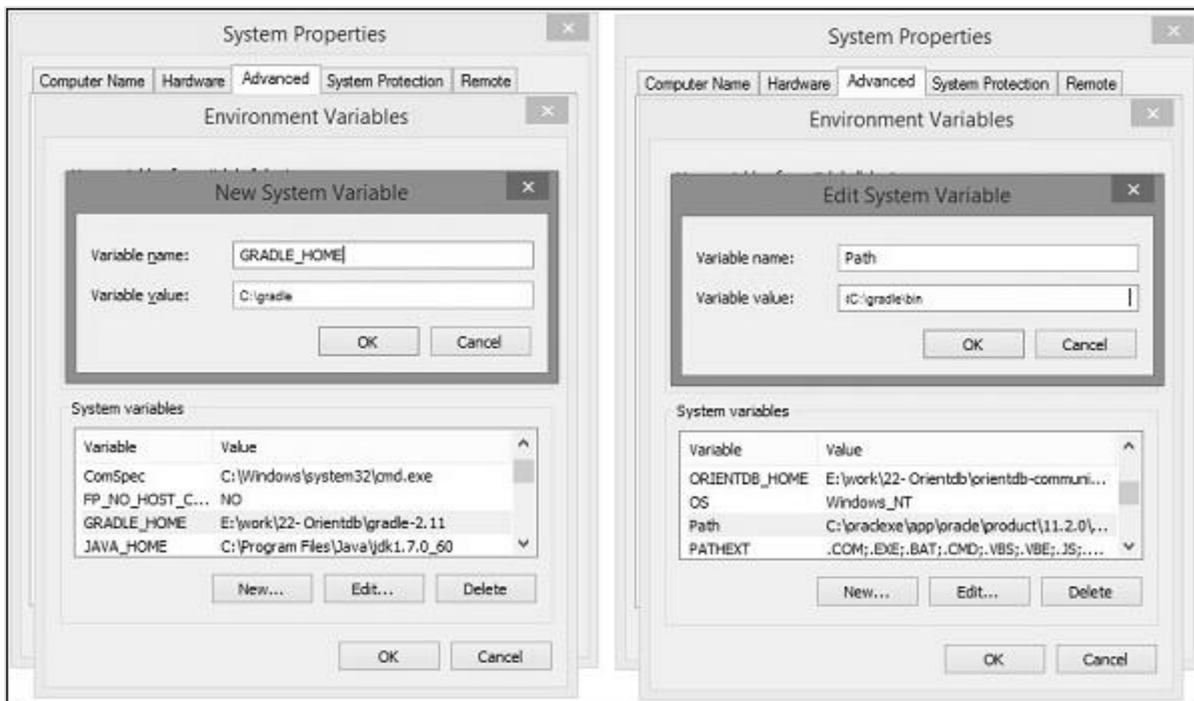
Follow the given instructions, **right click on my computers -> click on properties -> advanced system settings -> click on environmental variables**.

There, you will find a dialog box for creating and editing the system variables.

Click on new button for creating **GRADLE_HOME** variable (follow the left side screenshot).

Click on Edit for editing the existing Path system variable (follow the right side screenshot).

Follow the screenshots given below:



In Linux

Extract the downloaded zip file named **gradle-2.11-all.zip** and 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.

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


tutorialspoint
SIMPLY EASY LEARNING

```
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
```


tutorialspoint
SIMPLYEASYLEARNING

3. Gradle — Build Script

Gradle builds a script file for handling two things; one is **projects** and other 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 is 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 writing builds. The Groovy language is used, in order to, make it easier to describe a build. Each build script of Gradle is encoded using UTF-8, saved offline and is named as build.gradle.

build.gradle

We describe about the tasks and projects by using a Groovy script. You can run a Gradle build using the Gradle command, which 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
```

Output

You will see the following output:

```
tutorialspoint
```


tutorialspoint
SIMPLYEASYLEARNING

If you think, task works similar to ANT's target, then that is correct. **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'
}
```

Now, you can execute the above script using **gradle –q hello** command.

The Grade script mainly uses two real Objects, one is Project Object and the other is Script Object.

- **Project Object** – Each script describes about one or multiple projects. While in the execution, this scripts configures the Project Object. You can execute 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, it is executed. This means that 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.

Groovy Basics

Gradle build scripts use the full length Groovy API.

As a startup, take a look at the following examples.

Example 1

This example explains about converting a string to upper case.

Copy and save the code which is given below, 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 script mentioned above. You should execute this, where the build.gradle file is stored.

```
C:\> gradle –q upper
```

Output

When you run the code, you will see the following output:

```
Original: TUTORIALS point
Upper case: TUTORIALS POINT
```

Example 2

The following example explains about printing the value of an implicit parameter ($it) for four 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 script stated above. You should execute this, where the build.gradle file is stored.

tutorialspoint
SIMPLYEASYLEARNING

```
$ gradle -q count
```

Output

This produces the following output:

```
0 1 2 3
```

Features of Groovy

Groovy language provides plenty of features. Some important features are discussed below:

Groovy JDK Methods

Groovy adds a lot 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 stores.

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

Output

When you execute the above code, you should see the following output:

```
M
a
r
c
```

Property Accessors

You can automatically access 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**.

tutorialspoint
SIMPLYEASYLEARNING

```
// 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, which is that the parentheses are optional for method calling. This feature applies to Gradle scripting as well.

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

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

Closure as the Last Parameter

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 that 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

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

The default import packages to the Gradle script are listed below:

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
```


tutorialspoint
SIMPLYEASYLEARNING

```
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.*
```


tutorialspoint
SIMPLYEASYLEARNING

```
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.*
```


tutorialspoint
SIMPLYEASYLEARNING

```
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.*
```


tutorialspoint
SIMPLYEASYLEARNING

4. Gradle — Tasks

Gradle build script describes about one or more Projects. Each project is made up of different tasks and 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 achieves to the repositories.

This chapter explains about 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 into 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 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
```

Output

Given below is the output of the code:

```
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. It executes the script given above. You should execute this, where the build.gradle file stores.

```
C:\> gradle –q hello
```

Output

The output is shown below:

```
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. It executes the script which is mentioned above. You should execute this, where the build.gradle file stores.

```
C:\> gradle –q hello
```

Output

When you execute the above code, you should see the following output:

```
tutorialspoint
```

You can also use an alternative syntax for defining a task. That is, using create() method to define a task. Take a look into the same hello example which is given below.

Copy and save the below given code into **build.gradle** file.

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

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

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

Output

Upon execution, you will receive the following output:

```
tutorialspoint
```

Locating Tasks

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

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

Copy and save the below given code into **build.gradle** file.

```
task hello


println hello.name

println project.hello.name
```

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

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

Output

The output is mentioned below:

```
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. It executes the script which is mentioned above. You should execute this, where the build.gradle file stores.

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

Output

This produces the following output:

```
hello
hello
```

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

Copy and save the below given 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. It executes the script which is given above. You should execute this, where the build.gradle file stores.

```
C:\> gradle –q hello
```

Output

The output is stated below:

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

Adding Dependencies to Tasks

You can make a task dependent on another task and that means, when one task is done then only other task will begin.

Each task is differentiated with the task name. The 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 below given code into **build.gradle** file. Take a look into the following code.

tutorialspoint
SIMPLYEASYLEARNING

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

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

```
C:\> gradle –q taskY
```

Output

The output is given herewith:

```
taskX
taskY
```

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

Let us take the same example of **taskY** being dependent on **taskX,** but here, 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
```

Output

The output is given below:

```
taskX
taskY
```

The above example is adding dependency on task by using its names.

tutorialspoint
SIMPLYEASYLEARNING

There is another way to achieve task dependency which is, to define dependency using a Task object.

Here, we take the same example that **taskY** is dependent on **taskX** but, we are using task objects instead of task references names.

Copy and save the below given code into **build.gradle** file. Take a look into the following code.

```
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. It executes the above given script. You should execute this, where the **build.gradle** file stores.

```
C:\> gradle –q taskX
```

Output

The output is cited below:

```
lib1
lib2
taskX
```

Adding a Description

You can add a description to your task. This description is displayed when you execute the **Gradle tasks** and this is possible by using, the description keyword.

tutorialspoint
SIMPLYEASYLEARNING

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

```
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, the 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()
   }
```


tutorialspoint
SIMPLYEASYLEARNING

```
}
```

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

```
C:\> gradle –q eclipse
```

Task Structure

Gradle has different phases, when it comes to working with the 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.

tutorialspoint
SIMPLYEASYLEARNING

5. Gradle — Dependency Management

Gradle build script defines a process to build projects; each project contains some dependencies and some publications. Dependencies refer to the things that supports in building 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, build files and war files.

All the projects are not self-contained. They need files which are built by the 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 handles building and publishing the outcomes. Publishing is based on the task that you define. It might want to copy the files to 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. We can call the process of publishing a task as publication.

Declaring Your Dependencies

Dependency configuration defines a set of dependencies. You can use this feature to declare external dependencies, which you want to download from the web. This defines different standers such as follows.

```
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 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.

tutorialspoint
SIMPLYEASYLEARNING

- **Compile** – The dependencies required to compile the production source of the project.
- **Runtime** – The dependencies required by the production classes at runtime. By default, it 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 are one of the type of dependencies. This is a dependency on some files built outside on the current build, and stored in a repository of some kind, such as Maven central, or a corporate Maven or Ivy repository, or a directory I which is 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 the external dependencies and the shortcut form looks like **"group: name: version"**.

Repositories

While adding external dependencies, Gradle looks for them in a repository. A collection of files, organised by group, name and version is termed as a repository. 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

Plugin is nothing but set of 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 means that it allows the plugin to extend the project's capabilities.

The plugins can do the things such as:

- 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 organisational repositories or enforce standards).

Types of Plugins

There are two types of plugins in Gradle, which are as follows:

- **Script 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:** Binary plugins are the classes, that implements 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 specify 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 plugin id. Some core plugins use short names to apply the plugin id and some community plugins use fully qualified name for plugin id. Sometimes, it allows to specify the class of plugin.

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

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

Take a look into the following code for applying community plugin using 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 example contains a greeting plugin, which adds a hello task to the project. Take a look into the following code and 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
```


tutorialspoint
SIMPLYEASYLEARNING

Output

This produces 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
```

Output

When you run the code, you will see the following output:

tutorialspoint
SIMPLYEASYLEARNING

```
Hi from Gradle
```

In this example, GreetingPlugin is a simple, 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
```

Output

The output is mentioned below:

```
Hello from Gradle
```


tutorialspoint
SIMPLYEASYLEARNING

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.

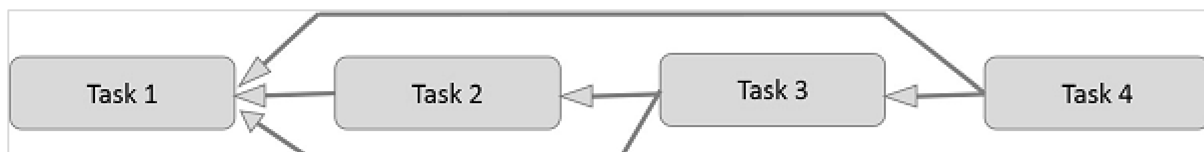
7. Gradle - Running a Build

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 the build file using **gradle command**. This command will compile each task in such an 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 4 tasks are dependent on each other represented with an arrow symbol. Take a look into the following code. Copy can 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 above task.

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

Output

tutorialspoint
SIMPLYEASYLEARNING

The output is stated below:

```
: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
```

Output

Cited below is the output of the code:

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

BUILD SUCCESSFUL

Total time: 1 secs
```

Continuing the Build

Gradle will abort 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 dependences.

tutorialspoint
SIMPLYEASYLEARNING

The main point is that 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 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 absolute path.

The following example selects 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
```

Output

This produces 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, the dependencies of your build and for debugging the 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. Use the following command to list all the project in the build file. Here is the example,

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

Output

The output is stated below:

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


tutorialspoint
SIMPLYEASYLEARNING

```
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
```

Output

The output is given herewith:

```
------------------------------------------------------------
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]
```


tutorialspoint
SIMPLYEASYLEARNING

```
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
```

Output

When you execute the above code, you should see 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)
```


tutorialspoint
SIMPLYEASYLEARNING

```
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'.
```

The list of commands is given below along with the description.

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.

tutorialspoint
SIMPLYEASYLEARNING

8. Gradle—Build a JAVA Project

This chapter explains how to build a java project using Gradle build file.

First of all, we have to add java plugin to the build script, because, it provides the tasks to compile Java source code, to run the unit tests, to create a Javadoc and to 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 assumes a certain setup of your Java project (similar to Maven). Take a look into 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 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
```

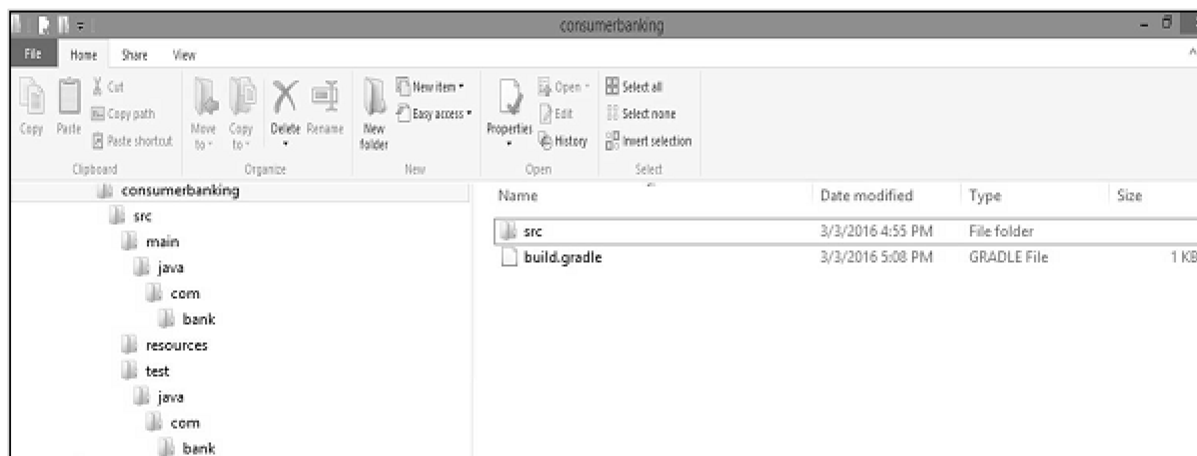
If the artifact is an executable Java application, the **MANIFEST.MF** file must be aware of the class with the main method.

```
apply plugin: 'java'



jar {
   manifest {
```

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

Example

Create a directory structure as shown in the below screenshot.



Copy the below given java code into App.java file and store 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 below given 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 below given 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 below given 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.

tutorialspoint
SIMPLYEASYLEARNING

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'
```

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

```
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

The Groovy plugin assumes a certain setup of the 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 where **build.gradle** file places for build folder.

tutorialspoint
SIMPLYEASYLEARNING

10. Gradle — Testing

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

The test task provides a **Test.getDebug()** method which can be set in order to launch, so that the JVM can 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 and only those classes will be scanned.

Depending on the test framework used (JUnit / TestNG), the test class detection uses the 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 the test class detection, you can disable it by setting **scanForTestClasses** to **false**.

Test Grouping

JUnit and TestNG allows sophisticated grouping 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, which 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'
   }
}
```


tutorialspoint
SIMPLYEASYLEARNING

Include and Exclude Tests

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

Use the below mentioned code to run only the included tests:

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

Use the code given below to skip the excluded tests:

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

The sample **build.gradle** file as stated below, 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, which are as follows:

- 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
```

Output

You will receive 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'
```

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.

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, usually, the ui project 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.

Maven-publish Plugin

By default, **maven-publish** plugin is provided by Gradle. It is used to publish the gradle script. Take a look into 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'
```


tutorialspoint
SIMPLYEASYLEARNING

```
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 from Maven to Gradle

There is a special command for converting Apache Maven **pom.xml** files to Gradle build files and is executed, if all the 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 into it.

```
<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>
```


tutorialspoint
SIMPLYEASYLEARNING

```
  <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 one mentioned below:

```
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 {
```


tutorialspoint
SIMPLYEASYLEARNING

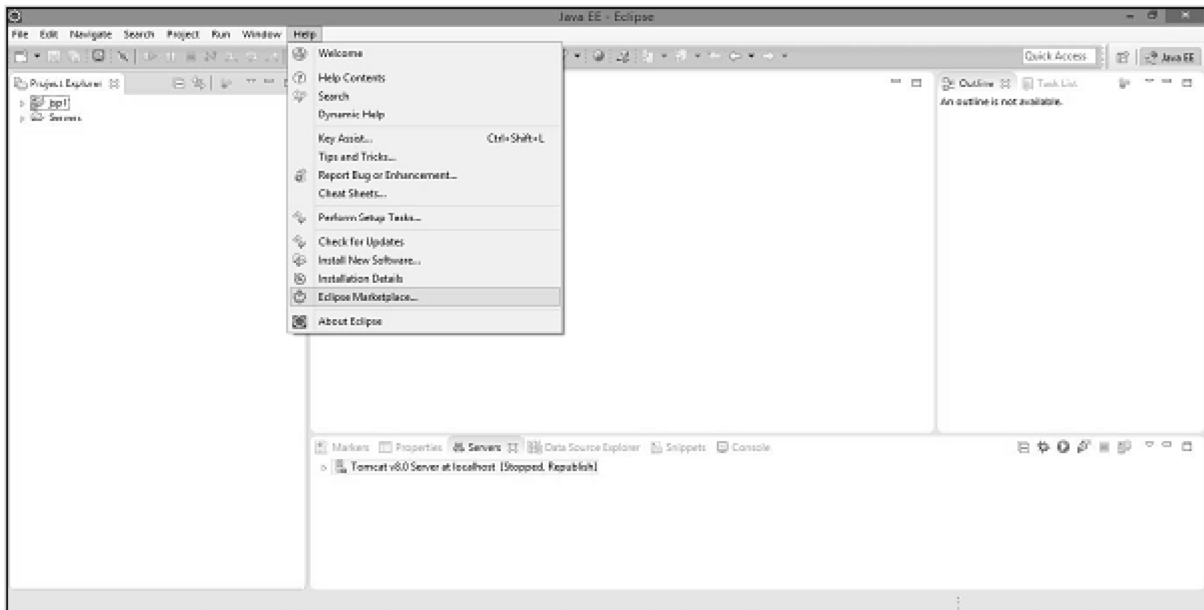
```
    testCompile group: 'junit', name: 'junit', version:'4.11'
}
```

13. Gradle — Eclipse Integration

This chapter explains about the integration of eclipse and Gradle. Follow the below given steps for adding Gradle plugin to eclipse.

Step 1 − Open Eclipse Marketplace

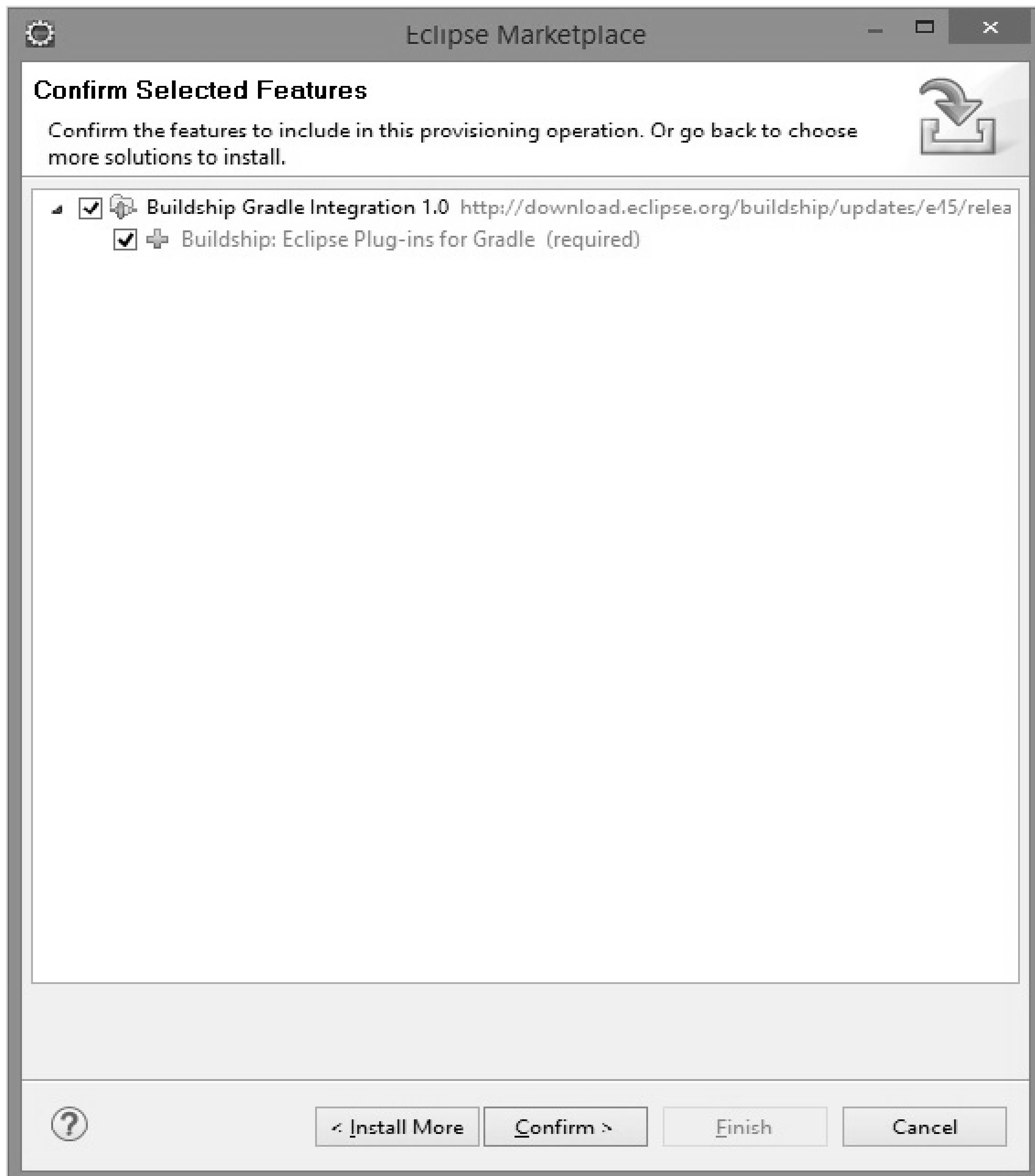
First of all, open the eclipse which is installed in your system. Go to help -> click on **EclipseMarketplace**. Take a look at the following screenshot:



Step 2 − Install Buildship Plugin

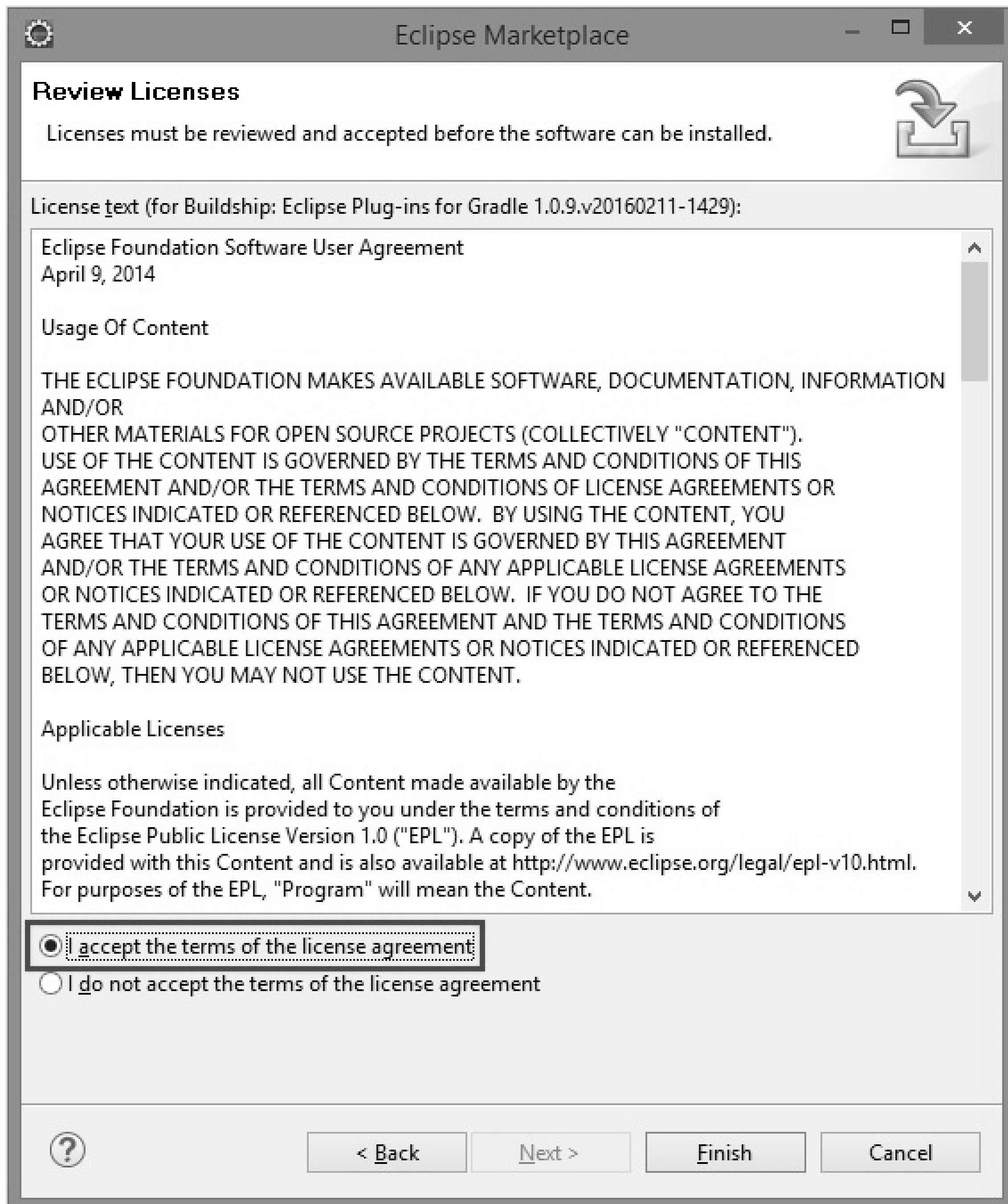
After click on the Eclipse Marketplace, you will find the screenshot which is given below. Here, in the left side search bar type **buildship**, which is a Gradle integration plugin. When you find the buildship on your screen, click on install on the right side.

After that, you will find the following screenshot. Here, you need to confirm the software installation by clicking on the confirm button.

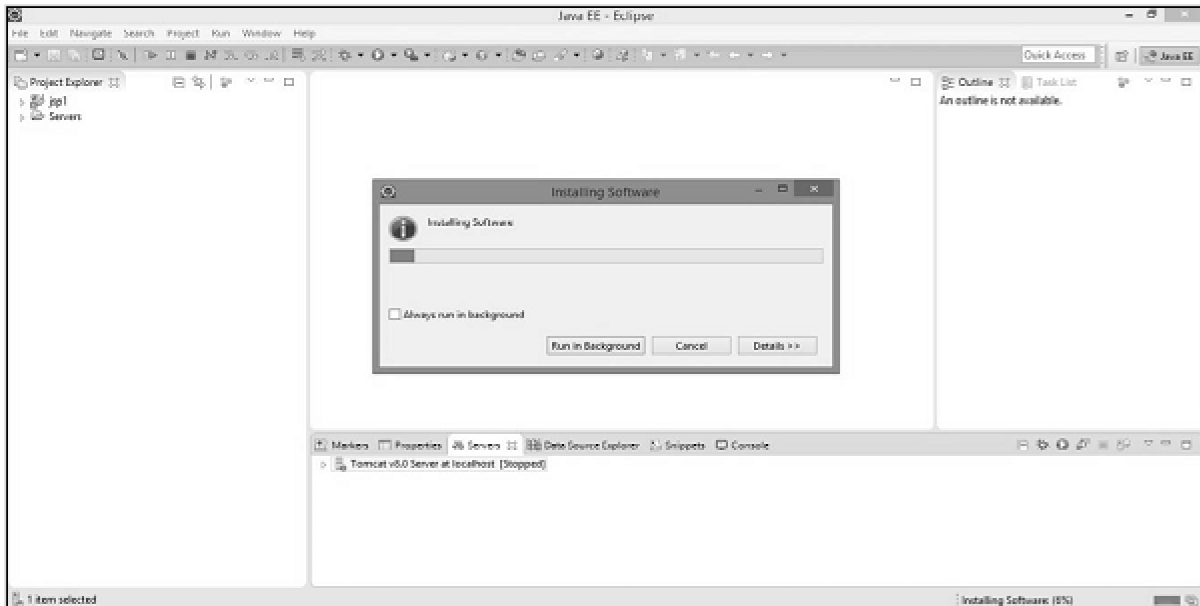


Then, you need to click on **accept license agreement** as shown in the following screen and later on click **finish**.

tutorialspoint
SIMPLYEASYLEARNING



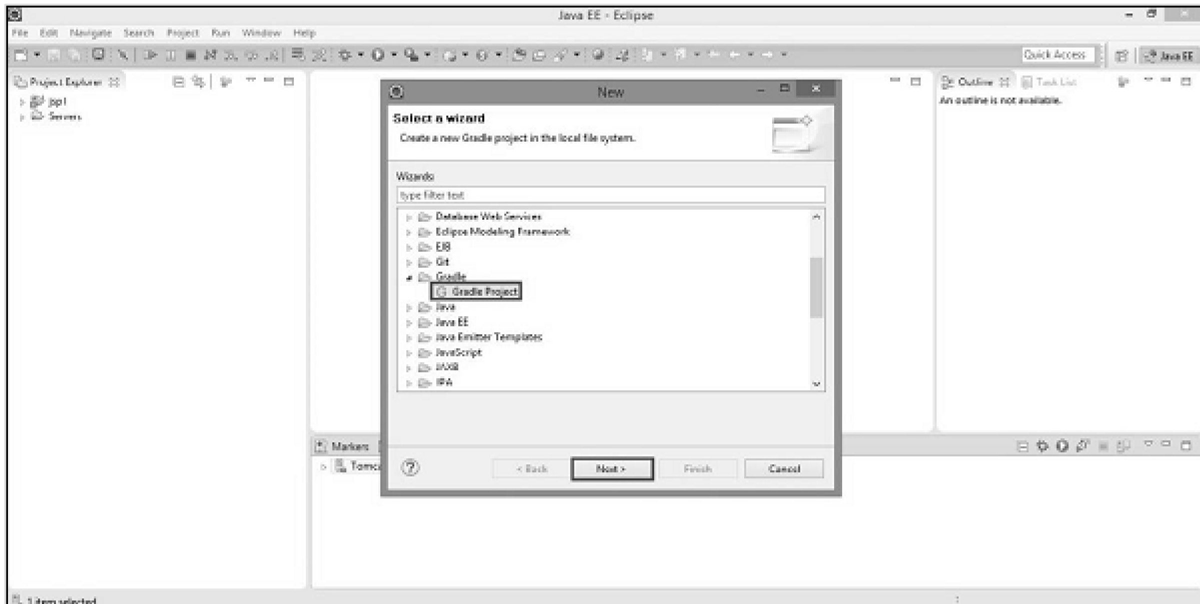
It will take some time to install. Refer the screenshot given below for detailed understanding.



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

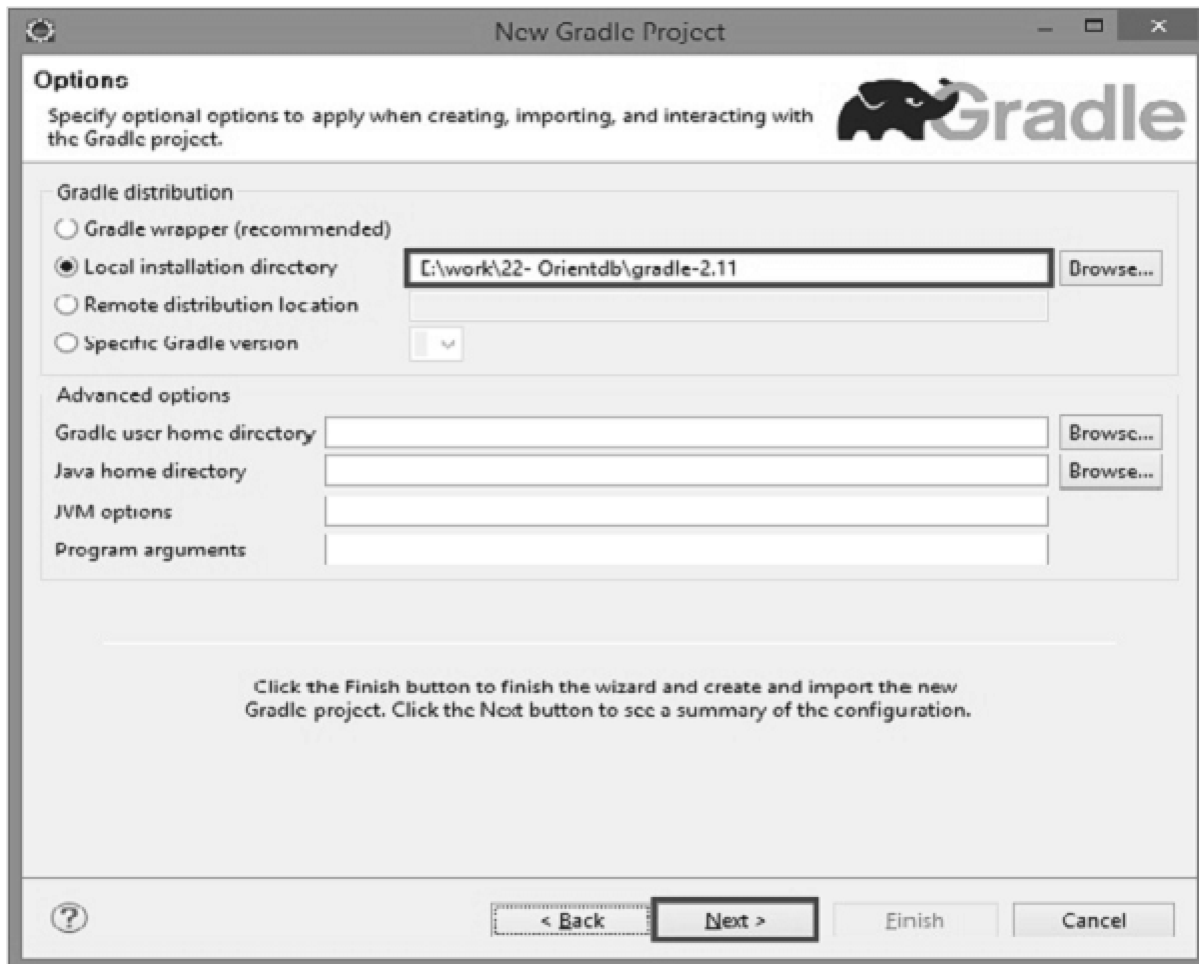
Step 3 – Verifying Gradle Plugin

While verifying, we will create a new project by following the given procedure. In the eclipse, go to file **-> click on new-> click on other projects**. Now, you will see the following screen. Later on, select Gradle project and click next. Refer the below mentioned screen shot.



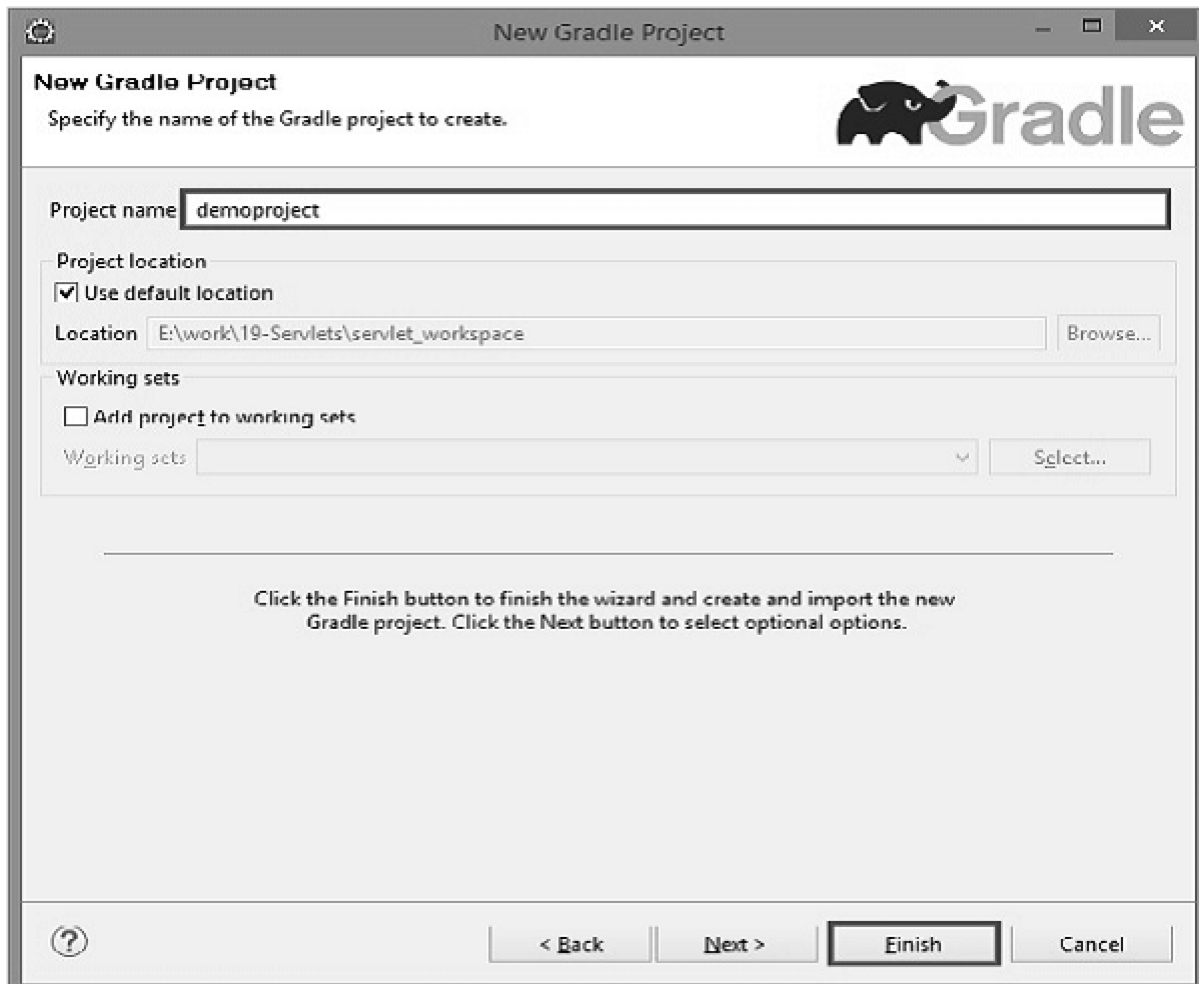
After clicking next button, you will find the following screen. After that, you will provide the Gradle home directory path of local file system and click on next button. The screenshot is given below:

tutorialspoint
SIMPLY EASY LEARNING



You will have to provide the name for Gradle project. In this tutorial, we are using **demoproject** and click **finish** button. The screenshot is mentioned below:

tutorialspoint
SIMPLY EASY LEARNING

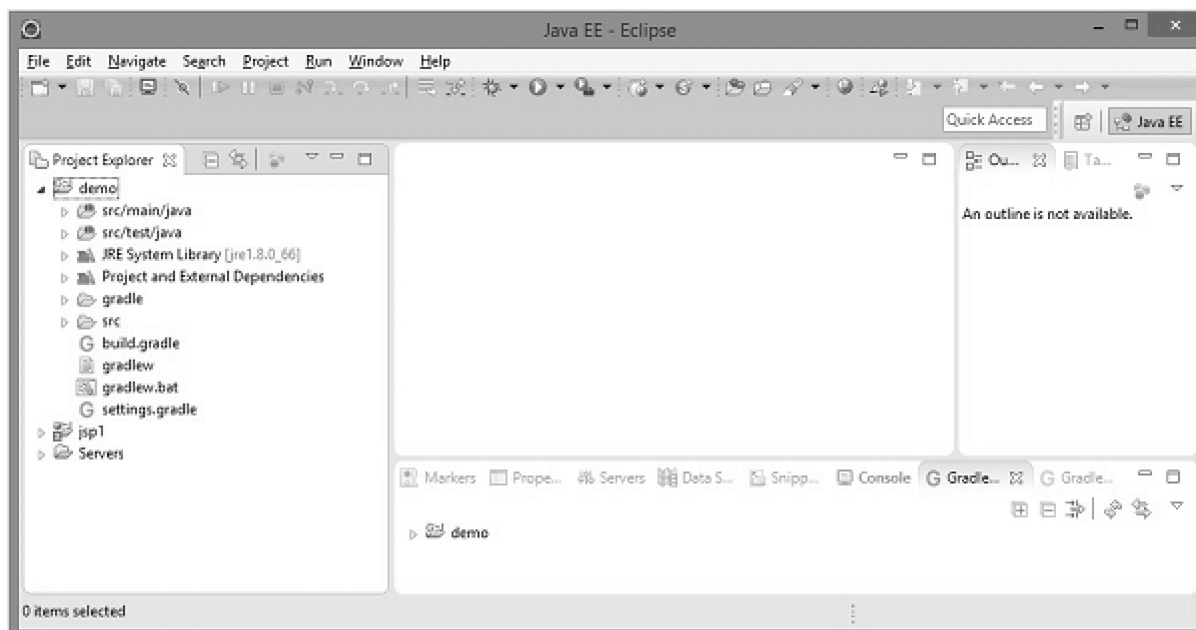


We need to confirm the project. For that, we have to click finish button in the following screen.

tutorialspoint
SIMPLYEASYLEARNING

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.



tutorialspoint
SIMPLY EASY LEARNING