Gradle

COURSE MATERIAL

By

Mr.Natraj sir

NARESHØ™
technologies
An ISO 9001 : 2008 Certified Company

Sri Raghavendra Xerox
Near sathyam theatre Ameerpet Hyd
CELL : 9951596199

Gradle – General Introduction

Introduction

Gradle is an open source, advanced general purpose build management system. It is built on ANT, Maven, and Ivy repositories. It supports Groovy based Domain Specific Language (DSL) over XML. This Document explains how you can use Gradle as a build automation tool for Java as well as Groovy projects.

Prerequisites

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

Gradle VS Maven VS ANT

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.

Gradle - Overview

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
- **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).

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

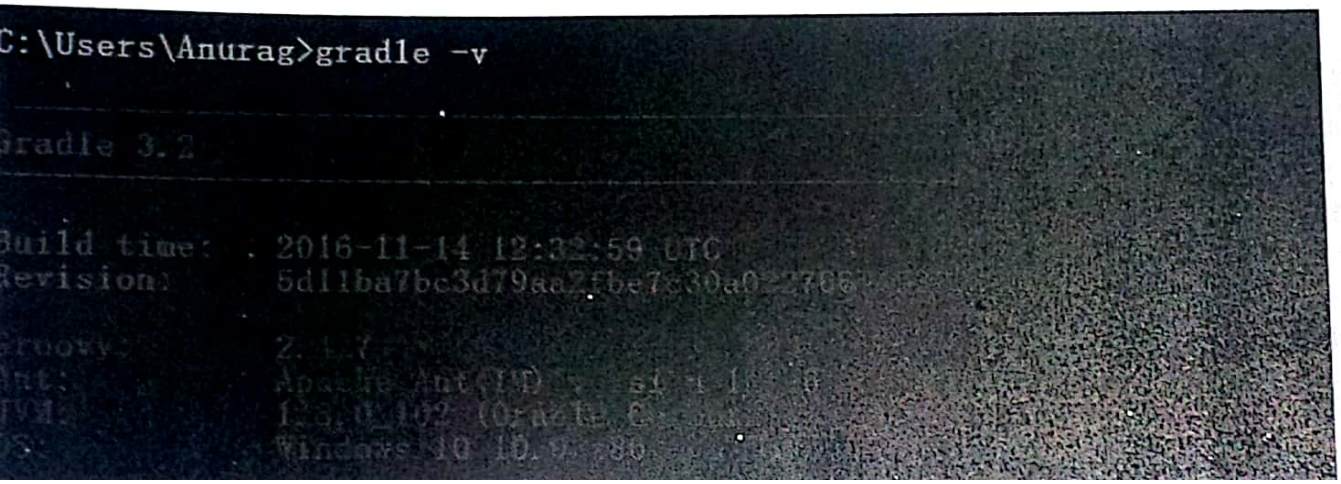
First of all, you need to have Java Software Development Kit (SDK) installed on your system. To verify this, execute **Java –version** command in any of the platform you are working on.

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.

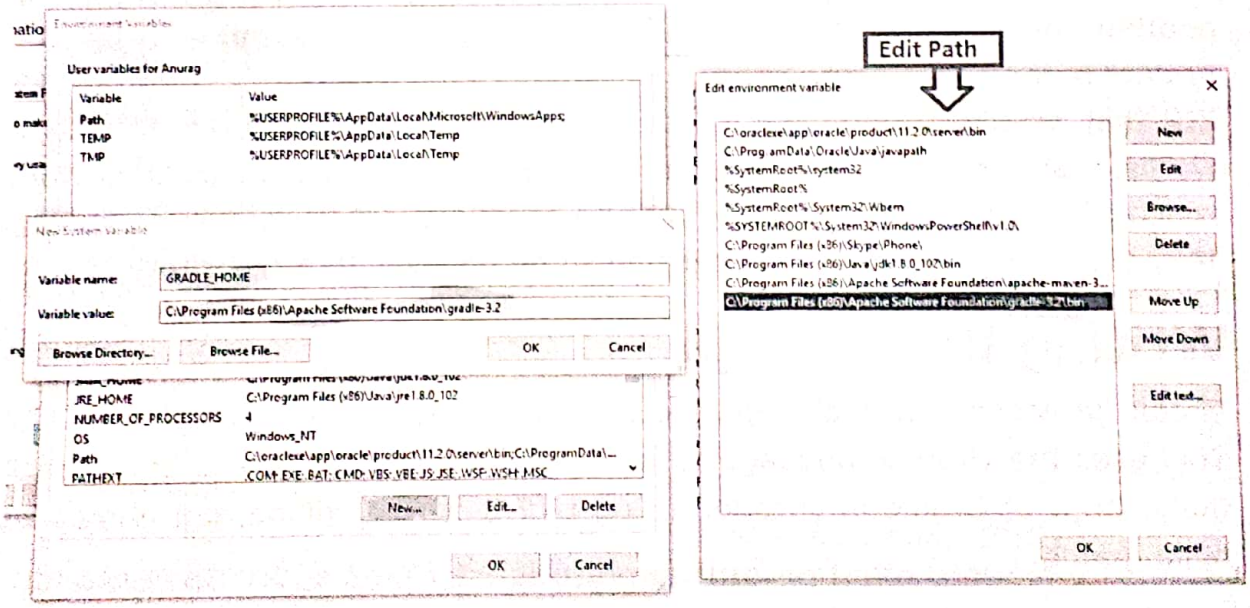


In Windows –

Extract the downloaded zip file named **gradle-3.2-all.zip** and copy the distribution files from **Downloads\gradle-3.2** to **C:\Program Files (x86)\Apache Software Foundation\gradle-3.2.**

Later, add the **C:\Program Files (x86)\Apache Software Foundation\gradle-3.2.**and **C:\Program Files (x86)\Apache Software Foundation\gradle-3.2\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.



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 **Natraj Sir**. 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 **Natraj Sir** string.

```
task hello {
   doLast {
      println 'Natraj Sir '
}
}
```

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.

The Gradle script mainly uses **two real Objects**

- ✓ **ProjectObject**
- ✓ **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.

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

```
task upper <<{
StringexpString='NATRAJ sir'
   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.

```
C:\Users\Anurag\Desktop\Gradle>gradle -q upper
Original: NATRAJ sir
Upper case: NATRAJ SIR
C:\Users\Anurag\Desktop\Gradle>
```

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

gradle local repository is available in c:\users\<windows user>\.gradle file ..... all jars, plugins of gradle will be downloaded to

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

```
task count <<{
4.times{
print"@nu"
}
}
```

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 count
```

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

```
C:\Users\Anurag\Desktop\Gradle>gradle -q count
@nu@nu@nu@nu
C:\Users\Anurag\Desktop\Gradle>
```

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="Natraj";
myName.each(){
println"Nt"
};
```


```
}
```

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.

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**.

```
// 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" })
```

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 **Natraj Sir**. 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 Natraj Sir string.


```
task hello {
   doLast {
      println ' Natraj Sir '
}
}
```

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.

```
C:\Users\Anurag\Desktop\Gradle>gradle -q hello
C:\Users\Anurag\Desktop\Gradle>
```

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 same output as above output.

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 " Natraj Sir "
}
```

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 same output as above output.

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.

```
C:\Users\Anurag\Desktop\Gradle>gradle -q hello
hello
hello
C:\Users\Anurag\Desktop\Gradle>
```

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.

```
C:\Users\Anurag\Desktop\Gradle>gradle -q hello
hello
hello
C:\Users\Anurag\Desktop\Gradle>
```

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.

```
C:\Users\Anurag\Desktop\Gradle>gradle -q taskY
taskX
taskY
C:\Users\Anurag\Desktop\Gradle>
```

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.

```
C:\Users\Anurag\Desktop\Gradle>gradle -q taskY
taskX
taskY
C:\Users\Anurag\Desktop\Gradle>
```

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
```

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'
```


```
}
```

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"
}
}
```

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"
}
```

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.

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'
}
}
}
```

Example

DemoApp.java (In src/main/java folder)

```
package com.nt.gradle;
public class DemoApp
{
        public static void main(String args[]){
        System.out.println("hello"+ args[0]);
        }
}
```

build.gradle

```
apply plugin: 'java'

task task1( type: JavaExec) {
   main = 'com.nt.gradle.DemoApp'
   classpath = sourceSets.main.runtimeClasspath
   args 'raja'
 }
defaultTasks 'task1'
```

build.gradle (with dependencies)

```
apply plugin: 'java'
 repositories{
   mavenCentral()
 }
dependencies{
  // https://mvnrepository.com/artifact/log4j/log4j
   //compile group: 'log4j', name: 'log4j', version: '1.2.12'
    compile  'log4j:log4j:1.2.12'
}
task task1( type: JavaExec) {
    main = 'com.nt.gradle.DemoApp'
    classpath = sourceSets.main.runtimeClasspath
    args 'raja'
   }
 defaultTasks 'task1'
```

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.

cmd>gradle init –type=java-library [It is alternate to archetype-quick-st of maven]

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.

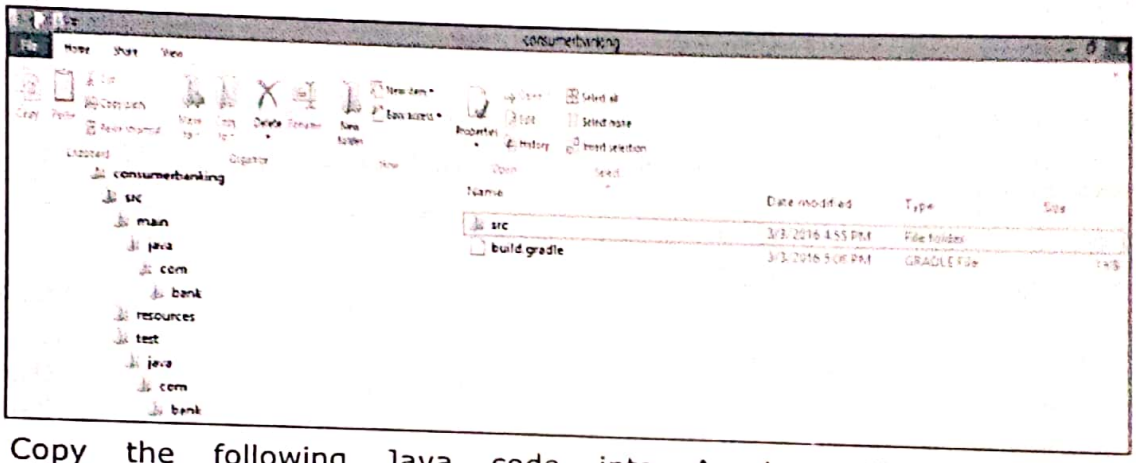
```
version = 0.1.0
sourceCompatibility = 1.8
```

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.



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

```
packagecom.bank;

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.

```
packagecom.bank;

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\libs** folder for **consumerbanking.jar** file.

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.

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.

```
<projectxmlns="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>
```

1) Create Directory structure

src
 |-> main
  |-> java
   |-> com
    |-> nt
     |-> servlet
      |-> FormServlet.java
  |-> webapp
   |-> form.html
   |-> WEB-INF
    |-> web.xml

```
<version>1.0.0-SNAPSHOT</version>

<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>

<version>4.11</version>
<scope>test</scope>
</dependency>
</dependencies>

</project>
```

2) Develop

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 ="""""
```

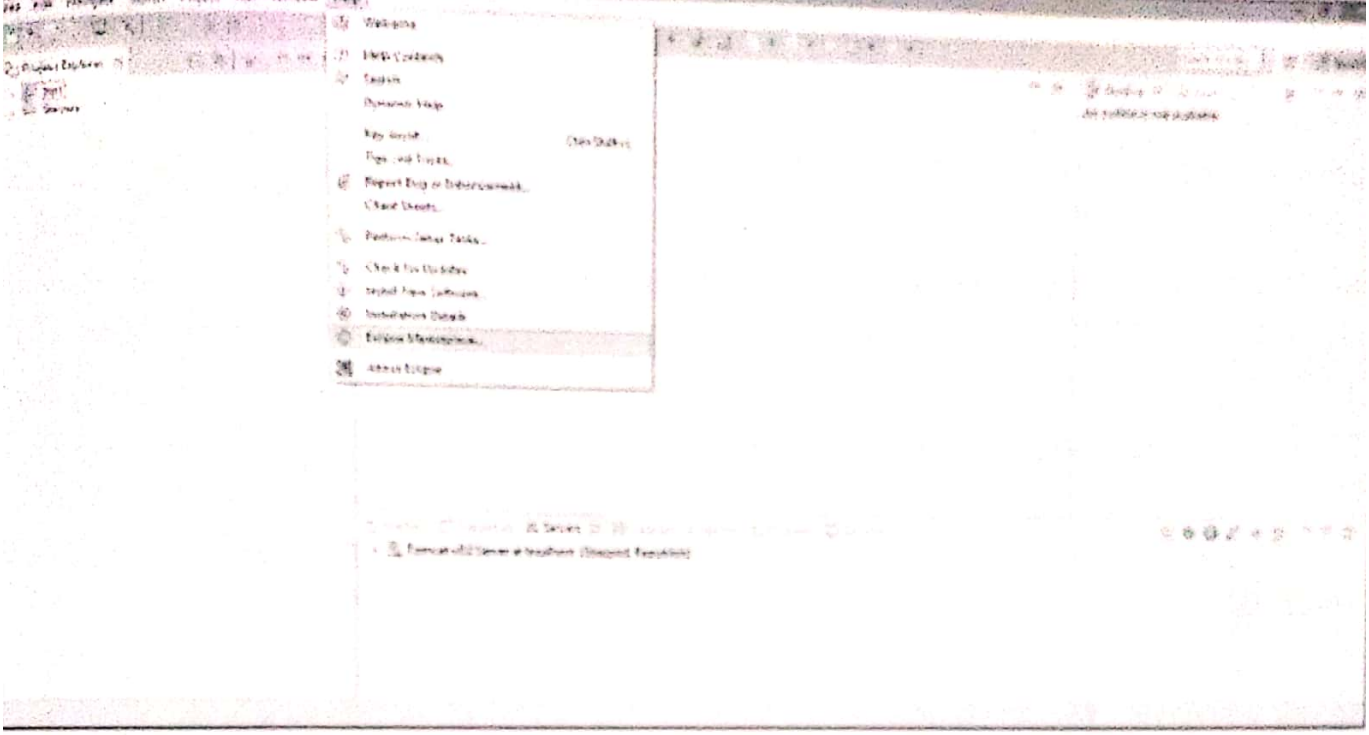

```
sourceCompatibility =1.5
targetCompatibility =1.5

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

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

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. (the left search bar, type **buildship**. Buildship is a Gradle integration plug When you find buildship on your screen, click Install button present on t right side of the screen as shown in the following screenshot.

Eclipse Marketplace

Eclipse Marketplace

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.

Search | Recent | Popular | Installed | February Newsletter

Find: buildship | All Markets | All Categories | Go

Buildship Gradle Integration 1.0

Gradle

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

★ 13 | Installs: 29.1K (3,832 last month) | Install

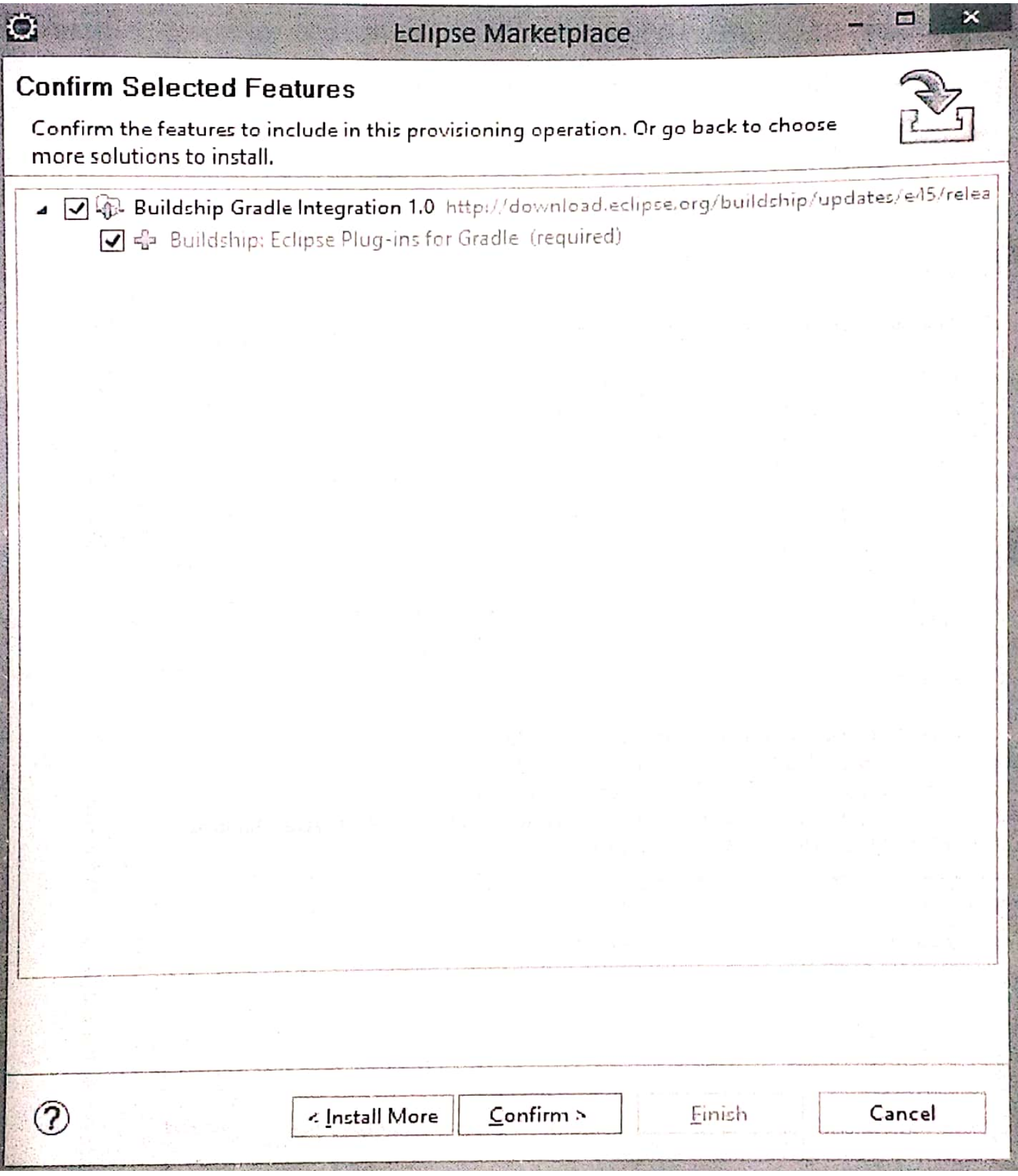
Minimalist Gradle Editor 1.0.1

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

by Nodorbase Studio, GPL

Marketplaces

< Back | Install Now > | Finish | Cancel



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

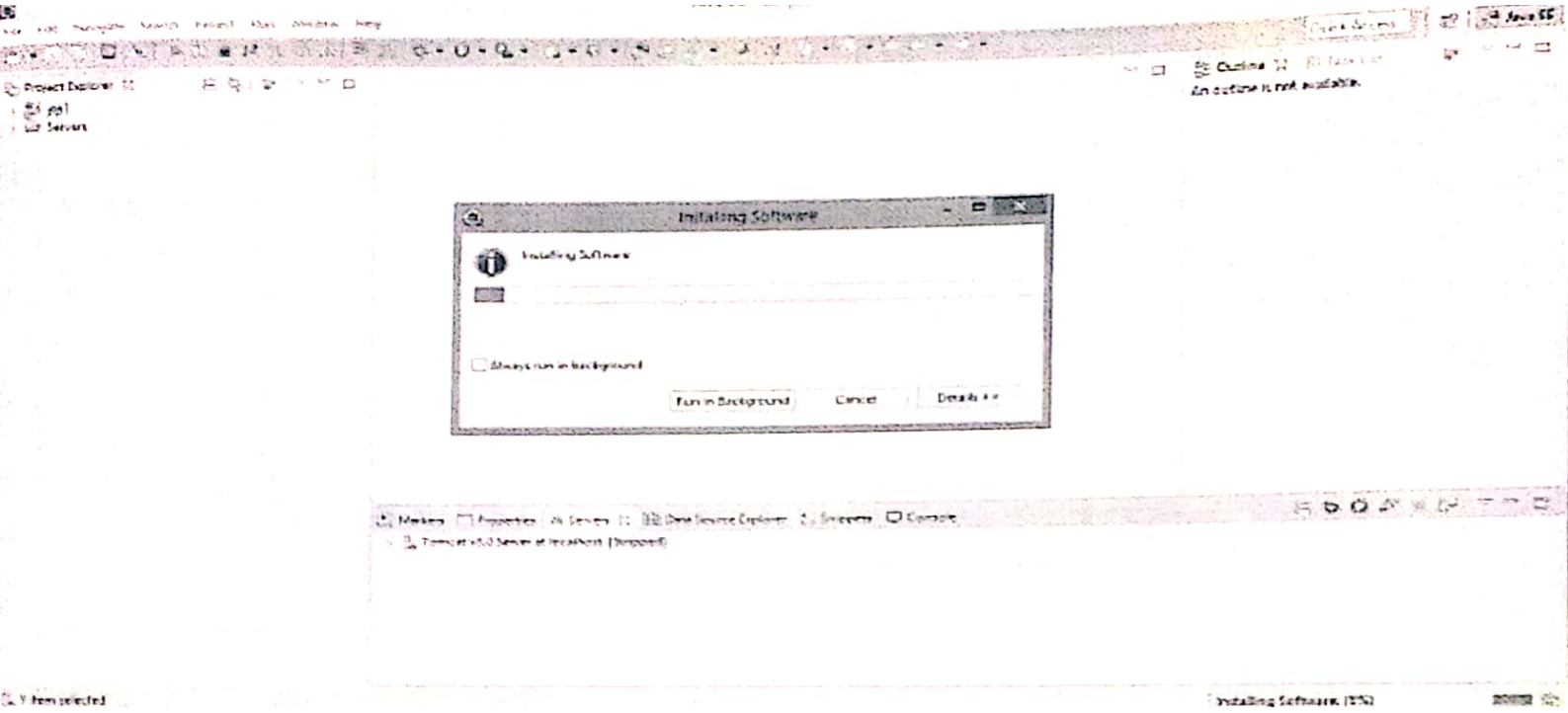
THE ECLIPSE FOUNDATION MAKES AVAILABLE SOFTWARE, DOCUMENTATION, INFORMATION AND/OR
OTHER MATERIALS FOR OPEN SOURCE PROJECTS (COLLECTIVELY "CONTENT").
USE OF THE CONTENT IS GOVERNED BY THE TERMS AND CONDITIONS OF THIS
AGREEMENT AND/OR THE TERMS AND CONDITIONS OF LICENSE AGREEMENTS OR
NOTICES INDICATED 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.

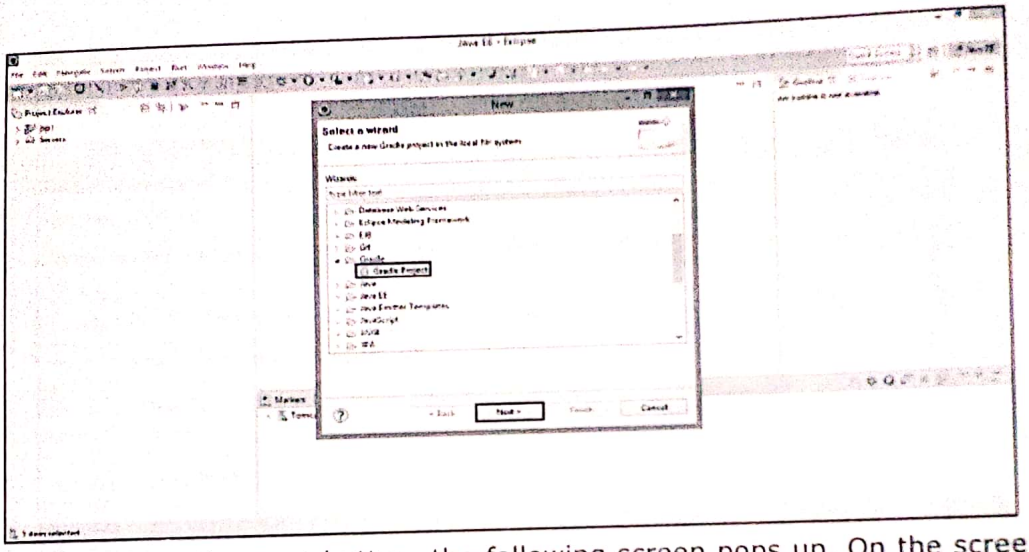
Applicable Licenses

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.

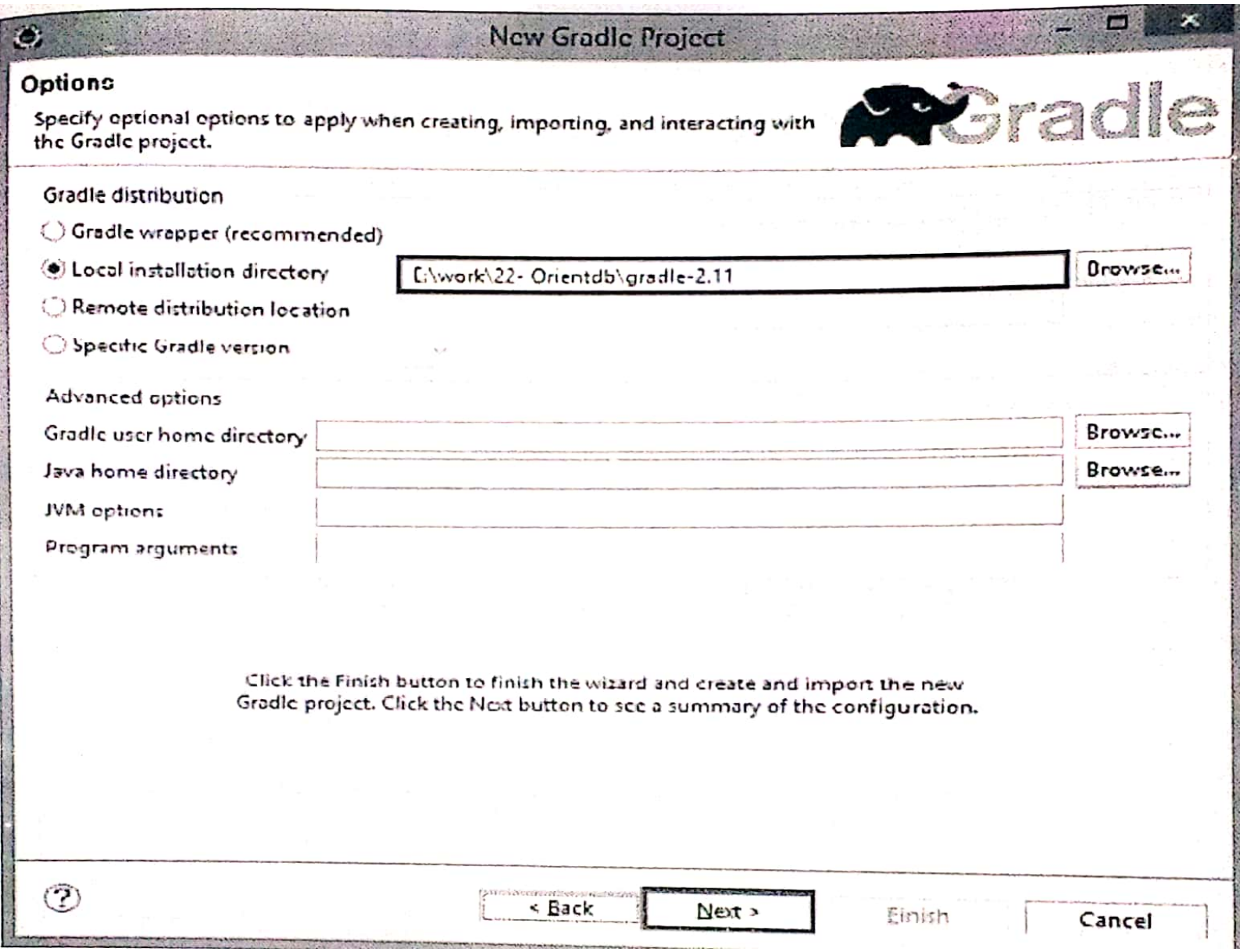
(•) I accept the terms of the license agreement
( ) I do not accept the terms of the license agreement

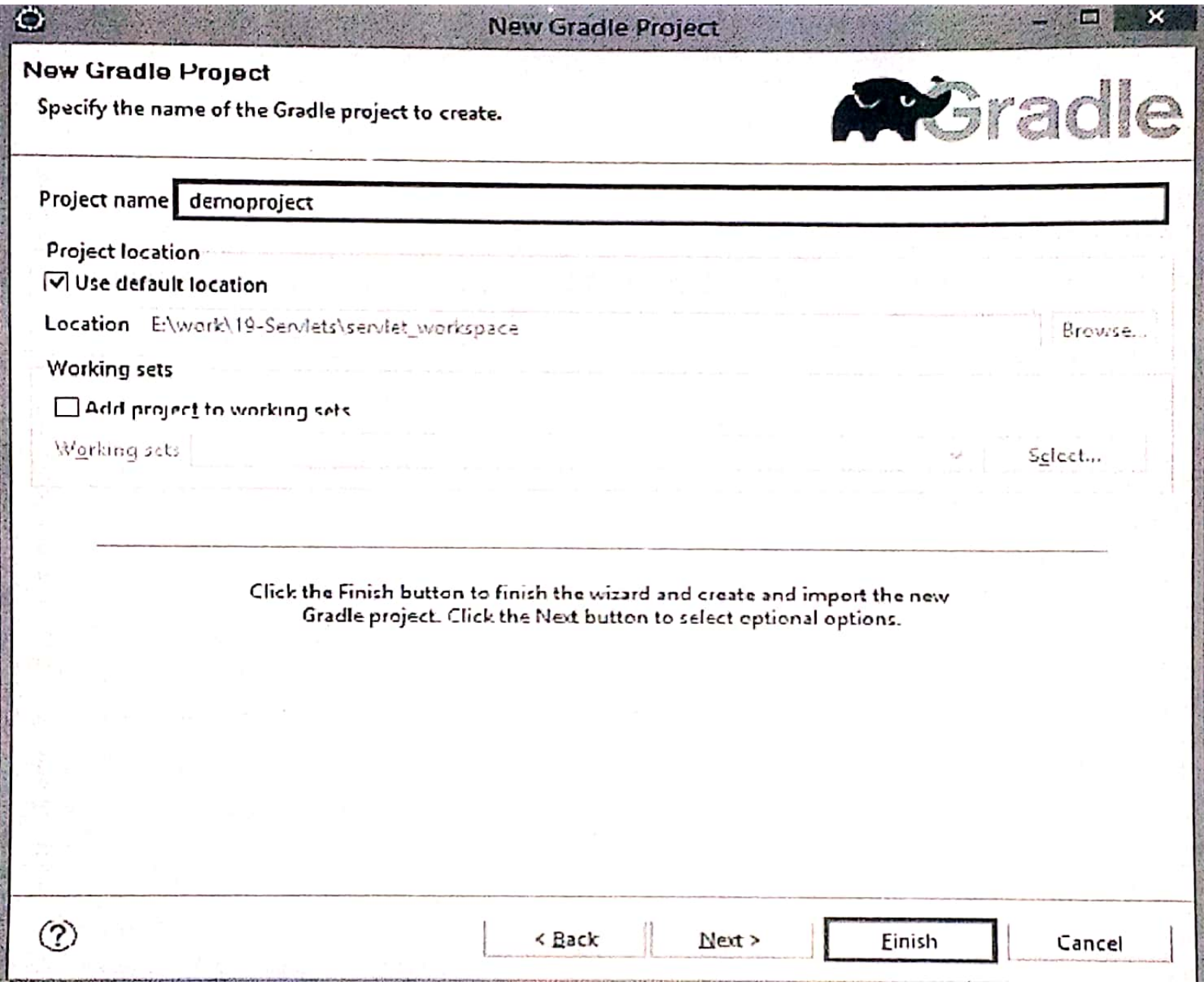
< Back | Next > | Finish | Cancel

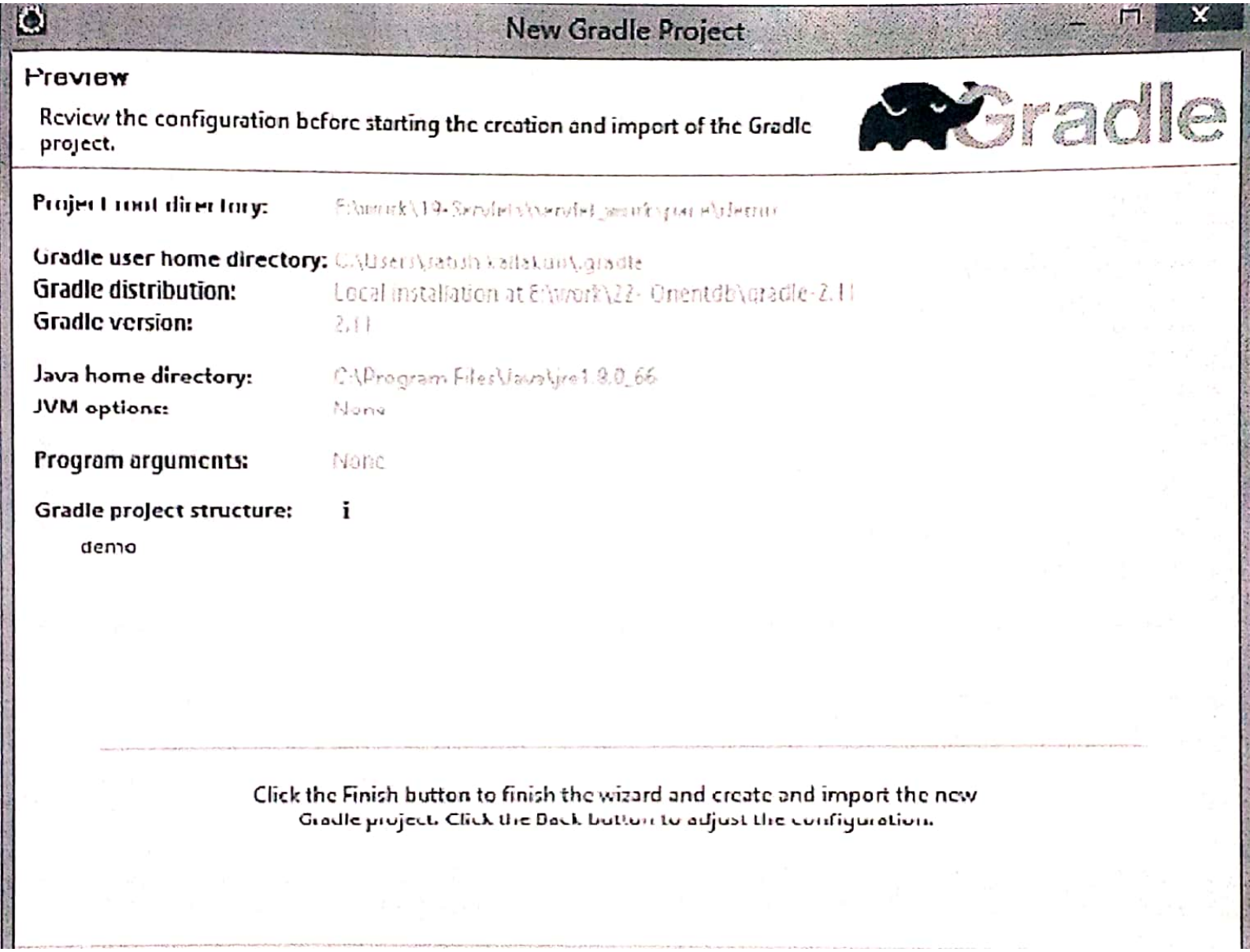


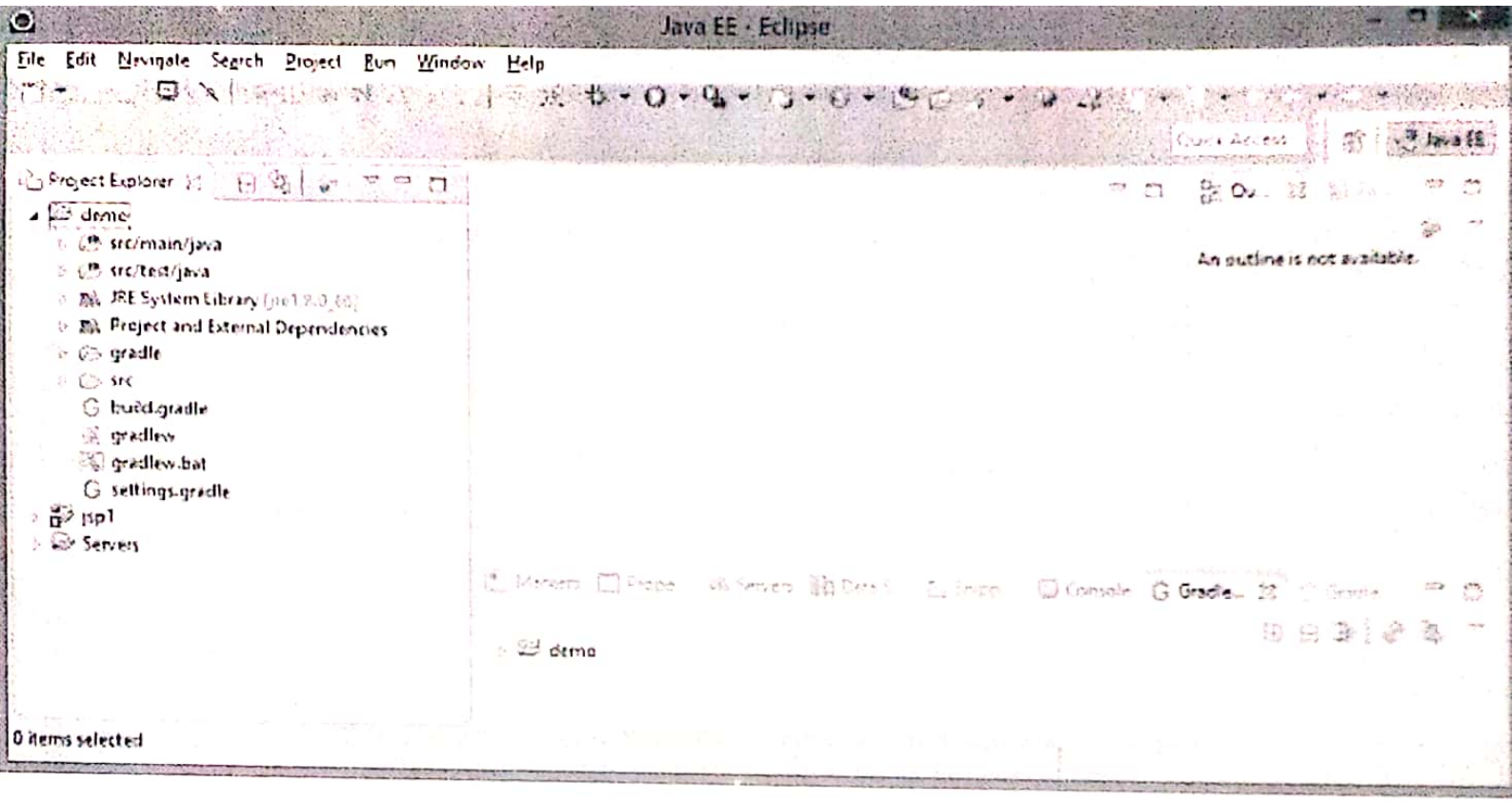


After clicking the Next button, the following screen pops up. On the screen, you will have to provide the Gradle home directory path of local file system and then click Next button as shown in the following screenshot.









Creating a Web Application Project with

We can create a Java project by applying the Java plugin. We can do this by adding the following line to the *build.gradle* file:

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

Packaging Our Web Application

Before we can package our web application by using the War plugin, we have to add it to our build. After we have applied the War plugin, the *build.gradle* file looks as follows:

```
1   apply plugin: 'java'
2   apply plugin: 'war'
```

The War plugin adds a new directory to the project's directory layout, adds two new dependency management configurations, and adds a new task to our project. These changes are described in the following:

- The War plugin adds the *src/main/webapp* directory to the project's directory layout. This directory contains the sources of the web application (CSS files, Javascript files, JSP files, and so on).
- The War plugin adds two new dependency management configurations called *providedCompile* and *providedRuntime*. These two two configurations have the same scope than the *compile* and *runtime* configurations, but the difference is that that the dependencies belonging to these new configurations are not added to the WAR archive.
- The War plugin also adds the *war* task to our web application project. This task assembles a WAR archive to the *build/libs* directory.

We can now package our web application by running the command *gradle war* at command prompt. When we do this, we should see the following output:

```
> gradle war
:compileJava
:processResources
:classes
:war

BUILD SUCCESSFUL
```


```
Total time: 4.937 secs
```

Let's find out how we can run our web application in a development environment.

Running Our Web Application

We can run our web application in a development environment by using Gretty. It supports both Jetty and Tomcat, and it doesn't suffer from the problem caused by Gradle's leaking SLF4J bindings. Let's move on and configure our build to run our web application with Gretty.

First, we have to configure the dependencies of our build script. We can do this by following these steps:

1. Configure the build script to use the Bintray'sJCenter Maven repository when it resolves its dependencies.
2. Add the Gretty plugin dependency to the classpath of the build script.

The source code of the *build.gradle* file looks as follows:

```
1
2   buildscript {
3       repositories {
4           jcenter()
5       }
6       dependencies {
7           classpath 'org.akhikhl.gretty:gretty:+'
8       }
9   }
10
11  apply plugin: 'java'
12  apply plugin: 'war'
```

Second, we have to apply the Gretty plugin. After we have done this, the *build.gradle* file looks as follows:

```
1   buildscript {
2       repositories {
3           jcenter()
4       }
5       dependencies {
6           classpath 'org.akhikhl.gretty:gretty:+'
7       }
8   }
9
10  apply plugin: 'java'
```


```
10  apply plugin: 'war'
11  apply plugin: 'org.akhikhl.gretty'
12
```

Third, we need to configure Gretty by following these steps:

1. Configure Gretty to use Jetty 9 as a servlet container when it runs our web application.
2. Configure Jetty to listen to port **2626.**
3. Configure Jetty to run our web application by using the context path '/'.

The source code of the *build.gradle* file looks as follows:

```
1   buildscript {
4       repositories {
5           jcenter()
6       }
7       dependencies {
8           classpath 'org.akhikhl.gretty:gretty:+'
9       }
10  }
11
12  apply plugin: 'java'
13  apply plugin: 'war'
14  apply plugin: 'org.akhikhl.gretty'
15
16  gretty {
17      port = 2626
18      contextPath = '/'
19      servletContainer = 'jetty9'
20  }
```

We can now start and stop our web application by running the following commands at command prompt:

- the command ***gradle appStart*** will run our web application.
- The command ***gradle appStop*** will stop our web application.